Federico Busato

High-Performance and Power-Aware Graph Processing on GPUs

Ph.D. Thesis

January 13, 2018

Università degli Studi di Verona
Dipartimento di Informatica

Advisor:
prof. Nicola Bombieri

Series N°: **???? (ask the PhD coordinator!)**

Università di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona
Italy

Contents

1 **Thesis Abstract** 1

Part I Background

2 **Graphic Processing Unit (GPU)** 11
2.1 Computed Unified Device Architecture (CUDA) 11
2.2 Modern GPU architectures 13

3 **Static Graph Representation** 15
3.0.1 Adjacency Matrices 15
3.0.2 Adjacency Lists 16
3.0.3 Edges List 18

Part II Performance-Oriented Implementations of Graph Algorithms for GPU

4 **Related Work** 25
4.1 The workload partitioning problem in GPUs 25
4.1.1 Static mapping techniques 26
4.1.2 Semi-dynamic mapping techniques 28
4.1.3 Dynamic mapping techniques 31
4.2 Graph Traversal and Breadth-First Search 34
4.3 Single-Source Shortest Path 35
4.4 Strongly Connected Components 36
4.4.1 Forward-Backward algorithm 37
4.4.2 Coloring algorithm 37
4.4.3 Other algorithms 38

5 **Load Balancing - Multi-Phase Search Algorithm** 41
5.1 Introduction 41
5.2 The proposed Multi-phase Mapping 43
5.2.1 Hybrid partitioning 43

5.2.2 Iterative Coalesced Expansion 45
5.2.3 Optimizing the Multi-Phase implementation 47
5.3 Comprehensive comparison of complexity and limiting factors of the approaches 49
5.4 Experimental results 52
5.4.1 *Multi-phase Mapping* Analisys 58
5.5 Conclusions 60

6 Breadth-First Search - BFS-4K 61
6.1 Introduction 61
6.1.1 Breadth First Search (BFS) 63
6.2 BFS-4K Overview 63
6.3 Implementation Features in Details 65
6.3.1 Exclusive Prefix-Sum 65
6.3.2 Dynamic Virtual Warps 66
6.3.3 Dynamic Parallelism 67
6.3.4 Edge-discover 69
6.3.5 Single-block vs. Multi-block Kernel 71
6.3.6 Coalesced Read/Write Memory Accesses 72
6.4 Duplicate Detection and Correction 74
6.5 Experimental results 77
6.6 Concluding remarks 82

7 Breadth-First Search - Helix 83
7.1 Introduction 83
7.2 Parallel graph traversal through BFS 84
7.3 The *Helix* framework 85
7.3.1 Load balancing 86
7.3.2 Load balancing support techniques 88
7.3.3 Frontier queue types 90
7.3.4 Synchronization between BFS iterations 90
7.3.5 Frontier updating 91
7.3.6 Duplicate removing 91
7.3.7 Bitmask status lookup 92
7.3.8 Programming model 92
7.3.9 Modified Graph Representation 93
7.3.10 Optimized Prefix-sum 94
7.4 Experimental Results 95
7.4.1 Graph dataset and system setup 95
7.4.2 The configurability analysis 96
7.4.3 Performance evaluation 100
7.5 Conclusions 101
7.6 Appendix 101

8 Single-Source Shortest Path - H-BF 105
8.1 Introduction 105
8.2 The Bellman-Ford's algorithm 107
8.3 The frontier-based algorithm and its optimizations 107
8.3.1 The edge classification optimization 109
8.3.2 Duplicate removal with Kepler 64-bit atomic instructions . . . 111
8.4 Architecture-oriented Optimizations 113
8.4.1 Memory coalescing, cache modifiers, and texture memory . . . 113
8.4.2 Dynamic virtual warps 114
8.4.3 Dynamic parallelism 116
8.5 Experimental Results 116
8.5.1 Experimental setup 116
8.5.2 Execution time analysis and comparison 117
8.6 Concluding Remarks 120

9 Strongly Connected Components - Multi-Step Algorithm 123
9.1 Introduction 123
9.2 Multi-step Parametric Scheme for SCC Decomposition 124
9.2.1 Parallelization strategy for graph traversal 126
9.3 Experimental results 127
9.4 Conclusions 131

10 Approximate Subgraph Isomorphism - APPAGATO 133
10.1 Materials and methods 133
10.1.1 Definitions and notations 133
10.1.2 The *APPAGATO* algorithm 134
10.1.3 The *APPAGATO* parallel implementation for GPUs 138
10.1.4 Datasets 141
10.2 Results and discussion 142
10.2.1 Performance 142
10.2.2 Quality measurements of matches 143
10.2.3 Querying protein complexes among different species. 144
10.2.4 Datasets and query details 145
10.3 *APPAGATO* Implementation 146
10.3.1 *APPAGATO* performance 152
10.4 Functional coherence measurement in querying protein complexes among different species. 153
10.4.1 Querying disease modules 156
10.5 Conclusions 159

Part III Profiling and Analysis Framework

11 Related Work 165
11.1 Performance models 165
11.2 GPU Microbenchmarking 166

12 Parallel Primitives Profiling 169
12.1 Introduction 169
12.2 Profiler Metrics 172
12.3 Optimization Criteria 173
12.3.1 Occupancy (OCC) 173
12.3.2 Host Synchronization (HSync) 174
12.3.3 Device Synchronization (DSync) 175
12.3.4 Thread Divergence (TDiv) 175
12.3.5 Warp Load Balancing ($LB_{\mathbf{W}}$) 175
12.3.6 Streaming Multiprocessor Load Balancing ($LB_{\mathbf{SM}}$) 176
12.3.7 L1/L2 Granularity ($Gran_{\mathbf{L1}}$/$Gran_{\mathbf{L2}}$) 176
12.3.8 Shared Memory Efficiency ($SMem_{\mathbf{eff}}$) 177
12.3.9 Computation Intensity (CI) 177
12.3.10Data Transfer (DT) 177
12.3.11Overall Quality Metrics (QM) 178
12.4 Weighing of Optimization Criteria on the Overall Quality Metrics . 178
12.4.1 Parallel Primitives 178
12.4.2 Evaluation 180
12.5 Case Studies 183
12.5.1 The *Load Balancing Search* Primitive 183
12.5.2 The *Matrix Transpose* 184
12.6 Conclusion 185

13 A Fine-grained Performance Model 187
13.1 Introduction 187
13.2 Microbenchmarks 189
13.2.1 Microbenchmark Development 190
13.2.2 GPU Device Characterization through Microbenchmarks . . . 190
13.3 Optimization Criteria 191
13.3.1 Host Synchronization 192
13.3.2 Device Synchronization 192
13.3.3 Thread Divergence 192
13.3.4 Warp Load Balancing 193
13.3.5 Streaming Multiprocessor (SM) Load Balancing 193
13.3.6 L1/L2 Granularity 193
13.3.7 Shared Memory Efficiency 194
13.3.8 Throughput/Occupancy 194
13.4 Performance Prediction 195
13.5 Experimental Results 196
13.5.1 Case study 1: Parallel Reduction 196
13.5.2 Case study 2: BFS 197
13.5.3 Case study 3: Matrix Transpose 198
13.6 Remarks 199

14 Power/Performance/Energy Microbenchmarking ... 201
14.1 Introduction ... 201
14.2 The Microbenchmark Suite ... 203
14.2.1 Arithmetic processing benchmarks ... 204
14.2.2 Memory benchmarks ... 205
14.3 Experimental Results ... 207
14.4 Conclusions ... 211

15 Power-aware Performance Tuning of GPU Applications Through Microbenchmarking ... 213
15.1 Introduction ... 213
15.2 The Microbenchmark Suite ... 215
15.2.1 GPU static characteristics ... 216
15.2.2 GPU dynamic characteristics ... 217
15.3 Experimental Results ... 219
15.3.1 GPU Device Characterization ... 219
15.3.2 Vector Reduction ... 223
15.3.3 Matrix Transpose ... 224
15.4 Conclusions ... 225

16 A Performance, Power, and Energy Efficiency Analysis of Load Balancing Techniques for GPUs ... 227
16.1 Introduction ... 227
16.1.1 The *Multi-phase* technique ... 228
16.2 Load balancing analysis ... 231
16.2.1 Characteristics of datasets, GPU devices, and equipment for performance, power, energy efficiency measurement ... 231
16.2.2 Performance, power, energy efficiency analysis ... 233
16.3 Conclusion ... 237

Part IV Dynamic Graph Processing

17 Related Work ... 243
17.0.1 Dynamic Sparse Formats ... 243
17.0.2 K-Truss ... 246
17.0.3 Triangle Counting ... 247
17.0.4 GPU Triangle Counting ... 247
17.0.5 Streaming and Dynamic Triangle Counting ... 247

18 Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices ... 249
18.1 Introduction ... 249
18.2 Hornet Overview ... 250
18.2.1 Block arrays ... 253
18.2.2 Vectorized Bit Tree ... 253
18.2.3 B^+Trees of block arrays ... 255
18.2.4 Data structure initialization ... 255

18.2.5 Dynamic Updates 256
18.2.6 Handling graphs with extra properties 258
18.3 Experimental Results 259
18.4 The *Hornet* data structure 259
18.4.1 Memory utilization efficiency 259
18.4.2 Initialization Time 261
18.4.3 Update rates 262
18.4.4 Dynamic Triangle Counting 263
18.4.5 Breadth-first search 263
18.4.6 SpMV 263
18.5 Conclusions 265

19 Quickly Finding a Truss in a Haystack 267
19.1 Introduction 267
19.2 KTruss Algorithm Using Dynamic Graphs 268
19.2.1 Problem Definition 269
19.3 Proposed Algorithm 269
19.3.1 Triangle Subtraction 269
19.3.2 Triangle Detection For Single Edge-Pair Deletions 270
19.3.3 Triangle Detection For Dual Edge-Pair Deletions 271
19.4 Experimental Results 271
19.4.1 Dynamic Graphs 272
19.4.2 Benchmarks 272
19.4.3 Dataset 273
19.5 Conclusions 276

Part V Applications

20 Invariant Mining 281
20.1 Introduction 281
20.2 Preliminary definitions 282
20.3 Invariant mining 282
20.3.1 The parallel implementation for GPUs 283
20.3.2 Generation of the variable dictionary 285
20.3.3 Data transfer and overlapping of the mining phase 285
20.4 Experimental results 286
20.5 Concluding remarks 287

21 cuRnet: an R package for graph traversing on GPU 289
21.1 Background 289
21.2 Methods 290
21.2.1 Parallel implementation of Breadth-First Search for GPUs . . 293
21.3 Results and discussion 294
21.3.1 Data 295
21.3.2 *cuRnet* performance 295
21.4 Conclusion 299

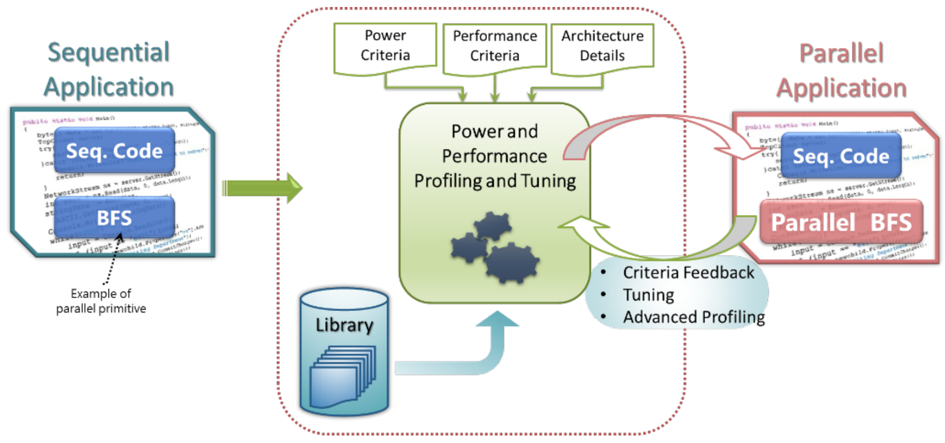
22 Conclusions 301

Part VI Ph.D. Candidate's Bibliografy

References 309

1 Thesis Abstract

Graphs are a common representation in many problem domains, including engineering, finance, medicine, and scientific applications. Different problems map to very large graphs, often involving millions of vertices. Even though very efficient sequential implementations of graph algorithms exist, they become impractical when applied on such actual very large graphs. On the other hand, graphics processing units (GPUs) have become widespread architectures as they provide massive parallelism at low cost. Parallel execution on GPUs may achieve speedup up to three orders of magnitude with respect to the sequential counterparts. Nevertheless, accelerating efficient and optimized sequential algorithms and porting (i.e., parallelizing) their implementation to such many-core architectures is a very challenging task. The task is made even harder since energy and power consumption are becoming constraints in addition, or in same case as an alternative, to performance. This work aims at developing a platform that provides (I) a library of parallel, efficient, and tunable implementations of the most important graph algorithms for GPUs, and (II) an advanced profiling model to analyze both performance and power consumption of the algorithm implementations.



Overview of the proposed platform.

The platform goal is twofold. Through the library, it aims at saving developing effort in the parallelization task through a primitive-based approach. Through the profiling framework, it aims at customizing such primitives by considering both the architectural details and the target efficiency metrics (i.e., performance or power).

Performance-Oriented Implementations of Graph Algorithms for GPU

A library of the most important graph algorithms has been implemented for GPU architectures. It provides high efficient, maintainable, flexible, extendable and power-aware parallel implementations of graph algorithms. The library is focused on main graph algorithms in the literature that have huge applications in real world problems. It includes:

Breadth-first search (BFS-4K). BFS is one of the most common graph traversal algorithms and the building block for a wide range of graph applications. This work presents BFS-4K, a parallel implementation of BFS for GPUs that exploits the more advanced features of GPU-based platforms (i.e., NVIDIA Kepler) and that achieves an asymptotically optimal work complexity. An analysis of the most representative BFS implementations for GPUs at the state of the art and their comparison with BFS-4K are reported to underline the efficiency of the proposed solution.

Fully Configurable BFS (Helix). This work presents *Helix*, a fully configurable BFS for GPUs. It relies on a flexible and expressive programming model that allows selecting, for each BFS feature (e.g., frontier handling, load balancing, duplicate removing, etc.) and among different implementation strategies of them, the best combination to address the graph characteristics. Thanks to the high reconfigurability, Helix provides high-performance and customized BFSs with speedups ranging from 1.2x to 18.5x with regard to the best parallel BFS solutions for GPUs at the state of the art.

Single source shortest path (H-BF). Finding the shortest paths from a single source to all other vertices is a common problem in graph analysis. This work presents a parallel implementation of the Bellman-Ford algorithm that exploits the architectural characteristics of recent GPU architectures (i.e., NVIDIA Kepler, Maxwell, Pascal) to improve both performance and work efficiency. The work presents different optimizations to the implementation, which are oriented both to the algorithm and to the architecture. The experimental results show that the proposed implementation provides an average speedup of 5x higher than the existing most efficient parallel implementations for SSSP, that it works on graphs where those implementations cannot work or are inefficient.

Approximate Sub-graph Isomorphism in Biological Network (APPAGATO). Biological network querying is a problem in bioinformatics which solution requires a considerable computational effort. Given a target and a query network, the problem is to find occurrences of the query in the network allowing nodes and edges mismatches, i.e. similarities on node labels, nodes or edges deletions. This work proposes APPAGATO, a stochastic and parallel algorithm to find approximate occurrences of a query network in biological networks. APPAGATO allows nodes and edges mismatches. Thanks to its randomic and parallel nature, it results feasable on large networks compared to existing tools and also statically more accurate.

Strongly Connected Component (SCC). The problem of decomposing a directed graph into strongly connected components (SCCs) is a fundamental graph problem that is inherently present in many scientific and commercial applications. This work introduces a novel parametric multi-step scheme to evaluate existing GPU-accelerated algorithms for SCC decomposition in order to alleviate the burden of the choice and to help the user to identify which combination of existing techniques for SCC decomposition would fit an expected use case the most. The study confirms that there is no algorithm that would beat all other algorithms in the decomposition on all of the classes of graphs. The contribution thus represents an important step towards an ultimate solution of automatically adjusted scheme for the GPU-accelerated SCC decomposition.

Load balancing in graph algorithms. Load balancing is a fundamental performance aspect when facing irregular problems and in particular graph algorithms. The latter strongly rely on prefix-scan values to provide an efficient memory representation. Prefix-scan is one of the most common operation and building block for a wide range of parallel applications for GPUs. It allows the GPU threads to efficiently find and access in parallel to the assigned data. Nevertheless, the workload decomposition and mapping strategies that make use of prefix-scan can have a significant impact on the overall application performance. This work presents presents Multi-Phase Search, an advanced dynamic technique that addresses the workload unbalancing problem by fully exploiting the GPU device characteristics whose complexity is sensibly reduced with respect to the other dynamic approaches in the literature.

Profiling and Analysis Framework

The increasing programmability, performance, and cost/effectiveness of GPUs have led to a widespread use of such many-core architectures to accelerate general-purpose applications. Nevertheless, tuning applications to efficiently exploit the GPU potentiality is a very challenging task, especially for inexperienced programmers. This is due to the difficulty of developing a SW application for the specific GPU architectural configuration, which includes managing the memory hierarchy and optimizing the execution of thousands of concurrent threads while maintaining the semantic correctness of the application. Even though several profiling tools exist, which provide programmers with a large number of metrics and measurements, it is often difficult to interpret such information for effectively tuning the application. This research focuses on the development of a performance model that allows accurately estimating the potential performance of the application under tuning on a given GPU device and, at the same time, it provides programmers with interpretable profiling hints.

Although, in general, the goals of algorithms and applications in computer science are to minimize measures related to time and space, another interesting and tangible cost on a modern computing platform is the power consumption. Many analytical and quantitative models have been proposed in the literature to estimate or predict the power consumption of parallel applications but none of these approaches takes into account how algorithms and parallel programming techniques are related to power and energy consumption of the device. In particular, existing solutions provide information that is rarely useful to effectively improve the code. This work aims at developing a power consumption model and profiling framework that provides useful feedback to identify the causes of power and energy consumption of an application, to understand where and suggest how to modify the application to improve the power behavior. It includes:

Parallel Primitives Profiling Framework (Pro++). Parallelizing software applications through the use of existing optimized primitives is a common trend that mediates the complexity of manual parallelization and the use of less efficient directive-based programming models. On the other hand, the spreading of such a primitive-based programming model and the different GPU architectures have led to a large and increasing number of third-party libraries, which often provide different implementations of the same primitive, each one optimized for a specific architecture. This work presents *Pro++*, a profiling framework for GPU primitives that allows measuring the implementation quality of a given primitive by considering the target architecture characteristics.

GPU Performance Model. Tuning applications to efficiently exploit the GPU potentiality is a very challenging task, especially for inexperienced programmers. This is due to the difficulty of developing a SW application for the specific GPU architectural configuration, which includes managing the memory hierarchy and optimizing the execution of thousands of concurrent threads while maintaining the semantic correctness of the application. This work presents a performance model that allows accurately estimating the potential performance of the application under tuning on a given GPU device and, at the same time, it provides programmers with interpretable profiling hints.

Microbenchmark Suite (MIPP). This work presents a suite of microbenchmarks, which are specialized chunks of GPU code that exercise specific device components (e.g., arithmetic instruction units, shared memory, cache, DRAM, etc.) and that provide the actual characteristics of such components in terms of throughput, power, and energy consumption. The suite aims at enriching standard profiler information and guiding the GPU application tuning on a specific GPU architecture by considering all three design constraints (i.e., power, performance, energy consumption).

Power-aware Performance Tuning Through Microbenchmarking. Tuning GPU applications is a very challenging task as any source-code optimization can sensibly impact performance, power, and energy consumption of the GPU device. Such an impact also depends on the GPU on which the application is run. This work presents a suite of microbenchmarks that provides the actual characteristics of specific GPU device components (e.g., arithmetic instruction units, memories, etc.) in terms of throughput, power, and energy consumption. It shows how the suite can be combined to standard profiler information to efficiently drive the application tuning by considering the three design constraints (power, performance, energy consumption) and the characteristics of the target GPU device.

Case Study: Performance, Power, and Energy efficiency of Load Balancing Algorithms. Load balancing is a key aspect to face when implementing any parallel application for GPUs. It is particularly crucial if one considers that it strongly impacts on performance, power and energy efficiency of the whole application. This work shows and compares, in terms of performance, power, and energy efficiency, the experimental results obtained by applying all the different static, dynamic, and semi-dynamic techniques at the state of the art to different datasets and over different GPU technologies.

Dynamic graph representation and algorithms

Sparse data computations are ubiquitous in science and engineering. Unlike their dense data counterparts, sparse data computations have less locality and more irregularity in their execution, making them significantly more challenging to optimize. Even more challenging is their optimization for parallel applications and algorithms. Dynamic sparse data applications are now pervasive and can be found in many domains. *Dynamic* refers to the fact that the data is changing at very high rates. For example, updates might represent the change in the current-flow of a power network or the road-congestion for a transportation network. The number of applications is considerably high and many formulations of these problems end up being either graph-based or linear-algebra based. This thesis section includes two main contributions:

Efficient Data Structure for Dynamic Sparse Graphs and Matrices (Hornet). Most of the existing formats for sparse data representations on parallel architectures are restricted to static data sets, while those for dynamic data suffer from inefficiency both in terms of performance and memory footprint. This work presents Hornet, a novel data representation that targets dynamic graph analytics and linear algebra based problems. The data structure includes an optimized memory manager that is responsible for memory allocation, reclamation, and for ensuring low overhead to represent the data during its evolution. Hornet is scalable with the input data, flexible in representing data set properties, and does not require any data re-allocation or re-initialization during the data evolution.

K-Truss. The KTruss of a graph is a subgraph such that each edge is tightly connected to the remaining elements in the k-truss. This work presents a novel algorithm for finding both the KTruss of the graph (for a given k), as well as the maximal KTruss using a dynamic graph formulation. The algorithm has two main benefits. The algorithm shows high work-efficiency and it is extremely scalable in contrast to past approaches.

Applications

Invariant mining (Mangrove). Invariant mining is a promising strategy that extracts logic formulas holding between a couple (or several couples) of points in an implementation. Such formulas express stable conditions in the behavior of the system under verification (SUV) for all its executions, which can be used to analyze several aspects in verification of SW programs and HW designs, at different abstraction levels. For complex SUV, this could require to elaborate thousands of execution traces, including millions of clock cycles, and predicating over hundreds of variables, which becomes an unmanageable time-consuming activity for existing approaches. This research activity aims at providing very efficient and flexible parallel algorithm based on advanced GPU optimization techniques and on run-time inference to avoid redundant elaboration to allow the analysis of huge SUV traces in short time.

cuRnet. R has become the de-facto reference analysis environment in Bioinformatics. Plenty of tools are available as packages that extend the R functionality, and many of them target the analysis of biological networks. Several algorithms for graphs, which are the most adopted mathematical representation of networks, are well-known examples of applications that require high-performance computing, and for which classical sequential implementations are becoming inappropriate. This work presents *cuRnet*, a R package that provides a parallel implementation for GPUs of the breath-first search (BFS), the single-source shortest paths (SSSP), and the strongly connected components (SCC) algorithms. The package allows offloading computing intensive applications to GPU devices for massively parallel computation and to speed up the runtime up to one order of magnitude with respect to the standard sequential computations on CPU.

Part I

Background

Introduction

This part of the thesis presents some preliminary concepts concerning GPU (Graphic Processing Unit), CUDA programming model, and static graph representation, which facilitate the reader to better understand the context of the work.

The part is organized as follows. Section 2 summarizes the programming and architectural characteristics of the most recent GPUs that have been exploited in the following works. Besides GPU characteristics, the section focuses on the programming model of such devices (CUDA) and related advanced primitives which allows to improve the common parallel operations. Section 3 introduces the topic of static graph data structures on GPUs. It presents and compares the main important data structures and techniques applied for representing graphs on GPUs at the state of the art.

2

Graphic Processing Unit (GPU)

GPU devices are massive multithreaded architectures composed by scalable arrays of parallel processors called Streaming Multiprocessors (*SMs*). Each SM contains a set of cores, called Stream Processors (*SPs*) that executes warp instructions. Each SP executes fixed-point and floating-point single-precision operations through dedicated ALU and FPU units. SPs are supported by special purpose units that execute double-precision instructions (*DFU*), transcendental operations (*SFU*), such as trigonometric functions, and load/store units to issue memory instructions and to calculate memory addresses. The number of SPs per streaming multiprocessor is fixed by the *compute capability* of the device, while the number of DFU, SFU, load/store units depends on the particular chip. On the other hand, the SM has limited instruction throughput per clock cycle. GPUs also feature a sophisticated memory hierarchy, which involves thread registers, shared memory, DRAM memory and two-level cache (L1 within a SP, while L2 accessible to all threads).

In the last NVIDIA GPU architectures, Kepler, Maxwell, Pascal and Volta, a small read-only cache per-SM (called Texture cache) is also available to reduce global memory data access.

2.1 Computed Unified Device Architecture (CUDA)

The threads run the same kernel concurrently, and each one is associated with a unique thread ID. A kernel is executed by a 1-, 2-, or 3-dimensional *grid* of thread *blocks*. Threads are arranged into three-dimensional thread blocks.

CUDA is a parallel computing platform and programming model proposed by NVIDIA. CUDA comes with a software environment that allows developers to use C/C++ as a high-level programming language targeting heterogeneous computing on CPUs and GPUs. Through API function calls, called *kernels*, and language extensions, CUDA allows enabling and controlling the offload of compute-intensive routines. A CUDA kernel is executed by a *grid* of *thread blocks*. A thread block is a batch of *threads* that can cooperate and synchronize each other via shared memory, atomic operations and barriers. Blocks can execute in any order while threads in different blocks cannot directly cooperate.

Groups of 32 threads with consecutive indexes within a block are called *warps*. A thread warp executes in SIMD-like way the same instruction on different data concurrently. In a warp, the synchronization is implicit since the threads execute in lockstep. Different warps within a block can synchronize through fast barrier primitives. In contrast, there is no native thread synchronization among different blocks as the CUDA execution model requires independent block computation for scalability reasons. The lack of support for inter-block synchronization requires explicit synchronization with the host, which involves significant overhead.

A warp thread is called *active* if it successfully executes a particular instruction issued by the warp scheduling. A CUDA core achieves the full efficiency if all threads in a warp are active. Threads in the same warp stay idle (not active) if they follow different execution paths. In case of *branch divergence*, the core serializes the execution of the warp threads.

In modern GPU architectures (e.g., NVIDIA Kepler, Maxwell, Pascal, and Volta) each SM can handle up to 2048 threads and 64 warps concurrently. The number of warps per SM is called *theoretical occupancy* of the device. Each SM has four warp schedulers, that issue the instructions from a given warp to the corresponding SIMD core, allowing 8 instructions to be execute per clock cycle. Thread blocks are dynamically dispatched to the SMs through a hardware scheduler that works at device-level. The grid configuration and the thread block/warp scheduling strongly affects performance.

Threads of the same block cooperate by sharing data through fast on-chip shared memory and by synchronizing their execution through extremely fast (i.e., HW implemented) barriers. Shared memory is organized in a 32-column matrix (i.e., memory banks). When multiple threads of the same block access different 32-bit words of the same bank, a conflict occurs. Such a *bank conflict* involves re-execution of the memory instructions, with a consequent lost of performance. The GPU memory hierarchy includes also the DRAM, constant, and L2 cache memories, which are visible to all the threads of a grid. The constant cache is a fast and small read-only memory space commonly used to kernel parameter passing and for storing data that will not change during the kernel execution. In contrast, DRAM and L2 cache provide high latency read/write spaces to all threads.

Private variables of threads and local arrays with static indexing are placed into registers. Large local arrays and dynamic indexing arrays are stored in L1 and L2 cache. Thread variables that are not stored in registers are also called *local memory*. The access pattern of global memory accesses is critical for the performance. In order to maximize the global memory bandwidth and to reduce the number of bus transactions, multiple memory accesses can be combined into a single transaction. *Memory coalescing* consists of executing memory accesses by different warp threads to an aligned and continuous segment of memory.

Finally, the host-GPU device communication bus allows overlapping CPU-GPU data transfers with the kernel computations to minimize the host-device data transfers.

2.2 Modern GPU architectures

With the most recent GPU architectures (e.g., NVIDIA Kepler [210], Maxwell [211], Pascal [212]), and Volta [9], several advanced programming techniques and primitives have been introduced to improve the efficiency of the most common parallel operations and to simplify the programming model. Some examples are *warp shuffle*, *warp voting*, and *dynamic parallelism*. They allow a more straightforward porting of sequential applications to GPUs through transparent thread coordination and communication mechanisms, while guaranteeing high-performance at the same time.

Warp shuffle instructions provide a fast mechanism to move data among threads of the same warp by exchanging thread register values in a single operation. Warp shuffle operations avoid shared memory accesses, thus increasing the available space of such memory at run-time for other uses.

Warp voting implements efficient procedures for intra-warp coordination (e.g., `any`, `all`, and `ballot` instructions), which allow simplifying the application control flow.

Recent GPUs also extend the execution model with the *dynamic parallelism*, which allows the GPU kernel to create and synchronize new nested work directly without returning the control to the host. Dynamic parallelism helps developers to balance irregular workload in applications that show complex patterns.

The recent architectures also introduced the *read-only data cache*. Such a separate cache (i.e., with separate memory pipe), which is available to each symmetric multiprocessor, SM, (generally 48KB per SM) allows improving performance through bandwidth-limited kernels. The implementation proposed in this work takes advantage of this feature to alleviate the L1 cache pressure during data loads from global memory .

Such architectures also expand the native support for *64-bit atomic operations* in global memory. This feature is used in this work to improve the work efficiency. The Kepler architecture also introduces the *8-byte access mode* to the shared memory. The shared memory throughput is doubled by increasing the bank width to 8 bytes.

From the perspective of architectural characteristics, GPU application developers can more and more afford on atomic operations for fast *hardware-implemented* shared memory and on an increasing amount of such a shared memory. From the NVIDIA Maxwell architecture on, for example, the amount of shared memory has been doubled compared to the Kepler architecture, and developers can exploit the unified L1/Texture cache for global memory loads. This allows reaching higher hit rates compared to the separated L1 and Texture caches of older architectures.

3

Static Graph Representation

The graph representation adopted when implementing a graph algorithm for GPUs strongly affects the implementation efficiency and performance. The three most common representations are *adjacency matrices*, *adjacency lists*, and *egdes lists* [74,237]. They have different characteristics and each one finds the best application in different contexts (i.e., graph and algorithm characteristics).

As for the sequential implementations, the quality and efficiency of the graph representation can be measured over three properties: the involved memory footprint, the time required to determine whether a given edge is in the graph, and the time it takes to find the neighbours of a given vertex. For GPU implementations, such a measure also involves the load balancing and the memory coalescing.

Given a graph $G = (V, E)$, where V is the set of vertices, E is the set of edges, and d_{max} is the largest diameter of the graph, Table 3.1 summarizes the main features of the data representations, which will be discussed in detail in the next paragraphs.

	Space	$(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{E}$	$(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{adj}(\boldsymbol{v})$	Load Balancing	Mem. Coalescing
Adj Matrices	$O(\|V\|^2)$	$O(1)$	$O(\|V\|)$	Yes	Yes
Adj Lists	$O(\|V\|+\|E\|)$	$O(d_{max})$	$O(d_{max})$	Difficult	Difficult
Edges Lists	$O(2\|E\|)$	$O(\|E\|)$	$O(\|E\|)$	Yes	Yes

TABLE 3.1: Main feature of data representations.

3.0.1 Adjacency Matrices

An adjacency matrix allows representing a graph with a $V \times V$ matrix $M = [f(i, j)]$ where each element $f(i, j)$ contains the attributes of the edge (i, j). If the edges do not have any attribute, the graph can be represented with a boolean matrix to save memory space (see Figure 3.1).

Common algorithms that use this representation are *all-pair shortest path* and *transitive closure* [54,92,139,171,181,266,269]. If the graph is weighted, each value of $f(i, j)$ is defined as follows:

$$M\left[i,j\right]\begin{cases}0 & \text{if } i=j\\ w\left(i,j\right) & \text{if } i\neq j \text{ and } \left(i,j\right)\in E\\ \infty & \text{if } i\neq j \text{ and } \left(i,j\right)\notin E\end{cases}$$

On GPUs, both directed and undirected graphs represented by an adjacency matrix take $O(|V|^2)$ memory space, since the whole matrix is stored in memory with a large continuous array. In GPU architectures, it is also important, for performance reasons, to align the matrix with memory to improve coalescence of memory accesses. In this context, the CUDA language provides the function *cudaMallocPitch* [207] to pad the data allocation, with the aim of meeting the alignment requirements for memory coalescing. In this case the indexing changes as follow:

$$M[i \cdot V + j] \to M[i \cdot pitch + j]$$

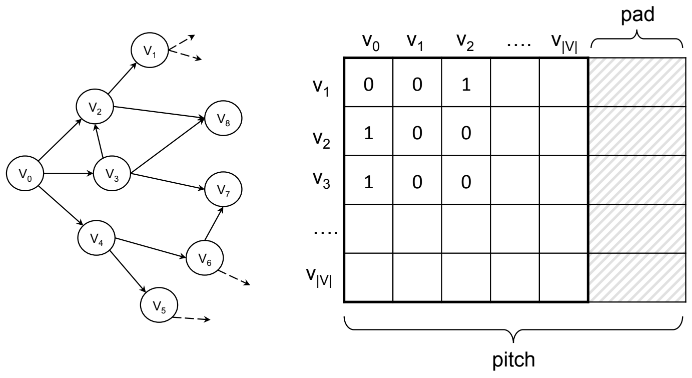


FIG. 3.1: *Matrix representation of a graph in memory*

The $O(|V|^2)$ memory space required is the main limitation of the adjacency matrices . Even on recent GPUs, they allow handling fairly small graphs. As an example, considering a GPU device with 4GB of DRAM, the biggest graph that can be represented through an adjacency matrix can have a maximum number of vertices equals to 32,768 (which, for actual graph datasets, is considered restrictive). In general, adjacency matrices best apply to represent small and dense graphs (i.e. $|E| \approx |V|^2$). In some cases, such as for the all-pairs shortest path problem, graphs larger than the GPU memory are partitioned and each part processed independently [92, 181, 266].

3.0.2 Adjacency Lists

Many linear systems and graph problems arising from the discretization of real-world problems show high sparsity in the sense that the elements of these structures are loosely coupled or connected. The adjacency lists are the most common representation for sparse graphs, where the number of edges is typically a constant factor larger than $|V|$. Since the sequential implementation of the adjacency lists relies on pointers, it is not suitable for GPUs. they are replaced, in GPU implementations, by the *Compressed Sparse Row* (CSR) or *Compressed Row Storage* (CRS) sparse matrix format [36, 37].

In general, an adjacency list consists of an array of vertices (ArrayV) and an array of edges (ArrayE), where each element in the vertex array stores the starting index (in the edge array) of the edges outgoing from each node. The edge array stores the destination vertices of each edge (see Figure 5.5). This allows visiting the neighbors of a vertex v by reading the edge array from ArrayV[v] to ArrayV[v + 1].

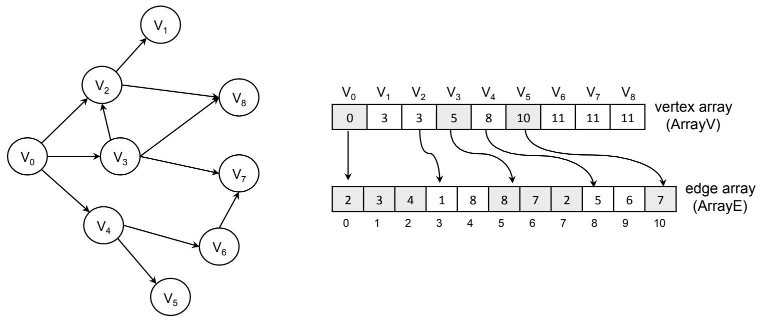


FIG. 3.2: *Adjacency list representation of a weighted graph*

The attributes of the edges are in general stored in the edge array through an *array of structures* (AoS). For example, in a weighted graph, the destination and the weight of an edge can be stored in a structure with two integer values (*int2* in CUDA [208]). Such a data organization allows many scattered memory accesses to be avoided and, as a consequence, the algorithm performance to be improved.

Undirected graphs represented with the CSR format take $O(|V| + 2|E|)$ space since each edge is stored twice. If the problem requires also the incoming edges, the same format is used to store the reverse graph where the vertex array stores the offsets of the incoming edges. The space required with the reverse graph is $O(2|V| + 2|E|)$.

The main issues of the CSR format are the load balancing and the memory coalescing, due to the irregular structure of such a format. If the algorithm involves visiting each vertex at each iteration, the memory coalescing for the vertex array is simple to achieve but, on the other hand, it is difficult to achieve for the edge array. Achieving both load balancing and memory coalescing requires advanced and sophisticated implementation techniques.

For many graph algorithms, the adjacency list representation guarantees better performance than adjacency matrix and edge lists [59, 118, 125, 172, 190].

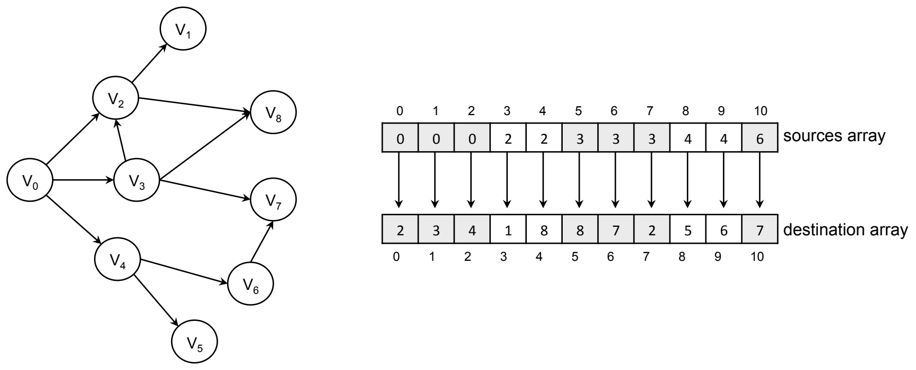


FIG. 3.3: *Edges list representation of a weighted graph*

3.0.3 Edges List

The edge list representation of a graph, also called *coordinate list* (COO) sparse matrix [81], consists of two arrays of size $|E|$ that store the source and the destination of each edge (see Figure 3.3). To improve the memory coalescing, similarly to CSR, the source, the destination and other edge attributes (such as the edge weight) can be stored in a single structure (AoS) [245].

Storing some vertex attributes in external arrays is also necessary in many graph algorithms. For this reason, the edge list is sorted by the first vertex in each ordered pair, such that adjacent threads are assigned to edges with the same source vertex. This allows improving coalescence of memory accesses for retrieving the vertex attributes. In some cases, sorting the edge list in the lexicographic order may also improve coalescence of memory accesses for retrieving the attributes of the destination vertices [135]. The edge organization in a sorted list allows reducing the complexity (from $O(|E|)$ to $O(\log |E|)$) of verifying whether an edge is in the graph, by means of a simple binary search [214].

For undirected graphs, the edge list should not be replicated for the reverse graph. Processing the incoming edges can be done by simply reading the source-destination pairs in the inverse order, thus halving the number of memory accesses. With this strategy, the space required for the edge list representation is $O(2|E|)$.

The edge list representation is suitable in those algorithms that iterate over all edges. For example, it is used in the GPU implementation of algorithms like *betweenness centrality* [135,183]. In general, this format does not guarantee performance comparable to the adjacency list but it allows achieving both perfect load balancing and memory coalescing with a simple thread mapping. In graphs with a non-uniform distribution of vertex degrees, the COO format is generally more efficient than CSR [135, 249].

Final Notes: These different formats vary in their representation characteristics: memory footprint, memory access patterns involved for applications, suitability to exploit the memory hierarchy, design complexity, implementation complexity, conversion time, and tuning capabilities. While many of them can be considered ideal data layout for specific applications, CSR still remains the most popular format, in part due to its simplicity and its reduced memory requirements.

Part II

Performance-Oriented Implementations of Graph Algorithms for GPU

Introduction

This part of the thesis presents the topic of efficient implementations of graph algorithms for GPU architectures. It starts by presenting and comparing the main important techniques applied for analyzing graphs on GPUs at the state of the art with particular emphasis on load balancing strategies and their main issues. It then presents the theory and an updated review of the state of the art implementations of graph algorithms for GPUs. Later, the part focuses on graph traversal algorithms (BFS), single-source shortest path (SSSP), strongly connected components decomposition (SCC), and approximate subgraph isomorphism.

This part of the thesis is organized as follows. Section 4 presents the related work concerning the workload partitioning problem, the most important graph algorithms and the related implementations targeting performance. Section 5 introduces an efficient dynamic partitioning and mapping technique, called *Multi-phase Mapping*, to address the workload unbalancing problem in both regular and irregular problems and how it has been implemented by fully exploiting the GPU device characteristics. Section 6 presents *BFS-4K*, a parallel implementation of BFS for GPUs, which exploits the more advanced features of NVIDIA Kepler GPU and achieve an asymptotically optimal work complexity. Section 7 describes *Helix*, a fully configurable BFS for GPUs which further improves the previous implementation by adapting and tuning its features depending on the graph characteristics. Section 8 introduces *H-BF*, a high-performance implementation of the Bellman-Ford algorithm for GPUs, which exploits the more advanced features of GPU architectures to improve the execution speedup and the work efficiency with respect to any implementation at the state of the art for solving the SSSP problem. Section 9 presents a new parametric *multi-step* scheme that allows us to compactly define a set of algorithms for SCC graph decomposition as well as a type of the parallelization for individual graph operations. The scheme covers the existing algorithms and techniques mentioned above, but also introduces several new variants of the multi-step algorithm. Finally, Section 10 describes *APPAGATO*, a stochastic and parallel algorithm to find approximate occurrences of a query network in biological networks. *APPAGATO*, handles node, edge, and node label mismatches. Thanks to its randomic and parallel nature, it applies to large networks and, compared to existing tools, it provides higher performance as well as statistically significant more accurate results.

4

Related Work

This section introduces the related work concerning efficient implementations of graph algorithms for GPU devices. It first describes the workload partition problem on fine-grained parallel architectures and the main techniques in the literature to deal with irregular workload applications such graph algorithms. Later, it describes state-of-the-art approaches for graph traversal and breadth-first search which represent a common aspect of most graph procedures. Then, it presents the related work for two important graph algorithms, single-source shortest path, and strongly connected components.

4.1 The workload partitioning problem in GPUs

Consider a workload to be partitioned and mapped to GPU threads (see Fig. 4.1). The workload consists of work-units, which are grouped into work-items. As a simple and general example, in the parallel breadth-first search (BFS) implementation for graphs, the workload is the whole graph, the work-units are the graph nodes,

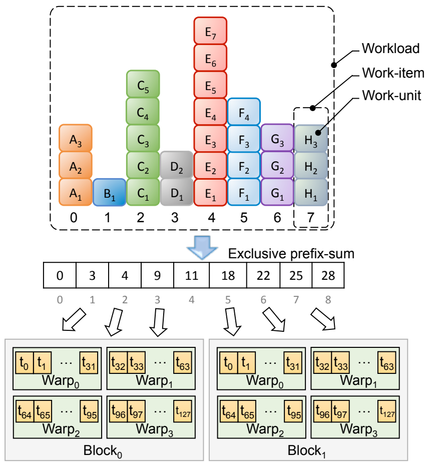


FIG. 4.1: *Overview of the load balancing problem in the workload partitioning and mapping to threads of scan-based applications*

and the work-items are the node neighbors of each node. The native mapping is implemented over work-items through the prefix-sum procedure. A prefix-sum array, which stores the offset of each work-item, allows the GPU threads to easily and efficiently access the corresponding work-units. Considering the example of Fig. 4.1 associated to the BFS, the neighbor analysis of eight nodes is partitioned and mapped to eight threads. t_0 elaborates the neighbors of node *0* (work-units A), t_1 elaborates the neighbors of node *1* (work-unit B), and so on. Even though such a native mapping is very easy to implement and does not introduce considerable overhead in the parallel application, it leads to load imbalance across work-items since, as shown in the example, each work-item may have a variable number of work-units.

In the literature, the techniques for partitioning and mapping (for the sake of brevity, *mapping* in the following) a workload to threads based on prefix-sum for GPU applications can be organized in three classes: *Static mapping*, *semi-dynamic mapping*, and *dynamic mapping*. They are all based on the prefix-sum array that, in the following, is assumed to be already generated (the prefix-sum array is generated, depending on the mapping technique, in a preprocessing phase [284], at run-time if the workload changes at every iteration [59, 190], or it could be already part of the problem [287]).

4.1.1 Static mapping techniques

This class includes all the techniques that statically assign each work-item (or block of work-units) to a corresponding GPU thread. In general, this strategy allows the overhead for calculating the work-item to thread mapping to be negligible during the application execution but, on the other hand, it suffers from load unbalancing when the work-units are not regularly distributed over the work-items. The main important techniques are summarized in the following.

Work-items to threads

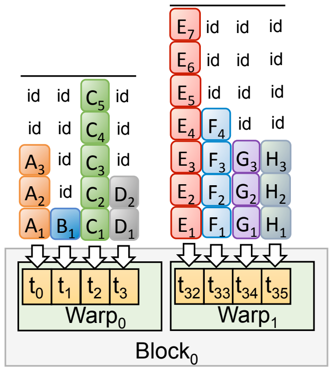


FIG. 4.2: *Example of work-items to threads mapping*

It represents the simplest mapping approach by which each work-item is mapped to a single thread [117]. Fig. 4.2 shows an example, in which the eight items of Fig. 4.1 are assigned to a corresponding number of threads. For the sake of clarity, only four threads per warp have been considered in the example to underline more than one level of possible unbalancing of this technique. First, irregular (i.e., unbalanced) work-items mapped to threads of the same warp lead the warp threads to be in idle state (i.e., branch divergence). t_1, t_3, and t_0 of $warp_0$ in Fig. 4.2 are an example. Irregular work-items lead to whole warps to be in idle state (e.g., $warp_0$ w.r.t. $warp_1$ in 4.2).

In addition, considering that work-units of different items are generally stored in non-adjacent addresses in global memory, this mapping strategy leads to sparse and non-coalesced memory accesses. As an example, threads t_0, t_1, t_2, and t_3 of $Warp_0$ concurrently access to the non adjacent units A_1, B_1, C_1, and D_1, respectively. For all these reasons, this technique is suitable to applications running on very regular data structures, in which any more advanced mapping strategy run at run time (as explained in the following sections) would lead to unjustified overhead.

Virtual Warps

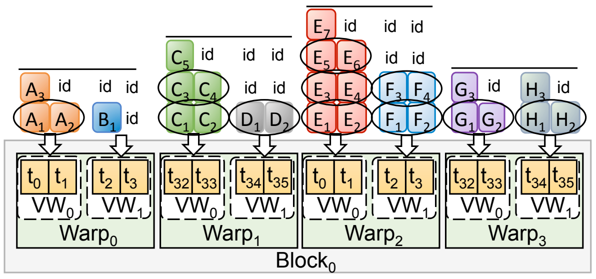


FIG. 4.3: *Example of Virtual warps work-units mapping (black circles represent coalesced memory accesses)*

This technique consists of assigning chunks of work-units to groups of threads called *virtual warps*, where the virtual warps are equally sized and the threads of a virtual warp belong to the same warp [125]. Fig. 4.3 shows an example in which the chunks correspond to the work-items and, for the sake of clarity, the virtual warps have size equal to two threads. Virtual warps allow the workload assigned to threads of the same group to be almost equal and, as a consequence, it allows reducing branch divergence. This technique improves the coalescing of memory accesses since more threads of a virtual warp access to adjacent addresses in global memory (e.g., t_0, t_1 of $Warp_2$ in Fig. 4.3). These improvements are proportional to the virtual warp size. Increasing the warp size leads to reducing branch divergence and better coalescing the work-unit accesses in global memory. Nevertheless, virtual warps have several limitations. Given the number of work-items and a virtual warp size, the required number of threads is:

VirtualWarp

```
1:  VW_INDEX = TH_INDEX / |VirtualWarp|
2:  LANE_OFFSET = TH_INDEX % |VirtualWarp|
3:  INIT = prefixsum[VW_INDEX] + LANE_OFFSET
4:  for i = INIT to prefixsum[VW_INDEX+1] do
5:      Output[i] = VW_INDEX
6:      i = i + |VirtualWarp|
7:  end
```

$$\#RequiredThreads = \#workitems \cdot |\,VirtualWarp|$$

If the number is greater than the available threads, the work-item processing is serialized with a consequent decrease of performance. Indeed, a wrong sizing of the the virtual warps can significantly impact on the application performance. In addition, this technique provides good balancing among threads of the same warp, while it does not guarantee good balancing among different warps nor among different blocks. Another major limitation of such a static mapping approach is that the virtual warp size has to be fixed statically. This represents a major limitation when the number and size of the work-items change at run time.

The algorithm run by each thread to access the corresponding work-units is summarized as follows:
where VW_INDEX and LANE_OFFSET are the virtual warp index and offset for the thread (e.g., VW_0, and 0 for t_0 in the example of Fig. 4.3), INIT represents the starting work-unit id, and the *for* cycle represents the accesses of the thread to the assigned work-units (e.g., A_1, A_3 for t_0 and A_2 for t_1).

4.1.2 Semi-dynamic mapping techniques

This class includes the techniques by which different mapping configurations are calculated statically and, at run time, the application switches among them.

Dynamic Virtual Warps + Dynamic Parallelism

This technique has been introduced in the work [59] and relies on two main strategies. It implements a virtual warp strategy in which the virtual warp size is calculated and set at run time depending on the workload and work-item characteristics (i.e., size and number). At each iteration, the right size is chosen among a set of possible values, which spans from 1 to the maximum warp size (i.e., 32 threads for NVIDIA GPUs, 64 for AMD GPUs). For performance reasons, the range is reduced to power of two values only. Considering that a virtual warp size equal to one has the drawbacks of the *work-item to thread* technique and that memory coalescence increases proportionally with the virtual warp size (see Section 4.1.1), too small sizes are excluded from the range a priori. The dynamic virtual warp strategy provides a fair balancing in irregular workloads. Nevertheless, it is inefficient in case of few and very large work-items (e.g., in benchmarks representing scale free networks or graphs with power-law distribution in general).

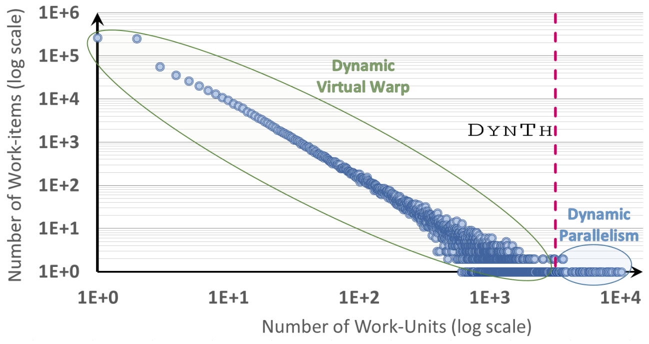


FIG. 4.4: *Example of Dynamic Virtual Warps + Dynamic Parallelism work-units mapping where the dynamic parallelism is applied to a subset of the workload with a power-law distribution*

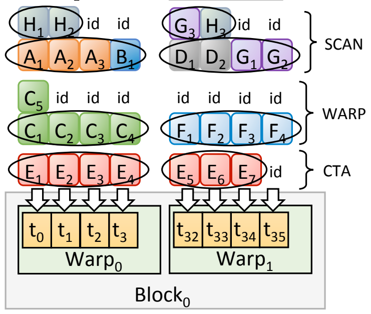


FIG. 4.5: *Example of CTA+Warp+Scan work-units mapping (black circles represent coalesced memory accesses)*

On the other hand, dynamic parallelism, which exploits the most advanced features of the GPU architectures (e.g., from NVIDIA Kepler on) [202] allows recursion to be implemented in the kernels and, thus, threads and thread blocks to be dynamically created and properly configured at run time without requiring kernel returns. This allows fully addressing the work-item irregularity. Nevertheless, the overhead introduced by the dynamic kernel stack may elude this feature advantages if replicated for all the work-items unconditionally [59] [62].

To overcome these limitations, dynamic virtual warps and dynamic parallelism are combined into a single mapping strategy and applied alternatively at run time. The strategy applies dynamic parallelism to the work-items having size greater than a threshold (DynTh), otherwise it applies dynamic virtual warps (Fig. 4.4 shows an example). It best applies to applications with few and strongly unbalanced work-items that may vary at run time (e.g., applications for sparse graph traversal). This technique guarantees balancing among threads of the same warps and among warps. It does not guarantee balancing among blocks.

CTA+Warp+Scan

In the context of graph traversal, Merrill et al. [190] proposed an alternative approach to the load balancing problem. Their algorithm consists of three steps:

Strip-Mined Gathering

```
 1: while any(Workloads[$TH_{ID}$] > $CTA_{TH}$) do
 2:     if Workloads[$TH_{ID}$] > $CTA_{TH}$ then
 3:         SharedWinnerID = $TH_{ID}$
 4:     sync
 5:     if $Th_{ID}$ = SharedWinnerID
 6:         SharedStart = prefixsum[$TH_{ID}$]
 7:         SharedEnd = prefixsum[$TH_{ID}$ + 1]
 8:
 9:     sync
10:     INIT = SharedStart + $TH_{ID}$%|$TH_{SET}$|
11:     for $i$ = INIT to SharedEnd do
12:         Output[$i$] = SharedWinnerID
13:         $i = i + |TH_{SET}|$
14:     end
15: end
```

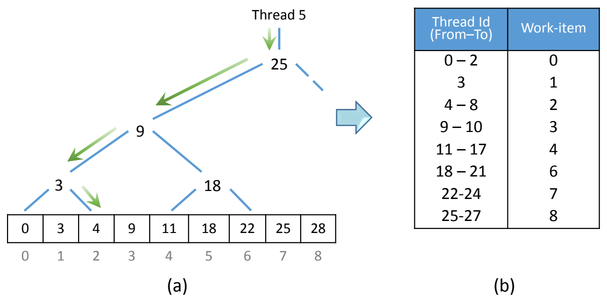
1. All threads of a block access the corresponding work-item (through the work-item to thread strategy) and calculate the item sizes. The work-items with size greater than a threshold (CTA_{TH}) are non-deterministically ordered and, one at a time, they are (i) copied in the shared memory, and (ii) processed by all the threads of the block (called cooperative thread array - CTA). The algorithm of such a first step (which is called *strip-mined gathering*) is run by each thread (TH_{ID}). It can be summarized as follows:
   where row 3 implements the non-deterministic ordering (based on iterative match/winning among threads), rows 5-8 calculate information on the work-item to be copied in shared memory, while rows 10-14 implement the item partitioning for the CTA. This phase introduces significant overhead for the two CTA synchronizations and, rows 5-8 are run by one thread only.
2. In the second step, the strip-mined gathering is run with a lower threshold ($WARP_{TH}$) and at warp level. That is, it targets smaller work-items and a cooperative thread array consists of threads of the same warp. This allows avoiding any synchronization among threads (as they are implicitly synchronized in SIMD-like fashion in the warp) and addressing work-items with sizes proportional to the warp size.
3. In the third step the remaining *work-items* are processed by all block threads. The algorithm computes a block-wide *prefix-sum* on the work-items and stores the resulting prefix-sum array in the shared memory. Finally, all threads of the block get use of such an array to access to the corresponding work-unit. If the array size exceeds the shared memory space, the algorithm iterates.

This strategy provides a perfect balancing among threads and warps. On the other hand, the strip-mined gathering procedure run at each iteration introduces a significant overhead, which slows down the application performance in case of quite regular workloads. The strategy well applies only in case of very irregular workloads.

Fig. 4.5 shows an example of the three phases of the algorithm in which the *CTA* phase computes the largest work-item in one iteration, the *Warp* phase is applied on work-items greater than three, and the *Scan* phase computes the remaining work-units in two steps.

4.1.3 Dynamic mapping techniques

Contrary to *static mapping*, the *dynamic mapping* approaches achieve perfect workload partition and balancing among threads at the cost of additional computational overhead at run time. The core of such a computation is the *binary search* over the prefix-sum array. The binary search aims at mapping work-units to the corresponding threads.



Thread Id (From–To)	Work-item
0 – 2	0
3	1
4 – 8	2
9 – 10	3
11 – 17	4
18 – 21	6
22-24	7
25-27	8

FIG. 4.6: *Example of assignment of thread th_5 to work-item 2 through binary search over the prefix-sum array (a), and overall threads-items mapping (b).*

Direct Search

Given the *exclusive prefix-sum* array of the work-unit addresses stored in global memory, each thread performs a binary search over the array to find the corresponding *work-item* index (Fig. 4.6 shows an example). This technique provides perfect balancing among threads (i.e., one work-unit is mapped to one thread), warps and blocks of threads. Nevertheless, the large size of the array involves an arithmetic intensive computation (i.e., $\#threads \times binarysearch()$) and the binary search performed by the threads to solve the mapping to be very scattered. This often eludes the benefit of the provided balancing.

Local Warp Search

To reduce both the binary search computation and the scattered accesses to the global memory, this technique first loads chunks of the prefix-sum array from the global to the shared memory. Each chunk consists of 32 elements, which are loaded by 32 warp threads through a coalesced memory access. Then, each thread of the warp performs a lightweight binary search (i.e., maximum $\log_2(\mathit{WarpSize})$ steps) over the corresponding chunk in the shared memory.

In the context of graph traversal, this approach has been further improved by exploiting data locality in registers [59]. Instead of working on shared memory, each warp thread stores the workload offsets in the own registers and then performs a binary search by using *Kepler* warp-shuffle instructions [202].

In general, the local warp search strategy provides a fast work-units to threads mapping and guarantees coalesced accesses to both the prefix-sum array and work-units in global memory. On the other hand, since the sum of work units included in each chunk of prefix-sum array is greater than the warp size, the binary search on the shared memory (or registers for the enhanced version for Kepler) is repeated until all work-units are processed. This leads to more work-units to be mapped to the same thread. Although this technique guarantees a fair balancing among threads of the same warp, it suffers from work unbalancing between different warps since the sum of work-units for each warp can be not uniform in general. For the same reason, it does not guarantee balancing among blocks of threads.

Block Search

To deal with the local warp search limitations, Davidson et al. [83] introduced the block search strategy through *cooperative blocks*. Instead of warps performing 32-element loads, in this strategy each block of threads loads a *maxi chunk* of prefix-sum elements from the global to the shared memory, where the maxi chunk is as large as the available space in shared memory for the block. The maxi chunk size is equal for all blocks. Each maxi chunk is then partitioned by considering the amount of work-units included and the number of threads per block. For example, considering that the nine elements of the prefix-sum array of Fig. 4.1 exactly fits the available space in shared memory and that each block is sized 4 threads (for the sake of clarity), the maxi chunk will be partitioned into 4 slots, each one including 7 work-units. Finally, each block thread performs only one binary search to find the corresponding slot.

With the block search strategy, all the units included in a slot are mapped to the same thread. As a consequence, all the threads of a block are perfectly balanced. The binary searches are performed in shared memory and the overall amount of searches is significantly reduced (i.e., they are equal to the block size). Nevertheless, this strategy does not guarantee balancing among different blocks. This is due to the fact that the maxi chunk size is equal for all the blocks, but the chunks can include a different amount of work-units. In addition, this strategy does not guarantee memory coalescing among threads when they access to the assigned work-units. Finally, this strategy cannot exploit advanced features for intra-warp communication and synchronization among threads, such as, Kepler warp shuffle instructions.

Two-phase Search

Davidson et al. [83], Green et al [113] and Baxter [30] proposed three equivalent methods to deal with the inter-block load unbalancing. All the methods rely on two phases: *partitioning* and *expansion*.

First, the whole prefix-sum array is partitioned into *balanced* chunks, i.e., chunks that point to the same amount of work-units. Such an amount is fixed as the biggest multiple of the block size that fits in shared memory. As an example, considering blocks of 128 threads, two prefix-sum chunks pointing to 128 $\times K$ units, and 1,300 slots in shared memory, K is set to 10. The chunk size may differ among blocks (see for example Fig. 4.1, in which a prefix-sum chunk of size 8 points to 28 units). The partition array, which aims at mapping all the threads of a block into the same chunk, is built as follows. One thread per block runs a binary search on the whole prefix-sum array in global memory by using the own global id times the block size ($TH_{global\_id} \times blocksize$). This allows finding the chunk boundaries. The number of binary searches in global memory for this phase is equal to the number of blocks. The new partition array, which contains all the chunk boundaries, is stored in global memory.

In the expansion phase, all the threads of each block load the corresponding chunks into the shared memory (similarly to the dynamic techniques presented in the previous sections). Then, each thread of each block runs a binary search in such a local partition to get the (first) assigned work-unit. Each thread sequentially accesses all the assigned work units in global memory. The number of binary searches for the second step is equal to the block size. Fig. 4.7 shows an example of expansion phase, in which three threads (t_0, t_1, and t_2) of the same warp access to the local chunk of prefix-sum array to get the corresponding starting point of assigned work-unit. Then, they sequentially access the corresponding K assigned units ($A_1 - D_1$ for t_0, $D_2 - F_2$ for t_1, etc.) in global memory.

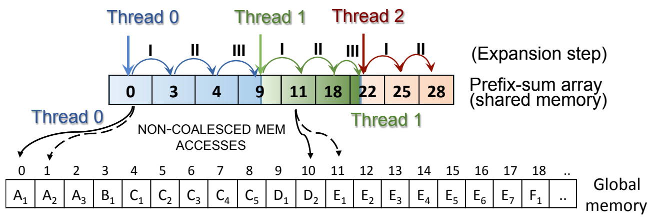

Thread 0
Thread 1
Thread 2
I
II
III
I
II
III
I
II
(Expansion step)
0
3
4
9
11
18
22
25
28
Prefix-sum array (shared memory)
Thread 0
Thread 1
NON-COALESCED MEM ACCESSES
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 ..
A_1 A_2 A_3 B_1 C_1 C_2 C_3 C_4 C_5 D_1 D_2 E_1 E_2 E_3 E_4 E_5 E_6 E_7 F_1 ..
Global memory

FIG. 4.7: *Example of expansion phase in the two-phase strategy (10 work-units per thread)*

In conclusion, the two-phase search strategy allows the workload among threads, warps, and blocks to be perfectly balanced at the cost of two series of binary searches. The first is run in global memory for the partitioning phase, while the second, which mostly affects the overall performance, is run in shared memory for the expansion phase. The number of binary searches for partitioning is proportional to the K parameter. High values of K involve less and bigger chunks to be partitioned and, as a consequence, less steps for each binary search. Nevertheless, the main problem of such a dynamic mapping technique is that the partitioning phase leads to very scattered memory accesses of the threads to the corresponding work-units (see lower side of Fig. 4.7). Such a problem worsens by increasing the K value.

4.2 Graph Traversal and Breadth-First Search

Several solutions for GPUs have been proposed in the last decade to accelerate graph traversal. Harish et al. [118], Hong et al. [128] and Jia et al. [135] presented the first solutions that exploit both node and edge parallelism to inspect every vertices/edges for each BFS iteration. Since they do not require to maintain additional data structures, they involve very simple implementations but, on the other hand, they perform quadratic work. Nevertheless, the proposed approach leads to a sensible workload imbalance whenever the graph is non homogeneous in terms of vertex degree. In addition, let D be the graph diameter, the computational complexity of such a solution is $O(VD + E)$, where $O(VD)$ is spent to check the frontier vertices and $O(E)$ is spent to explore each graph edge. While this approach fits on dense graphs, in the worst case of sparse graphs (where $D = O(V)$) the algorithm has a complexity of $O(V^2)$. This implies that, for large graphs, the proposed algorithm is slower than the sequential version of the algorithm.

The authors in [289] presents an alternative solution based on matrices for sparse graphs. Each frontier propagation is transformed into a matrix-vector multiplication. Given the total number of multiplications D (which corresponds to the number of levels), the computational complexity of the algorithm is $O(V + ED)$, where $O(V)$ is spent to initialize the vector, and $O(E)$ is spent for the multiplication at each level. In the worst case, that is, with $D = O(V)$ the algorithm complexity is $O(V^2)$.

More recent research focused on efficient algorithms for linear-work graph traversal. Luo et al. [172] described the first work-efficient BFS implementation based on hierarchical vertex queue management and global synchronization. The proposed solution shows significant speedup compared to the quadratic algorithms for graphs with high diameter.

Merrill et al. [190] presented a high-performance solution (B40C) that combines three different workload balancing techniques (CTA+Warp+Scan) to process vertices according to their degrees, duplicate removing, and a bitmask for the status lookup. The authors also proposed three strategies to organize the frontier exploration through vertex, edge, and two-phase queues.

Busato et al. [60] proposed an efficient BFS implementation specialized for Kepler architecture (BFS-4K), which exploits advanced device features such as dynamic parallelism, warp shuffle, and shared memory bank organization.

Bisson et al. [42] presented a BFS solution for distributed multi-node GPU platforms, which relies on a binary search algorithm to achieve perfect load balancing among all device threads. Davidson et al. [83] proposed another remarkable solution for load balancing, which relies on binary search even though at different thread hierarchy levels (i.e., warp, block, and device) in the context of the single-source shortest path (SSSP) problem.

Wang et al. [275] presented an optimized and flexible GPU graph library (Gunrock) that provides an high-level abstraction to reduce the developing effort of graph primitive programming. The Gunrock library relies on filtering operations and on two complementary thread mapping strategies selected according to the frontier size.

Beamer et al. [33] presented a hybrid top-down/bottom-up BFS solution focused on low-diameter and scale-free graphs. The algorithm visits the graph by propagating the exploration in the reverse direction when the number of unvisited edges is less than the frontier edges. The hybrid approach significantly reduces the exploration work as it does not necessarily explore all the graph edges. With a similar top-down/bottom-up approach, Liu et al. [164] proposed an efficient BFS solution for GPU architectures that relies on a quadratic-work graph traversal. Even though very efficient, such hybrid solutions provide advantages only on graphs with well-defined characteristics, they require double memory space to store the inverse graphs, and they can be applied to a subset of problems compared to the more standard top-down solutions. As an example, the algorithms single-source shortest path (SSSP), betweenness centrality (BS), cluster coefficient (CC), and other graph procedures that require propagating values along edges cannot be implemented through a bottom-up BFS visit due to such a partial visit of the graph edges.

4.3 Single-Source Shortest Path

At the state of the art, the reference approaches to SSSP are the Dijkstra's [91] and Bellman-Ford's [38, 102] algorithms. These two classic algorithms span a parallel vs. efficiency spectrum. Dijkstra's allows the most efficient ($O(|V|\log|V| + |E|)$) sequential implementations [70, 290] but exposes no parallelism across vertices. Indeed, the solutions proposed to parallelize the Dijkstra's algorithm for GPUs have shown to be asymptotically less efficient than the fastest CPU implementations [180, 215]. On the other hand, at the cost of a lower efficiency ($O(|V||E|)$), the Bellman-Ford's algorithm has shown to be more easily parallelizable for GPUs, by providing speedups up to two orders of magnitude compared to the sequential counterpart [56, 118].

Meyer and Sanders [192] proposed the Δ-stepping algorithm, a trade-off between the two extremes of Dijkstra and Bellman-Ford. The algorithm involves a tunable parameter Δ, whereby setting $\Delta = 1$ yields a variant of Dijsktra's algorithm, while setting $\Delta = \infty$ yields the Bellman-Ford algorithm. By varying Δ in the range $[1, \infty]$, we get a spectrum of algorithms with varying degrees of processing time and parallelism. Crobak et al. [76] and Chakaravarthy et al. [64] presented two different solutions to efficiently expose parallelism of this algorithm on the massively multi-threaded shared memory system IBM Blue Gene/Q.

Parallel SSSP algorithms for multi-core CPUs have been also proposed by Kelley and Schardl [140], who presented a parallel implementation of Gabow's scaling algorithm [105] that outperforms Dijkstra's on random graphs. Shun and Blelloch [242] presented a Bellman-Ford's scalable parallel implementation for CPUs on a 40-core machine. Recently, several packages have been developed for processing large graphs on parallel architectures including the parallel *Boost* graph library [96], Pregel [177] and Pegasus [86].

In the context of GPUs, Martin et al. [180] and Ortega et al. [215] proposed two different solutions to parallelize the Dijsktra's algorithm. Although both the solutions provide a good speedup in many cases, they have shown to be asymptotically less efficient than the fastest CPU implementations due to the intrinsic sequential nature of the Dijsktra's algorithm.

In contrast, Harish et al. [118] and Burtscher et al. [56] proposed two different parallel implementations of the Bellman-Ford's algorithm. Both the solutions always provide good speedups with respect to the sequential counterpart and, in any case, speedups higher than the Dijkstra's solutions. Nevertheless, they showed to have a poor work efficiency since they only target to performance.

Davidson et al. [82] proposed three different work-efficient solutions for the SSSP problem. *Workfront Sweep* implements a queue-based Bellman-Ford algorithm that reduces redundant work due to duplicate vertices during the frontier propagation. Such a fast graph traversal method relies on the merge path algorithm [214], which equally assigns the outgoing edges of the frontier to the GPU threads at each algorithm iteration. *Near-Far Pile* refines the Workfront Sweep strategy by adopting two queues similarly to the Δ-Stepping algorithm. Davidson et al. [82] also propose the *bucketing* method to implement the Δ-Stepping algorithm. Δ-Stepping algorithm is not well suited for SIMD architectures as it requires dynamic data structures for buckets. However, the authors provide an algorithm implementation based on sorting that, at each step, emulates the bucket structure. The Bucketing and Near-Far Pile strategies heavily reduce the amount of redundant work compared to the Workfront Sweep method but, at the same time, they introduce overhead for handling more complex data structure (i.e., frontier queue). These strategies are less efficient than the sequential implementation on graphs with large diameter since they suffer from thread under-utilization caused by such unbalanced graphs.

4.4 Strongly Connected Components

Data structures encoding the graph have to allow independent thread-local data processing and coalesced access. The adjacency list representation is typically encoded as two one-dimensional arrays [117]. One array keeps the target vertices of all the edges. The second array keeps an index to the first array for every vertex. The index points to the position of the first edge emanating from the corresponding vertex. Other data associated to a vertex are organized in vectors as well. In [24, 89, 162], techniques for improving memory consumption and access pattern for SCC decomposition algorithms have been proposed.

The core procedure of every graph algorithm is the graph traversal. The SCC decomposition algorithms build on several types of the traversal as explained in the next section. Parallelization of this procedure fundamentally affects the overall performance of the decomposition. There exist several approaches [57,117,125,190] that differ in the granularity of the task allocation (thread-per-vertex vs. warp-per-vertex vs. block-per-vertex) and in the number of vertices/edges processed during a single kernel (linear vs. quadratic parallelization). In the context of the SCC decomposition the performance of these approaches significantly depends on the structure of the graphs and the type of the traversal.

4.4.1 Forward-Backward algorithm

The FORWARD-BACKWARD (FB) algorithm [101] represents the fundamental algorithm for parallel SCC decomposition. It is listed as Algorithm 1 and proceeds as follows. A vertex called *pivot* is selected and the strongly connected component the pivot belongs to is computed as the intersection of the forward and backward *closure* of the pivot. Computation of the closures divides the graph into four subgraphs that are all SCC-closed. These subgraphs are 1) the strongly connected component with the pivot, 2) the subgraph given by vertices in the forward closure 3) the subgraph given by vertices in the backward closure , and 4) the subgraph given by the remaining vertices. The later three subgraphs form independent instances of the same problem, and therefore, they are recursively processed in parallel. The time complexity of the FB algorithm is $\mathcal{O}(n \cdot (m+n))$ since it performs $\mathcal{O}(m+n)$ work to detect a single strongly connected component.

Practical performance of the algorithm may be further improved by performing elimination of leading and terminal trivial strongly connected components – the so-called *trimming* [184]. The TRIMMING procedure builds upon a topological sort elimination. A vertex cannot be part of a non-trivial strongly connected component if its in-degree (out-degree) is zero. Therefore, such a vertex can be safely removed from the graph as a trivial SCC, before the pivot vertex is selected. The elimination can be iteratively repeated until no more vertices with zero in-degree (out-degree) exist.

In [24] is designed a GPU-acceleration of the FB algorithm that provides a good performance and scalability on regular graphs. In [162] the acceleration is improved by the linear parallelization of the graph traversal and by a better pivot selection, which result in a performance gain including also a good performance on less regular graphs. The main limitation of the FB algorithm is that it performs $\mathcal{O}(m+n)$ work to detect a single SCC. This mitigates the benefits of the GPU-acceleration if the graph contains many small but non-trivial components.

Algorithm 1 FB

```
  FB(V)
1:   pivot ← PivotSelection(V)
2:   F ← FWD-Reach(pivot, V)
3:   B ← BWD-Reach(pivot, V)
4:   F ∩ B is SCC
5:   In parallel do
6:       FB(F \ B)
7:       FB(B \ F)
8:       FB(V \ (B ∪ F))
```

4.4.2 Coloring algorithm

The COLORING algorithm [216] is capable of detecting many strongly connected components in a single recursion step, however, for the price of an $\mathcal{O}(n \cdot (m+n))$

Algorithm 2 COLORING

```
    Coloring(V)
1:  $(\mathrm{maxcolor}, \mathrm{V_k}) \leftarrow$ FDW-MAXCOLOR(V)
2:  Parallel for $k \in$ maxColor do
3:      $B_k \leftarrow \mathrm{BWD}(k, V_k)$
4:      $B_k$ is SCC
5:      if $(V_k \backslash B_k \neq 0)$ then
6:          COLORING$(V_k \backslash B_k)$
```

procedure. Therefore, the time complexity of the algorithm is $\mathcal{O}((l+1) \cdot n \cdot (m+n))$ where l is the longest path in the component graph.

The pseudo-code of the algorithm is listed as Algorithm 2. It propagates unique and totally ordered identifiers (colors) associated with vertices. Initially, each vertex keeps its own color. The colors are iteratively propagated along edges of the graph (line 2:1) so that each vertex keeps only the maximum color among the initial color and colors that have been propagated into it (maximal preceding color). After a fixpoint is reached (no color update is possible), the colors associated with vertices partition the graph into multiple SCC-closed subgraphs V_k. All vertices of a subgraph are reachable from the vertex v whose color is associated with the subgraph. Therefore, the backward closure of v restricted to the subgraph forms a SCC component that is removed from the graph before the next recursion step. Propagation procedure is rather expensive if there are multiple large components which limits the overall performance [24].

4.4.3 Other algorithms

Both the presented algorithms typically show limited performance and poor scalability when applied to large real-world graph instances with many nontrivial components and a high diameter. Fundamental properties of these graphs have been consider to propose a series of extensions of the FB algorithm [126] and a multi-step algorithm [250] that adequately combines the FB and COLORING algorithms. These two, originally multi-core, algorithms have been recently redesigned to allow data-parallel processing [89], which led to the fastest GPU-accelerated SCC decomposition.

Barnat et al. [25] introduced the OBF algorithm that aims at decomposing the graph in more than three SCC-closed subgraphs within a single recursion step. However, unlike the COLORING algorithm, the price of the OBF procedure is $\mathcal{O}(m + n)$. Despite the better asymptotic complexity, the work in [24] attempt indicate that effective data-parallelization of the OBF algorithm is a very hard problem and the approaches based on the multi-step algorithm performs generally better on SIMT-based architectures.

Very recently a multi-core version of the Tarjan algorithm based on parallelization of depth-first search [47] has been proposed. It preserves the liner complexity of SCC decomposition and on a variety of graph instances it outperforms previous multi-core solutions. However, on real-word graphs it considerably lags behind the approaches by [126,250] and the proposed parallelization is principally not suitable for SIMT architectures.

5

Load Balancing - Multi-Phase Search Algorithm

Workload partitioning and the subsequent work item-to-thread mapping are key aspects to face when implementing any efficient GPU application. Different techniques have been proposed to deal with such issues, ranging from the computationally simplest static to the most complex dynamic ones. Each of them finds the best use depending on the workload characteristics (static for more regular workloads, dynamic for irregular workloads). Nevertheless, no one of them provides a sound trade off when applied in both cases. Static approaches lead to load unbalancing with irregular problems, while the computational overhead introduced by the dynamic or semi-dynamic approaches often worsens the overall application performance when run on regular problems. This Section presents an efficient dynamic technique for workload partitioning and work item-to-thread mapping whose complexity is significantly reduced with respect to the other dynamic approaches in literature. The Section shows how the partitioning and mapping algorithm has been implemented by fully taking advantage of the GPU device characteristics with the aim of minimizing the involved computational overhead. The Section shows, compares, and analyses the experimental results obtained by applying the proposed approach and several static, dynamic, and semi-dynamic techniques at the state of the art to different benchmarks and over different GPU technologies (i.e., NVIDIA Fermi, Kepler, and Maxwell) to understand when and how each technique best applies.

5.1 Introduction

Partitioning a workload and mapping work items to threads are correlated important issues to face when structuring and implementing any parallel application. In the context of GPU applications, these tasks are generally implemented by exploiting scan-based operations [44, 46]. Given a list of input values and a binary associative operator, a *prefix-scan* procedure computes a list of elements in which each element is the reduction of the elements occurring earlier in the input list [40, 93, 186, 240]. When the operator is the addition, the prefix-scan represents a *prefix-sum*, which is useful when parallel threads have to allocate dynamic data within shared data structures such as global queues [74].

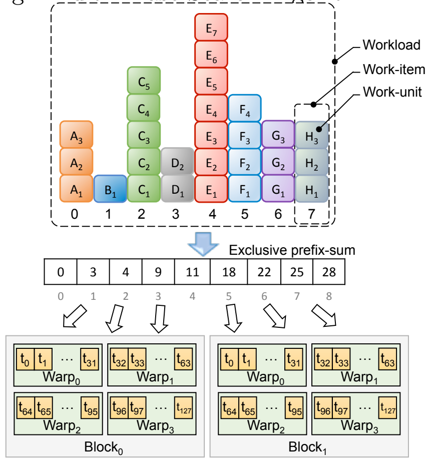


FIG. 5.1: *Overview of the load balancing problem in the workload partitioning and mapping to threads of scan-based applications*

Given a workload to be allocated over the GPU threads (see Fig. 5.1), prefix-sum is applied to efficiently calculate the offset for each thread to access to the corresponding work-items (coarse-grained mapping) or work-units (fine-grained mapping) [190]. Nevertheless, even though prefix-scan operations allows the threads to efficiently access in parallel to the corresponding data, they are not enough to solve the load balancing problem. Indeed, the workload decomposition and mapping strategies are left to the application designer. How the application implements such a mapping can have a significant impact on the overall application performance.

Different techniques have been presented in literature to decompose and map the workload to threads through the use of prefix-sum data structures [30, 59, 83, 113, 117, 125, 190]. All these techniques differ from the complexity of their implementation and from the overhead they introduce in the application execution to address the most irregular workloads. In particular, the simplest solutions [117, 125] apply well to very regular workloads while they cause strong unbalancing and, as a consequence, lost of performance in case of irregular workloads. More complex solutions [30,59,83,113,190] best apply to irregular problems through semi-dynamic or dynamic workload-to-thread mappings. Nevertheless, the overhead introduced for such a mapping often worsens the overall application performance when run on regular problems.

This Section first presents an accurate analysis of the most important and widespread load balancing techniques existing in the literature based on prefix-scan, by underlining their advantages and drawbacks over different workload characteristics. The analysis includes details on their coalescing issues involved during the memory accesses both to the prefix-sum structure and to the global memory, which are strictly related to the strategy implementation.

Then the Section presents an efficient dynamic partitioning and mapping technique, called *Multi-phase Mapping*, to address the workload unbalancing problem in both regular and irregular problems and how it has been implemented by fully exploiting the GPU device characteristics. In particular, *Multi-phase Mapping* implements a dynamic mapping of work-units to threads through an algorithm whose complexity is significantly reduced with respect to the other dynamic approaches in the literature. This allows the proposed approach to efficiently handle irregular problems and, at the same time, to provide good performance also when applied to very regular and balanced workloads.

The Section presents the experimental results obtained by applying all the analyzed techniques and *Multi-phase Mapping* to different benchmarks and over different GPU technologies (i.e., NVIDIA Fermi, Kepler, and Maxwell) to understand when and how each technique best applies.

The work is organized as follows. Section 5.2 presents the proposed multi-phase mapping technique. Section 5.4 presents the experimental results and their analysis, while Section 5.5 is devoted to the conclusions.

5.2 The proposed Multi-phase Mapping

The proposed strategy aims at exploiting the balancing advantages of the two-phase algorithms while overcoming the limitations of the scattered memory accesses. It consists of a *hybrid partitioning phase* and an *iterative coalesced expansion.*

5.2.1 Hybrid partitioning

Differently from the dynamic techniques in literature, which strongly rely on the binary search (see Section 4.1.3), the proposed approach relies on a hybrid partitioning strategy by which each thread searches the own work-items. Such a hybrid strategy dynamically switches between an *optimized binary search* and an *interpolation search* depending on the benchmark characteristics.

Optimized binary search

In the standard implementation of the binary search, each thread finds the searched element, on a prefix-sum array of N elements, through one memory access in the best case or through $2 \log N$ memory accesses in the worst case (see the example of Fig. 4.6). Indeed, at each iteration, each thread performs two memory accesses, to check the lower bound (value at the left of the index) and the upper bound (value at the right of the index) to correctly update the index for the next iteration. Nevertheless, in the context of binary search on prefix-sum, since all threads must be synchronized by a barrier before moving to the next iteration, and since at least one thread executes all iterations involving $2 \log N$ memory accesses, each binary search actually has a time complexity equal to $2 \log N$ memory accesses.

In the proposed approach, each thread checks, at each iteration, only the lower bound, thus involving only one memory access per iteration. On the other hand, this approach requires all the threads to perform all iterations ($\log N$) indistinctly. Overall, such an optimization halves the binary search complexity to $\log N$ memory accesses.

```
**Interpolation Search** $(Array, left, right, S)$
1:  while $S \geqslant Array[\text{left}]$ and $S \leqslant Array[\text{right}]$ do
2:      $K = \text{left} + (S - Array\,[\text{left}]) \cdot \frac{\text{right}-\text{left}}{Array[\text{right}]-Array[\text{left}]}$
3:      if $Array\,[K] < S$ then
4:          $left = K + 1$
5:      else if $Array\,[K] > S$ then
6:          $right = K - 1$
7:      else
8:          return $K$
9:      end
10: end
```

Interpolation search

In case of uniformly distributed inputs (i.e., low standard deviation of work-item size) and a low average number of work-units, the proposed approach implements an *interpolation search* [219] in alternative to the optimized binary search. The interpolation search has a very low complexity ($O(\log\log N)$) at the cost of additional computation. The algorithm pseudocode is the following:

The idea is to use information about the underlying distribution of data to be searched in a human-like fashion when searching a word in a dictionary. Given a chunk of prefix-sum elements (*Array*) and the item to be searched (S), the procedure iteratively calculates the next search position K (row 2 of the algorithm) by mapping S in the distribution *Array*[left], *Array*[right]. The algorithm shows an average number of comparisons equal to $O(\log\log n)$ that increase to $O(N)$ in the worst case, differently to the binary search that shows complexity $O(\log N)$ in all cases.

The main drawback is the higher computational cost to calculate the next index of the search (row 2), which involves double precision floating-point operations (division, multiplication, and casting). Such operations present a very low arithmetic throughput in GPU devices compared with single precision operations. To limit such a cost, Multi-Phase Search implemented the computation by minimizing the expensive double precision operations and by replacing them with 64-bit integer operations when possible.

The proposed hybrid approach switches between interpolation and binary search depending on the benchmark characteristics. In particular, the interpolation search runs if the following conditions hold:

$Std\_Dev\_WI_{size} \leqslant Threshold_{SD}$
and
$Average\_WI_{size} \leqslant Threshold_{AVG}$

where the standard deviation of the work-item size and the average work-item size of the benchmark are calculated runtime. The switching between the two methods is parametrized through the two thresholds that, as explained in the experimental results, have been heuristically set to $Threshold_{SD} = 8$ and $Threshold_{AVG} = 9$ for all the analysed benchmarks.

5.2.2 Iterative Coalesced Expansion

In the expansion phase, all the threads of each block load the corresponding chunks into the shared memory (similarly to the dynamic techniques presented in the previous sections). Then, each thread performs an binary search (optimized as in the partitioning phase presented in Section 5.2.1) in such a local partition to get the assigned work-unit.

Then, the expansion phase consists of three iterative sub-phases, by which the scattered accesses of threads to the global memory are reorganized into coalesced transactions. This is done in shared memory and by taking advantage of local registers:

1. *Writing on registers.* Instead of sequentially writing on the work-units in global memory, each thread sequentially writes a small amount of work-units in the local registers. Fig. 5.2 shows an example. The amount of units is limited by the available number of free registers.
2. *Shared mem. flushing and data reorganization.* After a thread block synchronization, the local shared memory is flushed and the threads move and reorder the work-unit array from the registers to the shared memory.
3. *Coalesced memory accesses.* The whole warp of threads cooperate for a coalesced transaction of the reordered data into the global memory. It is important to note that this step does not require any synchronization since each warp executes independently on the own slot of shared memory.

Steps two and three iterate until all the work-units assigned to the threads are processed. Even though these steps involve some extra computation with respect to the direct writings, the achieved coalesced accesses in global memory significantly improve the overall performance.

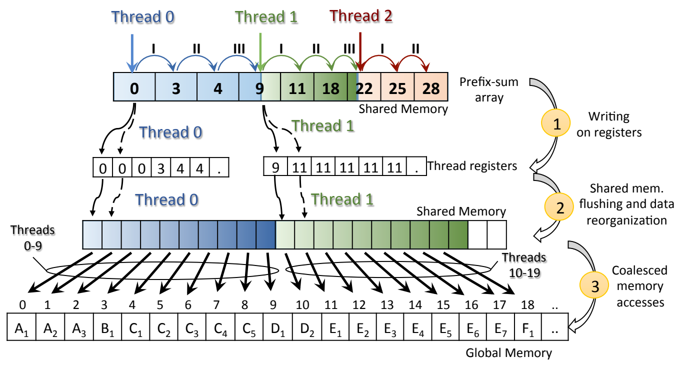


FIG. 5.2: *Overview of the coalesced expansion optimization (10 work-units per thread)*

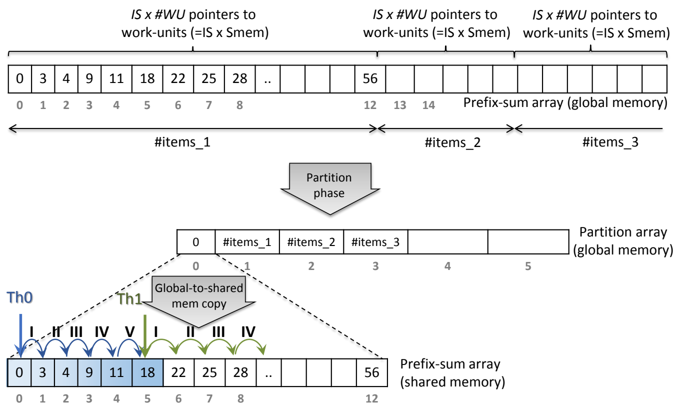

IS x #WU pointers to work-units (=IS x Smem)
IS x #WU pointers to work-units (=IS x Smem)
IS x #WU pointers to work-units (=IS x Smem)
Prefix-sum array (global memory)
#items_1
#items_2
#items_3
Partition phase
Partition array (global memory)
Global-to-shared mem copy
Th0
Th1
Prefix-sum array (shared memory)

FIG. 5.3: *Overview of the iterated search optimization (10 work-units per thread and IS=2)*

The shared memory size and the size of thread blocks play an important role in the coalesced expansion phase. The bigger the block size, the shorter the partition array stored in shared memory. On the other hand, the bigger the block size, the more the synchronization overhead among the block warps, and the more the binary search steps performed by each thread (see final considerations of the Two-phase search in Section 4.1.3).

In particular, the overhead introduced to synchronize the threads after the register writing (see sub-phase 1) is the bottleneck of the expansion phase (each register writing step requires two thread barriers). To reduce such an overhead, we propose an *iterative search* optimization as follows:

1. In the partition phase, the prefix sum array is partitioned into balanced chunks (see Fig. 5.3). Differently from the two-phase search strategy, the size of such chunks is fixed as a multiple of the available space in shared memory:
$$ChunkSize = BlockSize \times K \times IS$$
where $BlockSize \times K$ represents the biggest number of work-units (i.e., a multiple of the block size) that fits in shared memory (as in the two-phase algorithm), while IS represents the *iteration factor*. The number of threads required in this step decreases linearly with IS.
2. Each block of threads loads from global to shared memory a chunk of prefix-sum, performs the function initialization, and synchronizes all threads.
3. Each thread of a block performs IS binary searches on such an extended chunk.
4. Each thread starts with the first step of the coalesced expansion (upper-side of Fig. 5.3), i.e., it sequentially writes an amount of work-units in the local registers. Such an amount is IS times larger than in the standard two-phase strategy.
5. The local shared memory is flushed and each thread moves a portion of the extended work-unit array from the registers to the shared memory. The portion size is equal to $BlockSize \times K$. Then, the whole warp of threads cooperate for a coalesced transaction of the reordered data into the global memory, as in

the coalesced expansion phase presented in Section 5.2.2. This step iterates *IS* times, until all the data stored in the registers has been processed.

The iterative search optimization reduces the number of synchronization barriers by a factor of $2 * IS$, avoids many block initializations, decreases the number of required threads, and maximizes the shared memory utilization during the loading of the prefix-sum values with larger consecutive intervals. Nevertheless, the required number of registers grows proportionally to the *IS* parameter. Considering that the maximum number of registers per thread is a fixed constraint for any GPU device (e.g., 32 for NVIDIA Kepler devices) and that exceeding such a constraint involves data to be *spilled* in L1 cache and then in L2 cache or global memory, too high values of *IS* may compromise the overall performance of the proposed approach.

5.2.3 Optimizing the Multi-Phase implementation

The proposed algorithm achieves perfect load balancing and overcomes the limitations related to scattered memory accesses and synchronization overhead. In addition, the algorithm structure is particularly well suited for advanced optimizations targeting GPU architectures, which aim at reducing the computational workload, simplifying the overall execution flow, and improving the memory access pattern.

Full loop unrolling and instruction-level parallelism.

Loop unrolling is a common technique widely applied by sequential code compilers to reduce the number of branch-related instructions. Since GPU compilers cannot always guarantee such an optimization (while preserving the semantics correctness), loop unrolling has been forced in *Multi-phase Mapping*, through `#PRAGMA UNROLL` directives where possible, to take advantage of instruction level parallelism (ILP) on the GPU device [271].

Loop unrolling has been forced in the coalesced expansion phase: (i) in the chunk loading into shared memory and (ii) in the subsequent iterative subphases (writing on registers, shared memory flushing, and coalesced memory accesses). Indeed, loop unrolling in these phases can be applied since all threads perform the same number of loop iterations and such a number is known at compile time.

Data and Pointer Hoisting.

Similarly to the loop unrolling optimization, loop-invariant code motion has been forced to all kernel loops. It includes hoisting of data and address computations as in the example of Fig. 5.4.

Global data prefetching.

Data movement particularly affects the expansion phase of the proposed algorithm. Global-to-shared memory and shared-to-global memory data movement is

Before hoisting

```
for ( ; ; ) do
    x = y + z;
    devInput[blockIdx.x + i] = x * x;
end
```

After hoisting

```
devInput += blockIdx.x;
x = y + z;
t = x * x;
for ( ; ; ) do
    devInput[i] = t;
end
```

FIG. 5.4: *Example of data and pointer hoisting.*

optimized by introducing an additional intermediate step between the accesses to these memory spaces, as proposed in [29]. We exploited the thread registers as fast intermediate local memory, thus hiding the memory access latency.

Vectorized shared memory accesses.

The CUDA model provides vectorized memory accesses (up to 16-bytes per single transaction) to fully exploit the memory bandwidth. We implemented vectorized accesses in almost all steps that involve shared memory and during the expansion phase in global memory. The same technique cannot be applied to global memory loads since the offset of blocks in such a memory are not aligned.

Warp-synchronous programs.

In GPU computation, each thread warp executes in *lock-step* way and it does not require any explicit synchronization barrier to correctly preserve the semantics. In order to eliminate communications and explicit synchronizations also *between* warps, we organized the memory accesses by splitting the shared memory in chunks of the same size on which each warp can operate independently.

Scheduler overhead minimization.

Implementing the proposed partitioning and mapping approach requires a kernel structure similar to that shown in Alg. 3. The kernel mainly consists of (i) an initialization phase to declare and initialize data structures (rows 1-4), and (ii) the actual computational phase on the data structures (rows 6-8). The computational phase iterates (`for` loop in rows 5-10) if the grid size (number of thread blocks) are less than the generated chunks (see Section 5.2.2). The figure also shows three different kernel configurations. Considering that, in general, *WorkLoadSize* $\gg$ *ResidentThreads* (i.e., 3-4 orders of magnitude), configuration (a) generates much more blocks than the other configurations (b, c). This allows branch conditions involved by the loop construct and thread barriers (row 9) to be

Algorithm 3 Kernel structure and configurations.

Kernel implementation (device side):

```
$var = 0$                                                    \\ Initialization
thread_offset = ...                                          \\ Initialization
__shared__ SMem                                              \\ Initialization
...                                                          \\ Initialization
for (i = blockIdx.x; i < $\lceil \frac{WorkLoadSize}{ChunkSize} \rceil$; i += gridDim.x) do
    /* Computational phase
    through SMem accesses
    */
    __synchthreads()
end
```

Kernel configurations (host side):

(a) deviceFunc$\lll \lceil \frac{WorkLoadSize}{ChunkSize} \rceil, blockDim \ggg$()

(b) deviceFunc$\lll \lceil \frac{ResidentThreads}{BlockSize} \rceil, blockDim \ggg$()

(c) deviceFunc$\lll \lceil \frac{ResidentThreads}{BlockSize} \rceil \cdot \mathcal{K}, blockDim \ggg$()

avoided. On the other hand, the larger number of blocks also involves more overhead due to the initialization rows executed at each block context switch. Such an overhead increases linearly with the initialization activity and the number of generated blocks. The orthogonal configuration (b) generates a smaller number of blocks, reduces the block context switches but, on the other hand, involves loop iterations, branch conditions, and synchronization barriers. *Multi-phase Mapping* implements a trade-off solution (c), where the number of blocks of solution (c) is modulated by a constant, $\mathcal{K}$. As reported in the experimental results, solution (c) in which $\mathcal{K}$ has been heuristically set to 32, provided the best scheduler overhead minimization in all the analysed benchmarks.

Read-only cache and pointer aliasing.

Recent architectures introduce the read-only data cache [209, 211]. It is faster and larger than the L1 cache, but requires all data to be guaranteed read-only for the duration of the whole kernel and not to be overlapped with other output pointers (i.e., *restrict pointer*). We exploited the read-only cache to load the global prefix-scan into the local memory.

5.3 Comprehensive comparison of complexity and limiting factors of the approaches

To accurately compare *Multi-phase Mapping* and the existing counterparts, the Section presents an overall overview of the performance limiting factors and the complexity analysis of each approach core algorithm. Table 5.1 summarizes the main performance limiting factors ordered by relevance for each technique and

Technique	Sub-phase	Performance limiting factors
Work-it to Threads	\	Non-coalesced memory accesses, warp divergence
Virtual Warps	\	Non-coalesced memory accesses, warp divergence, block scheduling overhead, Nontrivial tuning
Dyn. Virtual Warps + Dyn. Parallelism	Dyn. Virtual Warps	Non-coalesced memory accesses, warp divergence, block scheduling overhead
	Dyn. Parallelism	Dynamic kernels overhead
Cta+Warp+Scan	CTA	Synchronization overhead
	Warp	\
	Scan	Available shared memory, synchronization overhead
Direct Search	\	Non-coalesced memory accesses, warp divergence
Local Warp Search	\	Non-coalesced memory accesses, compute intensive
Block Search	\	Non-coalesced memory accesses, synchronization overhead
Two-Phase Search	Partition	Non-coalesced memory accesses, warp divergence
	Expansion	Available shared memory
Multi-Phase Mapping	Partition - Binary Search	Non-coalesced memory accesses, warp divergence
	Partition - Interpolation Search	Compute intensive
	Expansion	Available shared memory

Table 5.1: Summary of the performance limiting-factors.

corresponding sub-phases. Non-coalesced memory accesses have a significant impact on the performance and penalize most of the procedures that do not implement an efficient cooperation among threads. Warp divergence heavily affects the whole partitioning phase of static techniques, while, in dynamic techniques, it is limited to the computation of the binary search. The amount of available shared memory also plays an important role from the performance point of view in all the techniques based on data locality. The partition phase of the *Multi-phase* and *Two-phase* techniques suffers from non-coalesced memory accesses and warp divergence. However, since this phase involves a small fraction of the overall computation, such limiting factors do not affect the overall performance significantly. In general, as shown in Table 5.1, *Multi-phase Mapping* presents several and different limiting factors. However, the impact of each factor in the corresponding sub-phase is significantly lower than that in the counterparts. This guarantees, as shown in the result section, the best overall performance for all the different dataset typologies.

Given a workload consisting of N work-items and a total number of W work-units, we express the complexity of each technique in terms of *work complexity* (i.e., the time required by a single-thread execution of the approach), *parallel complexity* (i.e., the time required by the parallel execution of the approach with a hypothetical infinite number of threads, also called *critical path*), *number of threads* required

Technique	Work Complexity	Parallel Complexity	N. of required threads	Coalesced accesses to prefix-sum array	Coalesced accesses to work-units
Work-it to threads	$O(W)$	$O(W_{\text{MAX}})$	N	Yes	No
Virtual Warp	$O(W)$	$O\left(\frac{W_{\text{MAX}}}{\lvert\text{VirtualWarp}\rvert}\right)$	$N \cdot \lvert\text{VirtualWarp}\rvert$	Yes	Partial
Dyn. Virtual Warps + Dyn. Parallelism	$O(W)$	$O\left(\frac{\text{Dyn}_{\text{Th}}}{\lvert\text{VirtualWarp}\rvert}\right)$	$N + \sum\limits_{W_i \geqslant \text{Dyn}_{\text{TH}}} W_i$	Yes	Partial/Yes
Cta+Warp+Scan	$O(W)$	$\max_i \begin{cases} W_i \geqslant \text{CTA}_{\text{TH}} & \frac{W_i}{\lvert\text{CTA}\rvert} \\ W_i \geqslant \text{Warp}_{\text{TH}} & \frac{W_i}{\lvert\text{Warp}\rvert} \\ \text{otherwise} & W_i \end{cases}$	N	Yes	Yes
Direct Search	$O(W \cdot \log N)$	$O(\log N)$	W	No	Yes
Local Warp Search	$O(W \cdot \log\lvert\text{Warp}\rvert)$	$O\left(\frac{\sum_{i\in\text{Warp}} W_i}{\lvert\text{Warp}\rvert} \cdot \log\lvert\text{Warp}\rvert\right)$	N	Yes	Yes
Block Search	$O\left(\frac{N\cdot\lvert Block\rvert}{\text{SMem}} \cdot \log \text{SMem} + \text{W}\right)$	$O\left(\log \text{SMem} + \frac{\sum_{\text{i}\in\text{Block}} \text{W}_\text{i}}{\lvert\text{Block}\rvert}\right)$	N	Yes	No

Table 5.2: Comprehensive comparison of complexity of the workload partitioning techniques. N is the number of work-items, W is the total number of work-units, W_i is the number of work-units of a single work-item, W_{MAX} is the maximum number of work-units among all work-items, and *SMem* is the available shared memory. The *Two-Phase Search* and *Multi-Phase Search* specify the complexity for the *Partition* and *Expansion* phases.

for the overall computation, and *coalesced memory accesses*. We distinguish the coalescing characteristics by specifying whether the technique performs coalesced accesses to the prefix-sum array (to load the work-unit addresses) and coalesced accesses to the work-units in global memory.

The work complexity allows us to understand the work efficiency of each technique, to be compared with the work complexity of the reference sequential technique (i.e., $O(W)$). The number of required threads allows us to understand how much each approach involves thread scheduling activity. This is particularly relevant when considering N much greater than the number of resident threads provided by the GPU device.

Table 5.2 reports the results. All the static and semi-dynamic techniques are *work-efficient*, as they achieve the same work complexity of the sequential algo-

Workload Source	Number of Rows/Columns	Number of nonzeros	Structure	Avg. work-item size	Std. Dev. work-item size	Max work-item size
great-britain_osm	7,733,822	16,313,034	symmetric	2.1	0.5	8
cit-Patents	3,774,768	16,518,948	asymmetric	4.8	7.5	770
web-NotreDame	325,729	1,497,134	asymmetric	5.2	21.4	3,445
regular8	1,000,000	8,000,000	asymmetric	8.0	0.0	8
circuit5M	5,558,326	59,524,291	asymmetric	10.7	1,356.6	1,290,501
as-Skitter	1,696,415	22,190,596	symmetric	13.1	136.9	35,455
kron_g500-logn20	1,048,576	89,239,674	symmetric	96.2	1,033.1	413,378

Table 5.3: Benchmark Characteristics

rithm. On the other hand, they present important differences in the parallel complexity, due to the different strategies adopted to deal with the workload unbalancing. In the static and semi-dynamic mapping classes, only *CTA+Warp+Scan* achieves coalesced accesses on both the prefix-sum array and work-units. The techniques based on virtual warps achieve coalesced accesses on the prefix-sum array only among threads of the same group. The *Dynamic Virtual Warp + Dynamic Parallelism* technique allows for fully coalesced accesses only in the *child kernels* invocations (i.e., with problems with very high average of work-items size).

All the dynamic techniques pay extra overhead in work complexity to uniformly distribute the workload among GPU threads, but, on the other hand, their parallel complexity is always logarithmic in the input size. *Two-phase Search* and *Multi-phase Mapping* have the same parallel complexity, but only the latter achieves full memory coalescing. In addition, thanks to the *iterative search* (see Section 5.2.2), *Multi-phase Mapping* improves the work complexity and the number of required threads by a term of IS both in the partition and expansion phases.

All techniques do not require extra (global) memory space in addition to the input and output data, except for *Two-phase Search* and *Multi-phase Mapping* that need $\frac{W}{\text{SMem}}$ and $\frac{W}{\text{SMem}\cdot\text{IS}}$ additional bytes, respectively, to store the partition array. Finally, *CTA+Warp+Scan*, *Two-phase Search*, and *Multi-phase Mapping* take advantage of the shared memory to address the load balancing among threads of the same block. As a consequence, they best apply in GPU devices with a large amount of shared memory.

5.4 Experimental results

We tested the load balancing efficiency of all the techniques presented in Section 4.1 and *Multi-phase Mapping* over different benchmarks, whose characteristics are reported in Table 5.3. The benchmarks have been selected from *The University of Florida Sparse Matrix Collection* [84], which consists of a huge set of data representation from different contexts (e.g., circuit simulation, molecular dynamic, road networks, linear programming, vibroacoustic, web-crawl). The six benchmarks have been selected among the whole collection to cover very different data characteristics in terms of average work-item size, standard deviation from the item size, and maximum work-item size. As summarized in the table, they span

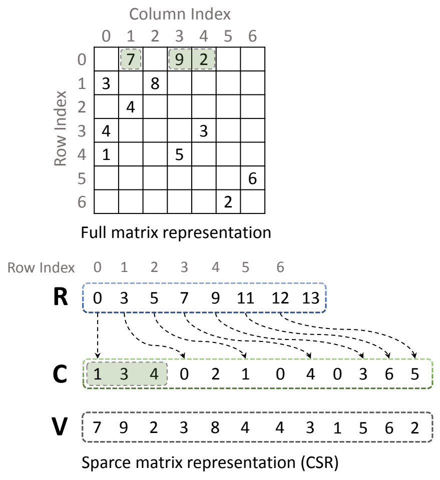
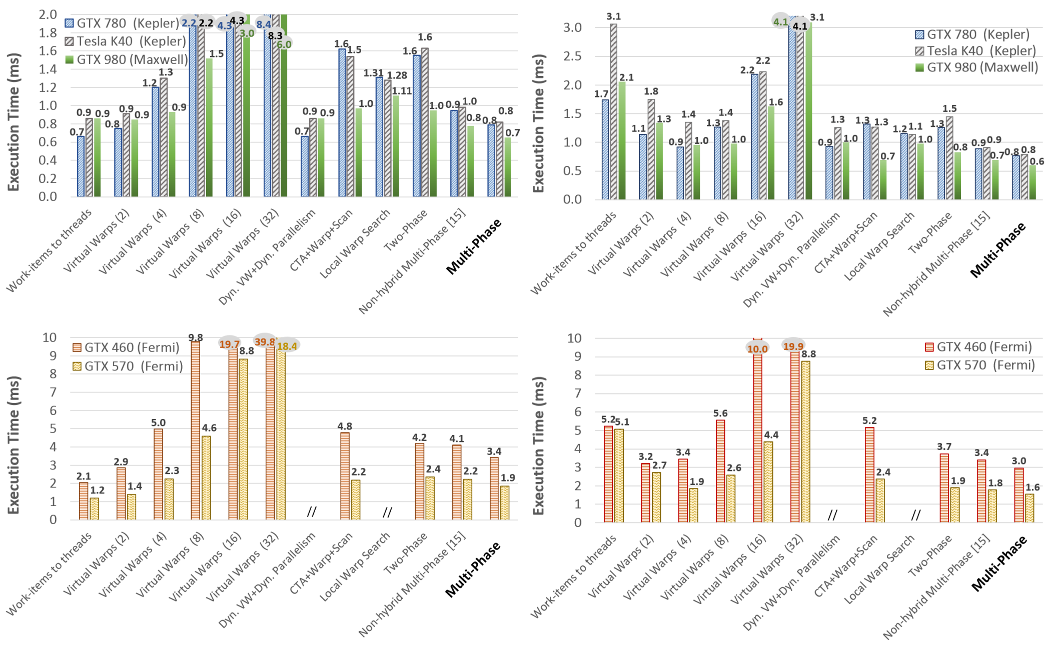

Column Index
Row Index
Full matrix representation
Row Index
R
C
V
Sparce matrix representation (CSR)

FIG. 5.5: *The CSR data structure.*

from very regular to strongly irregular workloads. Since sparse matrices are irregular by nature, we also included a synthetic benchmark (*regular8*) to understand how different algorithms behave in a very regular case.

In this problem formulation, the work-items correspond to the rows of the input matrix, while the number of work-units per work-item is the number of elements with nonzero values in the matrix columns for both symmetric and asymmetric matrix structures. The average work-item size and the standard deviation have been computed by considering the number of nonzero values independently from the matrix structure. We computed the prefix-sum of the number of work-units to generate the input data that is equivalent to the row offset array of the CSR sparse matrix format [37, 153, 195, 230]. CSR is one of most important and widely used sparse-matrix format. It allows storing nonzero elements (nnz) of a $m \times n$ matrix by using three arrays. Fig. 5.5 shows an example of a full matrix representation and the corresponding CSR data structure. The C array of size $|nnz|$ is a concatenation row-by-row of the nnz column indices. The R array consists of $m+1$ elements that point at where each row element list begins and ends within the array of the column indices. The V array holds the corresponding nnz values of the matrix. Since the specific values of the matrix are not relevant for the load balancing problem, we discard the V array and map the work-items and work-units respectively to the R and C arrays.

The *great-britain_osm* benchmark represents a road network with very uniform distribution and low average. *Cit-patent* represents the U.S. patent dataset, which has moderate average and not-uniform distribution. *web-NotreDame* is a web-crawl with a slightly higher average and middle-sized standard deviation. *Circuit5M* represents a circuit simulation instance, which shows a very high standard deviation. *As-skitter* is an autonomous system, while *kron_g500-logn20* is a synthetic graph based on the *Kronecker* model. The last two benchmarks are characterized both by highly not-uniform distribution, while they have low and high average, respectively.



(a) *great-britain_osm* (b) *cit-Patent*

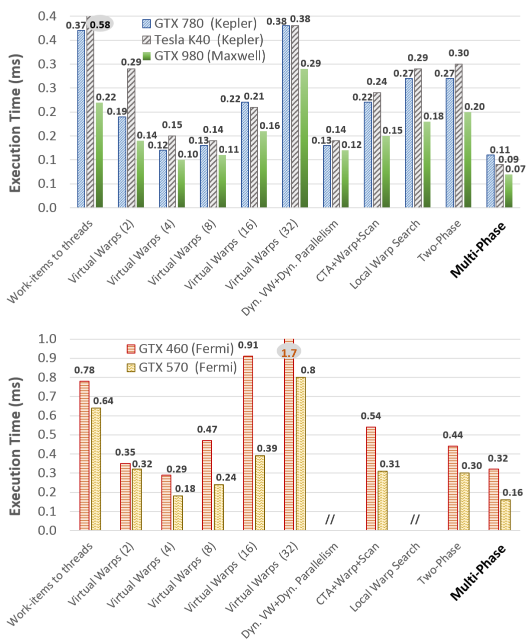
FIG. 5.6: *Comparison of execution time on the benchmarks.*

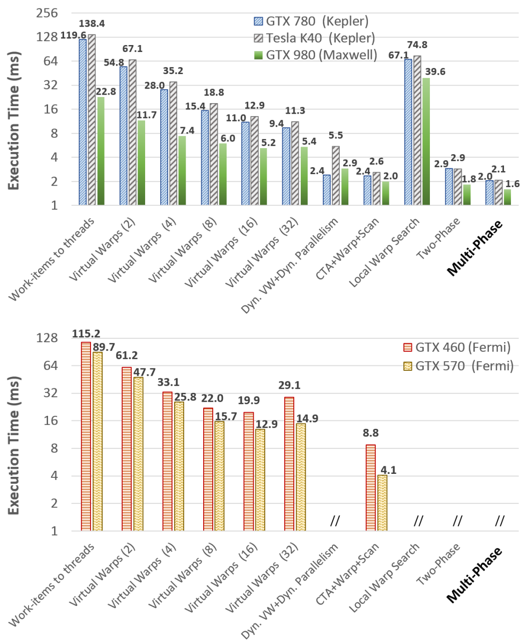
GPU model	N. of SMs	N. of cores	Mem. Bandwitdh	DRAM Memory	Shared Memory
GeForce GTX 780 (Kepler)	12	2304	288 GB/s	3 GB	48 KB
Tesla K40 (Kepler)	15	2880	288 GB/s	12 GB	48 KB
GeForce GTX 980 (Maxwell)	16	2048	224 GB/s	4 GB	96 KB
GeForce GTX 460 (Fermi)	7	480	115 GB/s	1 GB	48 KB
GeForce GTX 570 (Fermi)	15	336	152 GB/s	1.2 GB	48 KB

TABLE 5.4: GPU Characteristics

All the techniques have been integrated in a corresponding basic application, in which the threads access and update, in parallel, each work-unit of the benchmark workload. We ran the experiments on five GPU devices with three different micro-architectures (Fermi, Kerpler, and Maxwell). We included desktop-oriented devices (i.e., GeForce graphics cards) and a HPC-oriented device (Tesla K40). Table 5.4 summarizes their characteristics in terms of number of streaming multiprocessors (SMs), number of cores (stream processors), DRAM memory bandwidth, available DRAM memory, and shared memory.

Figures 5.6, 5.7, and 5.8 report the execution times required by the reference application (implemented with each of the analysed techniques) on the benchmarks. The benchmarks are orderly presented from the most regular to the most





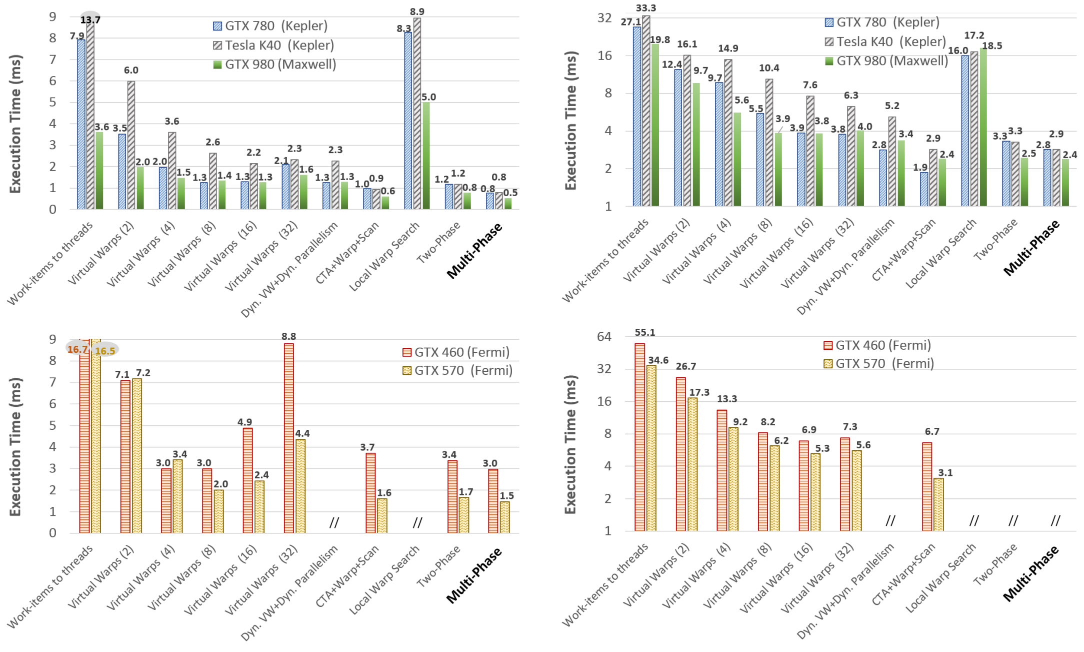
(a) *cit-Patent* (b) *cit-Patent*

FIG. 5.7: *Comparison of execution time on the benchmarks.*

irregular. The reported values represent the best performance obtained by tuning the kernel configuration in terms of number of threads per block. For the GPU devices used in this analysis, we obtained the best results with 128-256 threads per block for all the techniques. As confirmed by the profiler, such a configuration led to the maximum device occupancy and lowest synchronization overhead.

The results obtained with the *Direct Search* and *Block Search* techniques are far worse than the other techniques and, for the sake of clarity, have not been reported in the figures. For the *Two-Phase Search* algorithm, we used the well-know *ModernGPU* library [30] developed by NVIDIA Research, which is based on the merge path algorithm proposed by Green et al. [214]. All the other techniques have been implemented by accurately following the algorithm and optimization details presented in the corresponding papers. *Dynamic Virtual Warp* and *Local Warp Search* use advanced device features such as dynamic parallelism and registers shuffle among warp threads that are not supported by Fermi architectures (GTX 460 and GTX 570).

In the first benchmark (Fig. 5.6a), as expected, the static techniques are the most efficient. This is due to the very regular workload and to the low average work-item size. The semi-dynamic *Dyn. VW + Dyn. Parallelism* performs well since the dynamic parallelism feature is always switched off in such a regular workload. The static *Virtual Warps* approach provides good performance as long as the virtual warp size is properly set, while it sensibly worsens with wrongly-sized warps. In this benchmark, any overhead for the dynamic item-to-thread



(a) *cit-Patent* (b) *cit-Patent*

FIG. 5.8: *Comparison of execution time on the benchmarks.*

mapping may compromise the overall algorithm performance (see for instance *Local warp search* and *Two-phase Search*). However, the proposed *Multi-Phase Mapping* is among the most efficient technique for Kepler and Fermi architectures. The efficiency is comparable with the best static approaches with the GTX 780, while it is the most efficient technique with Tesla K40 and GTX 980. This underlines the reduced amount of overhead introduced by such a dynamic technique, which well applies also in case of very regular workloads.

The second benchmark (Fig. 5.6b) presents slightly higher average and standard deviation. *Multi-phase Mapping* shows the best results with all devices, while the best static techniques perform similarly to the semi-dynamic and dynamic ones. Beside strongly depending on the virtual warp sizing, the performance of the static techniques are very sensitive to the GPU device characteristics. Their performance strongly worsen (three times for the *Work-items to threads*, and almost twice for the best sized *Virtual Warps*) even on different devices of the same Kepler micro-architecture.

In *web-NotreDame* (Fig. 5.7a), *Multi-phase Mapping* is the most efficient technique and provides almost twice the performance with respect to the second best technique (*Virtual Warps*). It is three times faster than the other dynamic mapping techniques (*Local Warp Search* and *Two-Phase*) on all the GPU devices. Also with this benchmark, the virtual warp sizing strongly affects the *Virtual Warps* performance. We noticed that the optimal virtual warp size is proportional to the average of work-item sizes.

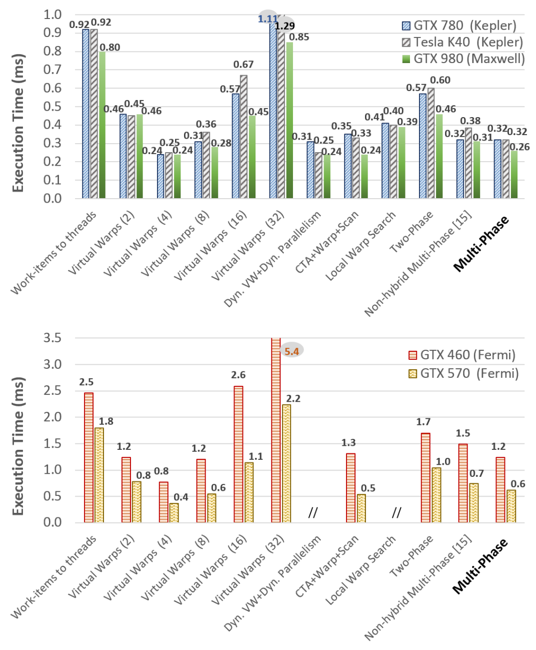


FIG. 5.9: *Comparison of execution time on the regular8 benchmark.*

In these first three benchmarks, *CTA+Warp+Scan*, which is one of the most advanced and sophisticated balancing technique at the state-of-the-art, provides low performance. This is due to the fact that the CTA and the Warp phases are never or rarely activated, while the activation controls involve much overhead.

Multi-phase Mapping provides the best results also in the *circuit5M* benchmark (Fig. 5.7b). In such a benchmark, we observed that the *CTA+Warp+Scan*, *Two-Phase Search*, and *Multi-phase Mapping* dynamic techniques are one order of magnitude faster than the static ones.

In *web-Notredame* and in *circuit5M*, *Multi-phase Mapping* shows the best results due to the low average (less than warp size) and high standard deviation.

In the last irregular benchmarks, *as-skitter* (Fig. 5.8a) and *kron_g500-logn20* (Fig. 5.8b), *Multi-phase Mapping* and *CTA+Warp+Scan* provide the best results. In the most irregular benchmark (*kron_g500-logn20*) *CTA+Warp+Scan* has slightly better performance than *Multi-phase Mapping*, particularly on the GTX 780 device, since the CTA and Warp phases are frequently activated and exploited. Since Kepler devices are throughput-oriented architectures (higher memory bandwidth) while Maxwell devices are more focused on power consumption, *CTA+Warp+Scan* provides better performance on GTX 780 than GTX 980 device by exploiting the higher memory bandwidth of the former.

Dynamic Virtual Warps and *Virtual Warps* provide similar performance. They are very efficient on benchmarks with high-average work-item sizes since, with a thread group size of 32, they completely avoid warp divergence.

In the regular benchmark, *regular8* (Fig. 5.9), the most efficient technique is the *Virtual Warps* for all the considered GPU devices as expected. The perfectly uniform workload benefits from the static techniques in which the size of the thread group is properly set according to the benchmark characteristics. While the work-unit average equal to eight should provide higher performance for 8-thread groups, 4-threads virtual warps show lower execution time thanks to a smaller block scheduling overhead. *CTA+Warp+Scan* and *Multi-Phase Mapping* provide slightly lower performance since they involve extra work to organize the computation.

Finally, we observed that the *Dynamic Parallelism* feature, implemented in the corresponding semi-dynamic technique, finds the best application with the GTX 780 device and only when the work-item sizes and their average are very large. In any case, all the dynamic load balancing techniques, and in particular *Multi-phase Mapping*, perform better without such a feature in all the analysed datasets. GPU devices with limited DRAM memory (GTX 460 and GTX 570) do not support *Circuit5M* and *kron_g500-logn20* with *Two-Phase Search* and *Multi-phase Mapping* as they need additional space to store the intermediate partitioning results. In general we observed that all the techniques provide performance two/three times better on recent architectures (i.e. Kepler and Maxwell) than on the previous GPU generation (Fermi).

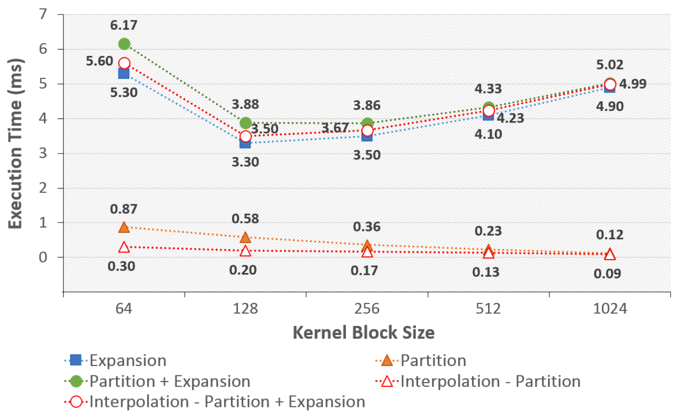


FIG. 5.10: *Execution time of Partition and Expansion phases by varying the block size (on 2^{26} items with unif. distributed random work-sizes).*

5.4.1 *Multi-phase Mapping* Analisys

Fig. 5.10 shows the impact of the thread block size on the performance of the main phases of *Multi-phase Mapping*. The *partition* phase performance improves linearly to the block size. This is due to the fact that large blocks involve the input workload to be partitioned in fewer work-unit chunks and, as a consequence, they require fewer threads for such a computation. The computation is completely independent among threads. In contrast, large block sizes penalize the performance of the *expansion* phase. This is due to the synchronization overhead required to

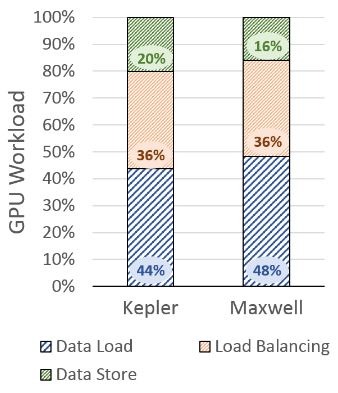


FIG. 5.11: *GPU workload breakdown of Multi-phase algorithm on Kepler and Maxwell Architectures*

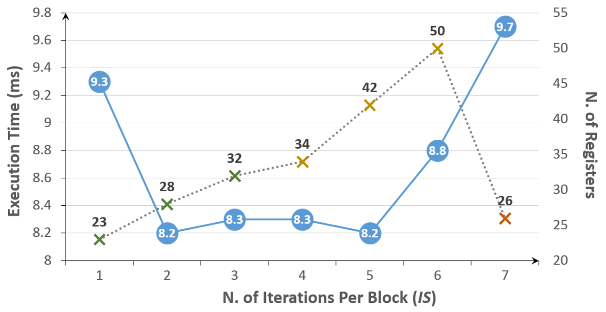


FIG. 5.12: *Execution time by varying the number of iterations (on 2^{26} work-items with uniformly distributed random work-item sizes).*

coordinate the shared memory accesses. We observed that the best trade-off size of blocks is 128 or 256 (see the *partition+expansion* line in Fig. 5.10).

Fig. 5.11 depicts the contribution of each of the three main steps of the expansion phase to the overall kernel execution time, for the Kepler and Maxwell architectures. For both architectures the main step of the load balancing (i.e., binary searches) takes more than one third of the whole execution. The time spent for the store activity is two/three times higher than the time spent for loading data, even though data storing involves much more memory accesses than data loading. This is due to the fact that the size of the data loaded into the block local memory is known only at run-time. This prevent us form applying any of the optimizations presented in Section 5.2.3, in particular, loop unrolling and vectorized memory accesses.

Fig. 5.12 reports the *Multi-phase Mapping* execution time obtained by varying the number of iterations (i.e., the *IS* value). *IS* affects the number of required registers and, as a consequence, the overall balancing performance. In the GPU devices used for these experiments, the maximum number of registers per thread is 32. As for the standard behaviour of GPU devices, exceeding such a threshold involves data to be *spilled* in L1 cache and then in L2 cache or global memory. With *IS* values from two to five, we obtained the best performance, as all the data elaborated by the threads mainly fits in registers and, in small part, in L1 cache. From seven iterations onwards, the performance drastically decreases since the compiler places the data variables outside the on-chip memory.

5.5 Conclusions

This Section presented an accurate analysis of the load balancing techniques based on prefix-scan in the literature, by underlining their advantages and drawbacks over different workload characteristics. The Section then presented an advanced dynamic technique, called *Multi-phase Mapping*, which addresses the workload unbalancing problem by fully exploiting the GPU device characteristics. The Section showed how *Multi-phase Mapping* implements a dynamic partitioning and mapping approach through an algorithm whose complexity is sensibly reduced with respect to the other dynamic approaches. This allows the proposed approach to provide good performance when applied both to very irregular and to regular and balanced workloads. The Section presented a comparison between the proposed solution and the existing approaches by considering different benchmarks as well as different GPU architectures in order to understand advantages and drawbacks of each technique also considering the underlying device characteristics.

6

Breadth-First Search - BFS-4K

Breadth-first search (BFS) is one of the most common graph traversal algorithms and the building block for a wide range of graph applications. With the advent of graphics processing units (GPUs), several works have been proposed to accelerate graph algorithms and, in particular, BFS on such many-core architectures. Nevertheless, BFS has proven to be an algorithm for which it is hard to obtain better performance from parallelization. Indeed, the proposed solutions take advantage of the massively parallelism of GPUs but they are often asymptotically less efficient than the fastest CPU implementations. This Section presents *BFS-4K*, a parallel implementation of BFS for GPUs that exploits the more advanced features of GPU-based platforms (i.e., NVIDIA Kepler) and that achieves an asymptotically optimal work complexity. The Section presents different strategies implemented in *BFS-4K* to deal with the potential workload imbalance and thread divergence caused by any actual graph non-homogeneity. The Section presents the experimental results conducted on several graphs of different size and characteristics to understand how the proposed techniques are applied and combined to obtain the best performance from the parallel BFS visits. Finally, an analysis of the most representative BFS implementations for GPUs at the state of the art and their comparison with *BFS-4K* are reported to underline the efficiency of the proposed solution.

6.1 Introduction

Graphs are a common representation in many problem domains, including engineering, finance, medicine, and scientific applications. Breadth-first search (BFS) is a crucial graph traversal algorithm used by many graph-processing applications. Different problems, such as VLSI chip layout, phylogeny reconstruction, data mining, and network analysis, map to very large graphs, often involving millions of vertices. Even though very efficient *sequential* implementations of BFS exist [71, 74, 159], they have work complexity of the order of number of vertices and edges. As a consequence, such sequential implementations become impractical when applied on very large graphs.

Recently, graphics processing units (GPUs) have become widespread platforms as they provide massive parallelism at low cost. Parallel executions on GPUs may achieve speedup up to three orders of magnitude with respect to the sequential counterparts on CPUs. Nevertheless, accelerating efficient and optimized sequential algorithms and porting (i.e., parallelizing) their implementation to such many-core architectures is a very challenging task. Several solutions in literature take advantage of the massive parallelism of GPUs [117, 125, 134, 249, 289] but they are often asymptotically less efficient than the fastest CPU implementations [74]. After a certain graph size and, thus, for graph sizes typical of many actual problem domains, the parallel implementations for GPUs become slower than the sequential implementations for CPUs.

Thread divergence, workload imbalance, and poorly coalesced memory accesses are the most representative issues that come up when traversing a graph with a parallel implementation. In particular, sparse graphs, scale free networks or graphs with power-law distribution in general show up the limits of the parallel implementations suffering from these problems [14, 156, 236, 280].

On the other hand, GPU vendors continue to innovate and meet that demand for high performance parallel computing with extremely powerful GPU computing architectures (e.g., NVIDIAs new Kepler GK110 [202]). The most recent architectures, not only offer much higher processing power than the prior GPU generations, but, also, they provide new programming capability to improve the efficiency of the parallel implementations.

This Section presents *BFS-4K*, a parallel implementation of BFS for GPUs, which exploits the more advanced features of GPU-based platforms (i.e., NVIDIA Kepler) to improve the execution speedup w.r.t. the sequential CPU implementations and to achieve an asymptotically optimal work complexity. The Section presents the different features implemented in *BFS-4K* to deal with the potential workload imbalance and thread divergence caused by the graph non-homogeneity (i.e., number of vertices, edges, diameter, and vertex degree variability). An analysis of every single technique is also presented to show how much they influence the overall performance and how they can be customized to exploit the architecture configurations for the graph characteristics.

Finally, the performance of the proposed implementation (which is available for download in $http://profs.sci.univr.it/\sim bombieri/BFS-4K/index.html$) is compared with the most efficient BFS implementations for GPUs at the state of the art over several graphs of different sizes and characteristics.

The work is organized as follows. Section 6.2 gives an overview of the proposed approach while the single techniques implemented in *BFS-4K* are detailed in Section 6.3. Section 6.4 presents the problem of duplicates and the proposed approach to deal with them. Finally, Section 6.5 presents the experimental results by underlying the single technique contributions in the overall visit performance and a comparison of *BFS-4K* with the BFS implementations for GPU at the state of the art. Section 6.6 is devoted to concluding remarks.

This section presents some preliminary concepts concerning CUDA architectures and BFS, which facilitate the reader to better understand the proposed solution.

6.1.1 Breadth First Search (BFS)

BFS is one of the most import graph algorithms. It is used in several different contexts such as image processing, state space searching, network analysis, graph partitioning, and automatic theorem proving. Given a graph $G(V, E)$, where V is the set of vertices and E is the set of edges, and a source vertex s, the BFS visit inspects every edge of E to find the minimum number of edges or the shortest path to reach every vertex of V from source s. The traditional sequential algorithm [74] can be summarized as follows:

```
for all verticies $u \in V(G)$ do
    $u.dist \leftarrow \infty$
    $u.\pi \leftarrow -1$
end
$v_0.dist \leftarrow 0$
$v_0.\pi \leftarrow v_0$
$Q \leftarrow \{v_0\}$
while $Q \neq \emptyset$ do
    $u \leftarrow$ DEQUEUE(Q)
    for all verticies $v \in adj\ [u]$ do
        if $v.dist = \infty$ then
            $v.dist \leftarrow u.dist + 1$
            $v.\pi \leftarrow u$
            ENQUEUE$(Q, v)$
    end
end
```

where Q is a FIFO queue data structure that stores not yet visited vertices, $v.dist$ represents the distance of vertex v from the source vertex s (number of edges in the path) , and $v.\pi$ represents the parent vertex of v. An unvisited vertex v is denoted with $v.dist$ equal to ∞. The asymptotic time complexity of the sequential algorithm is $O(V + E)$.

6.2 BFS-4K Overview

Given a graph $G(V, E)$ and a source vertex s, *BFS-4K* exploits the concept of *frontier* [74] to achieve work efficiency $O(V + E)$ for the parallel BFS visits of G. The tool generates a breadth-first tree that has root s and contains all reachable vertices. The vertices in each *level* of the tree compose a *frontier (F). Frontier propagation* checks every neighbor of a frontier vertex to see whether it is visited already. If not, the neighbour is added into a new frontier.

BFS-4K implements the frontier propagation through two data structures, Fd and Fd_{new}. Fd represents the actual frontier, which is read by the parallel threads to start the propagation step. Fd_{new} is written by the threads to generate the frontier for the next BFS step. At each step, Fd_{new} is filtered and swapped into

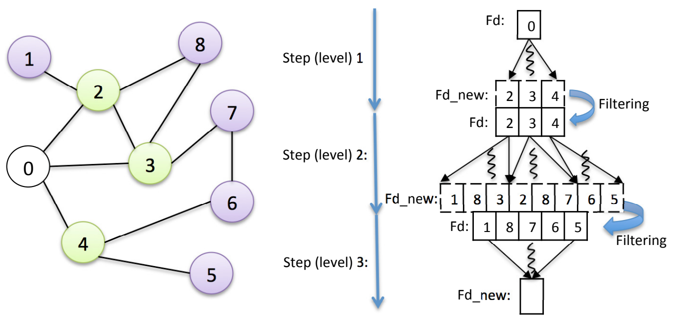


FIG. 6.1: *Example of BFS visit starting from vertex* "0"

Fd for the next iteration. Figure 6.1 shows an example, in which starting from vertex "0", the BFS visit concludes in three steps[1].

The filtering steps aims at guaranteeing correctness of the BFS visit as well as avoiding useless thread work and waste of resources. When a thread visits a neighbor already visited, that neighbor is eliminated from the frontier (e.g., vertex 2 visited by a thread from vertex 3 in step two of Figure 6.1). When more threads visit the same neighbor in the same propagation step (e.g., vertex 8 visited by threads 2 and 3 in step two), they generate *duplicate* vertices in the frontier. Duplicate vertices cause redundant work in the subsequent propagation steps (i.e., more threads visit the same path) and useless occupancy of shared memory. *BFS-4K* implements a duplicate detection and correction strategy based on hash tables, Kepler 8-byte memory access mode, and warp shuffle instructions, as explained in Section 6.4.

The considered graphs may have significant variability in terms of number of vertices, edges, diameter, and vertex degree, which may imply several issues to a parallel BFS visit. To handle the potential workload imbalance and thread divergence caused by such a graph non-homogeneity, *BFS-4K* implements the following features:

- *Exclusive prefix-Sum.* To improve data access time and thread concurrency during the propagation steps, the frontier data structures are stored in shared memory and handled by a *prefix-sum* procedure. Such a procedure is implemented through warp shuffle instructions of the Kepler architecture, as explained in Section 6.3.1.
- *Dynamic virtual warps.* The *virtual warp* technique presented in [125] is applied to minimize the waste of GPU resources and to reduce the divergence during the neighbour inspection phase. Differently from [125], this work proposes a strategy to dynamically calibrate the warp size at each frontier propagation step, as explained in Section 6.3.2.

[1] For the sake of clarity, the figure shows Fd_{new} firstly written and then filtered. As explained in the following sections, to reduce the global memory accesses, the next frontier is firstly filtered and, then, Fd_{new} is written. The Fd and Fd_{new} data structures have the same size in memory.

- *Dynamic parallelism.* In case of vertices with degree much greater than the average, (e.g., scale free networks or graphs with power-law distribution in general), *BFS-4K* applies the dynamic parallelism provided by the Kepler architecture instead of virtual warps. Dynamic parallelism implies an overhead that, if not properly used, may worse the algorithm performance. *BFS-4K* checks, at run time, the characteristics of the frontier to decide whether and how applying this technique, as explained in Section 6.3.3.
- *Edge-Discover.* With the edge-discover technique, threads are assigned to edges rather than vertices to improve the thread workload balancing during frontier propagation. The edge-discover technique makes intense use of warp shuffle instructions. *BFS-4K* checks, at each propagation step, the frontier configuration to apply this technique rather than dynamic virtual warps, as explained in details in Section 6.3.4.
- *Single-block vs. Multi-block kernel. BFS-4K* relies on a two-kernel implementation. The two kernels are alternately used and combined with the features presented above during frontier propagation. Section 6.3.5 presents an analysis of the two-kernel features and explains how they are applied to better exploit the GPU stream multiprocessor properties.
- *Coalesced read/write memory accesses.* To reduce the overhead caused by the many accesses in global memory, *BFS-4K* implements a technique to induce coalescence among warp threads through warp shuffle, as explained in Section 6.3.6..

The Section presents an analysis of the advantages and limits of each proposed technique to understand how and when they can be applied and combined to improve the performance of the BFS visits. As explained in the following sections, the techniques can also be calibrated through several *knobs* to customize *BFS-4K* depending on both the GPU device characteristics and the graphs to be visited.

6.3 Implementation Features in Details

This section deepens the *BFS-4K* implementation features and presents an analysis and some examples of each feature contribution to the overall visit performance.

6.3.1 Exclusive Prefix-Sum

Given a list of input values and a binary associative operator, a *prefix-scan* procedure computes an output list of elements in which each element is the reduction of the elements occurring earlier in the input list. Prefix-scan has been largely investigated in the past years and several solutions have been presented for both array processor architectures [45, 65] and GPUs [40, 93, 186, 240].

When the operator is the addition, the prefix-scan represents a *prefix-sum.* Prefix-sum is useful when parallel threads must allocate dynamic data within shared data structures such as global queues. Given a total amount of data to be allocate for each thread, prefix-sum calculates the offsets to be used by the threads to start writing the output elements [107].

```
exclusiveWarpPrefixSum
    for (i = 1; i $\leqslant$ 16; $i = i * 2$) do
        n = __shfl_up($v$, i, 32)
        if $lane_{id} \geqslant$ i then
            $v$ += n
    end
    __shfl_up($v$, 1, 32)
    if $lane_{id} = 0$ then
        $v = 0$
```

FIG. 6.2: *Overview of a prefix-sum procedure implemented with shuffle instructions*

BFS-4K exploits prefix-sum procedures to manage the frontier queues as well as the edge-discover visit (see Section 6.3.4). During frontier propagation, the prefix-sum is used to compute the scatter offset needed by each thread to assemble, in parallel, global edge frontiers from expanded neighbours and when producing unique unvisited vertices into global vertex frontiers. Since the first offset must be zero, the prefix-sum results are shifted to right of one position to implement the *exclusive* variant.

BFS-4K implements a two-level exclusive prefix-sum, that is, at warp-level and block-level. The first is implemented by using Kepler warp-shuffle instructions, which guarantee the result computation in $\log n$ steps rather than $2 \log n$ as in the most efficient implementations in literature that rely on shared memory (e.g., [107]). Figure 6.2 shows a high-level representation of such a prefix-sum procedure implemented with a warp shuffle instruction (i.e., *__shfl_up*()).

Each frontier assembling step requires also synchronization among thread blocks, which eventually write the final frontier into the global memory. These last steps are performed through the block-level exclusive prefix-sum, which is implemented through atomic operations and relies on shared memory. However, the warp-level prefix-sum computes the majority amount of work of the frontier assembling steps, and its efficient implementation trough shuffle instructions sensibly impacts on the overall BFS visit.

Finally, at each frontier propagation step, *BFS-4K* checks whether every frontier vertices have at most one neighbor (i.e., scatter offset either 0 or 1 for each thread). The check, which work complexity is $O(1)$, aims at running, when possible, a more efficient *binary* variant of the exclusive prefix-sum [120], which has been implemented with the intrinsics instructions __BALLOT and __POPC.

6.3.2 Dynamic Virtual Warps

The concept of *virtual warp* has been presented in [125] to address the problem of workload imbalance in GPU programming. The idea is to allocate a chunk of tasks to each warp and to execute different tasks as serial rather than assigning a different task to each thread. Multiple threads are used in a warp for explicit SIMD operations only, thus preventing branch-divergence altogether.

The speedup provided by virtual warps is strictly related to the virtual warp size. As shown in the experimental results [125], a wrong size setting could also lead to a speedup decrease.

In the BFS context, the virtual warps technique can be applied to increase the thread coalescence during the accesses to the adjacent lists and to reduce their divergence in the frontier propagation steps. The main limitation of such a technique in BFS occurs when the virtual warp size does not properly fit the vertex degree, thus leading to unused threads. In case of vertices with very different degree over the propagation steps, the size choice may not be always appropriated. Thus, differently from [125], *BFS-4K* implements a *dynamic* virtual warp, whereby the warp size is calibrated at each frontier propagation step i, as follows:

$$WarpSize_i = nearest\_pow2\left(\frac{\#Res\text{Threads}}{|F_i|}\right) \in [K_1, 32]$$

where *#ResThreads* refers to the maximum number of resident threads in case of multi-block kernel while thread block size in case of single-block kernel (see Section 6.3.5). $nearest\_pow2$ is the lower nearest power of two that rounds the division, while $|F_i|$ is the size of the actual frontier.

Even though the warp size may range between 1 and 32, *BFS-4K* is parametrized to set the minimum warp size (K_1). Too small sizes of virtual warps may lead to poor coalescence and thread divergence depending on the graph characteristics. As explained in the experimental results, we heuristically set $K_1 = 4$ for all the analysed graphs.

The choice of the warp size also directly affects the problem of duplicate vertices. A small size, which leads to finer granularity of warp work and fine grained synchronization, involves less duplicate vertices during frontier propagation. In contrast, large sizes of warps may reduce the synchronization overhead but they lead to more duplicates, thus requiring more resources for the duplicate detection and correction, as explained in Section 6.4.

6.3.3 Dynamic Parallelism

The exclusive prefix-sum and dynamic virtual warp strategies guarantee a fair workload balancing during the BFS visit of irregular graphs. Nevertheless, they found their main limitation in several categories of graphs, e.g., scale free networks or graphs with power-law distribution in general. In these cases, the visit of very few vertices with very high degree can compromise the performance of the entire BFS visit.

To overcome this limitation, *BFS-4K* exploits the dynamic parallelism feature of the Kepler architectures. Dynamic parallelism allows recursion to be implemented in the kernels and, thus, threads and thread blocks to be dynamically created at run time without requiring kernel returns. In the BFS context, the idea is to invoke a multi-block kernel (which we call *child kernel*) properly configured to manage the workload imbalance due to the difference of the vertex degrees. Nevertheless, even if low, the overhead introduced by the dynamic kernel stack may elude this feature advantages when replicated for all frontier vertices unconditionally.

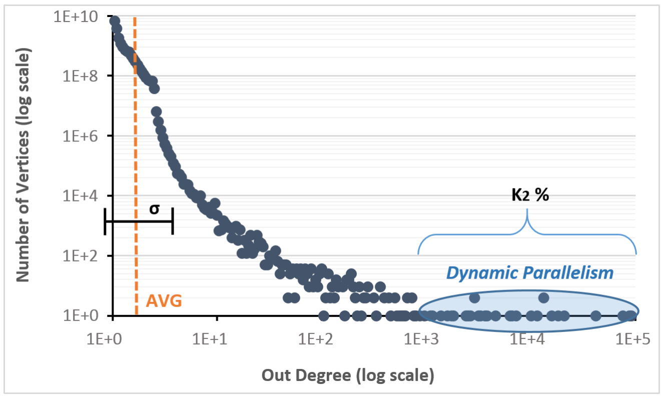


FIG. 6.3: *Example of dynamic parallelism applied to a sub-set of frontier vertices of a power-law graph*

BFS-4K applies dynamic parallelism to a limited number of frontier vertices at each frontier propagation step. Given the degree distribution of the visited graph, *BFS-4K* applies dynamic parallelism to the sub-set of vertices ($K_2\%$) having degree far from the average (AVG), starting from those with highest degree (Figure 6.3 shows an example).

In particular, *BFS-4K* combines dynamic parallelism with dynamic virtual warps. The threshold K_2 is a further knob to be set in *BFS-4K*, which switches the use of the former technique rather than latter. As explained in the experimental results, we heuristically fixed $K_2 = 0.15\%$ (% of the total number of vertices V) for all the analysed graphs.

The threshold is correlated with the virtual warp size and, in particular, with K_1. The smaller K_1, the larger K_2. That is, the larger the minimum warp size, the smaller the sub-set of vertices that can be managed by dynamic kernels to improve the BFS performance. This is due to the fact that large virtual warps can handle the workload imbalance more efficiently (i.e, with less overhead) than dynamic parallelism.

In *BFS-4K*, the child kernels are configured to ensure the minimum overhead of the child thread synchronization, and the best balancing among parent and child threads. Figure 6.4 shows an example of three different kernel settings in terms of number of blocks (with a fixed block size), given a parent kernel (leftmost side of figure) and a child kernel (lower side of the figure). Case *(a)* represents an over-sized kernel, in which the blocks are more than the vertex neighbor and, thus, they conclude shorter than the other threads of the parent. Nevertheless, the many child blocks involve many atomic operations to update the frontier data structures and an underutilization of the fast local queues. In contrast, case (c) represents an undersized kernel, in which less blocks manage many vertex neighbours. Even though it involves less atomic operations, this configuration leads to imbalance with regards to the parent threads, since the parent kernel must wait for all the threads (including child threads) to end before carrying on with the next propagation step (see thread synchronization in Kepler dynamic parallelism [202]).

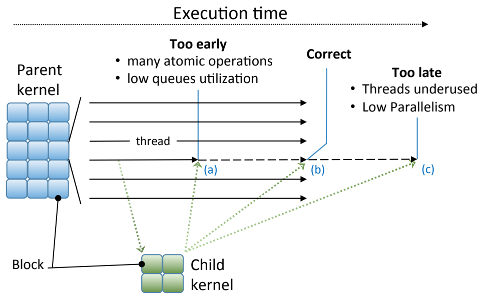


FIG. 6.4: *Block number setting: (a) oversized kernel, (b) correctly sized kernel, (c) undersized kernel*

Case (b) represent the trade-off solution implemented in *BFS-4K*, in which the child kernel returns at the same time or close to the parent kernel. The child kernel is configured as follows:

- $\#Blocks = \dfrac{VertexDegree}{K_3 \times ThreadBlockSize}$
- $BlockSize$ = block size of the parent kernel to fully exploit the resident threads on the streaming multiprocessors.

where $VertexDegree$ is the degree of the frontier vertex for which the thread dynamically calls a child kernel.

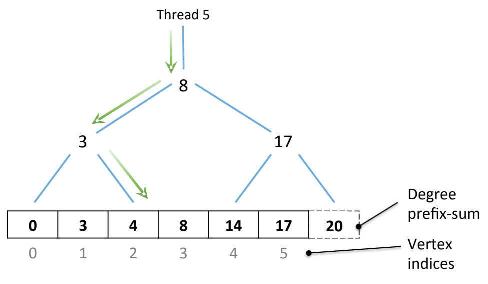
In the experimental results, $K_3 = 16$ (i.e, each thread of the child kernels sequentially manages a queue of 16 vertex neighbour) provides the best BFS performance for the analysed graphs.

6.3.4 Edge-discover

In the edge-discover technique, the idea is to assign threads to edges rather than to vertices during frontier propagation to better balance the thread workload. The main problem is the cost of such a thread partitioning and assignment, which may elude the advantages of the technique itself.

BFS-4K implements thread assignment through a binary search and by making intense use of warp shuffle instructions. Given a thread warp, and the actual frontier:

1) Each warp thread reads a frontier vertex, saves the degree and the offset of the first edge.
2) Each warp computes the warp shuffle prefix-sum on the vertices degree.
3) Each thread of the warp performs a warp shuffle binary search of the own warp id (i.e., $lane_{id} \in \{0, .., 31\}$) on the prefix-sum results. Figure 6.5 shows an example, in which 20 threads of a warp are assigned to 6 vertices of a frontier. In the example, thread 5 is assigned to vertex 2 of the frontier after two binary search steps. The warp shuffle instructions guarantee the efficiency of the search steps (which are less than $log_2(WarpSize)$ per warp).



(a)

Thread ID (From – To)	Assigned Vertex
0 – 2	0
3	1
4 – 7	2
8 – 13	3
14 – 16	4
17 – 19	5

(b)

FIG. 6.5: *Example of partitioning and assignment of warp threads in the edge-discover technique: (a) assignment of thread with lane$_{id}$ = 5 to vertex 2 of the frontier, (b) final assignment table of 20 warp threads to 6 frontier vertices*

4) The threads of warp share, at the same time, the offset of the first edge with an other warp shuffle operation.
5) Finally, the threads inspect the edges and store possible new vertices on the local queue.

With this procedure, the workload is always balanced, the local queues are filled equally and the duplicates are considerably reduced since the parallel visit is for edges (see Section 6.4). The local queue management and the global memory accessing and synchronization are similar to those implemented in the dynamic virtual warp strategy.

Finally, *BFS-4K* implements an extended edge-discover technique (EXT) to optimize the visit of middle size degree vertices. When the last thread of a warp finds a vertex with a degree greater than the warp size, it shares the offset with a shuffle operation and directly assigns threads without performing a new iteration of binary search. As shown in Section 6.5, this optimization provides a sensible speedup improvement in the BFS visit of several graphs.

BFS-4K applies the edge-discover technique as an alternative of dynamic virtual warps to be combined with dynamic parallelism. With this new combination, the threshold of dynamic parallelism (K_2) can be increased more than in the former combination. This is due to the fact that the warp parallelism on edges allows high-degree vertices to be handled more efficiently than the warp parallelism on vertices. Nevertheless, the overhead introduced by the thread assignment limits the edge-discover application.

In general, the edge-discover is more efficient than dynamic virtual warps if the frontier vertices are less than the available (resident) threads. *BFS-4K* checks the following condition at each frontier propagation step:

$$|F_d| < \frac{\#Res\text{Threads}}{K_4}$$

where K_4 is a further knob that allows the switch between one technique over the other to be calibrated depending on the graphs characteristics. In the experimental results, we found $K_4 \in \{1, 2, 4\}$ as the best configuration for the analysed graphs.

6.3.5 Single-block vs. Multi-block Kernel

In a parallel BFS visit based on frontier propagation, the frontier size follows a trend as that shown in Figure 6.6. In the first and last steps of the overall frontier propagation the available parallelism is particularly limited. As a consequence, in these propagation periods, it is more convenient to handle the frontier vertices with a single block of threads. This allows the whole frontier to be maintained in shared memory and the block threads to exploit the efficient synchronization and communication mechanisms.

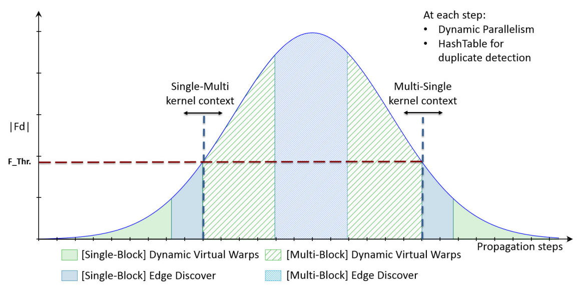


FIG. 6.6: *Single and multi-block kernel use during frontier propagation steps*

BFS-4K implements two different kernels (i.e., single-block and multi-block kernels) that are combined with the edge-discover and the dynamic virtual warp techniques presented in the previous sections.

A threshold (*F_Threshold*) is statically calibrated depending on the graph characteristics. At each propagation step, *BFS-4K* runs the single-block or the multi-block kernel if the current frontier size is smaller or larger, respectively, than the threshold. In general, the single-block kernel is run in the first and last propagation steps, while the multi-block kernel is run in the middle steps, as shown in Figure 6.6.

The threshold calibration impacts on the organization of the shared memory of each streaming multiprocessor. Figure 6.7 depicts the shared memory organization in case of single or multi-block kernel. In the first case, the shared memory stores the frontier data structures (*Fd* and *Fd_new*), kernel variables, and the hash

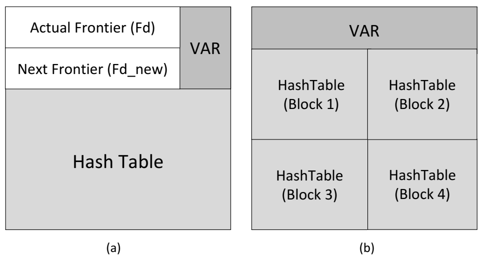


FIG. 6.7: *Shared memory organization: (a) single-block kernel, (b) multi-block kernel*

table for implementing duplicate detection and correction. The memory is sized as follows:

$$F\_Threshold = MaxThreadsPerBlock \cdot \mathrm{K}_5;$$

$$Fd_{size} \geqslant \frac{F\_Threshold \cdot 4}{2};$$

$$HashT_{size} = nearest\_pow2\left(|SM| - (FdSize \cdot 2) - Var_{size}\right).$$

where $MaxThreadsPerBlock$ is the maximum size of the single block (which must satisfy the GPU device constraints), and K_5 is a further knob to assign more frontier vertices per thread. $|SM|$ is the total size of the shared memory. For efficiency reason, the hash table partition must be a power of two. Considering, for example, a 48K shared memory, the hash table size can be set to 32K, 16K, 8K or less.

In case of multi-block kernel, the shared memory is organized as depicted in Figure 6.6(b). In this case, a hash table instance is dedicated to each thread block, and the tables are sized as follows:

$$HashT_{size} = nearest\_pow2\left(\frac{(|SM| - Var_{size}) \cdot K_6}{MaxThrPerMultiproc}\right).$$

where K_6 is the knob to size blocks (in terms of number of threads) and $MaxThrPerMultiproc$ is the maximum number of threads per multiprocessor (i.e., GPU device constraint).

K_5 impacts on the threshold and it aims at shifting the single-multi kernel switch points. This knob can be properly set to avoid both a premature switch to the multi kernel (with a consequent underutilisation of the multi-block threads and more overhead due the CPU synchronization) and a late switch whereby the single kernel serializes the visit of the many frontier vertices. In the single kernel context, K_5 allows the user to partition the shared memory between frontier data structures and hash table depending on the graph characteristics, in particular frontier size distribution and number of duplicates.

6.3.6 Coalesced Read/Write Memory Accesses

In the Kepler architectures, the maximum coalescence in memory accesses can be achieved by four threads belonging to the same half warp. In these cases, the memory access is performed by 128-bit transactions (32 bits per thread).

With *virtual* warps, the maximum coalescence is inversely proportional to the warp size. For example, given a 32 thread warp and 4 virtual warps (each one of 8 threads), the maximum coalescence can be achieved by two virtual warp threads belonging to the same half warp. In this case, the memory access is performed by 64-bit transactions. The worst case occurs when the virtual warps are sized 32, in which the accesses cannot be coalesced.

To deal with such a problem involved by virtual warps, *BFS-4K* takes advantage of warp shuffle instructions to share the read data among the virtual warp threads. To elude the overhead involved by the warp shuffle operations, such a reading technique is applied under two constraints:

1. $|Fd| > ResThreads$, that is, only if all the virtual warp threads are involved in the frontier propagation;
2. $WarpSize_i = 32$, that is, only in propagation steps in which there would not be coalescence in memory reading.

The coalescence problem for memory *reads* is suffered from the virtual warp technique only. In contrast, coalescence for memory writes is suffered from all the techniques in general (i.e., virtual warps, dynamic parallelism, and edge discover). At each propagation step, the threads exploit *local queues*, which are data structures in thread registers, to store and filter the neighbour vertices. After the filtering phase, each thread updates the own frontier segment in the global memory (Fd_{new}). In the classic context, the Fd_{new} updating is performed in parallel, where each thread sequentially writes the own vertices starting from the scatter offset calculated by prefix-sum (see Section 6.3.1). This leads to coalescence problems since the memory accesses rely on the number of vertices to be written in global memory.

BFS-4K implements a technique to induce coalescence in memory writes as follows (see Figure 6.8):

1. The shortest size of the queues (which we call *minimum*) is calculated through warp-shuffle instructions in *log* time.
2. Each thread updates Fd_{new} by writing the vertices stored in the local queues at the same position (e.g., the first thread writes the four blue vertices in global memory, the second thread writes the green four vertices, etc.). Each write is coalesced and the scatter offset is equal to the number of local queues. The minimum value represents the total number of coalesced writes and the starting point for the remaining writes with the prefix-sum technique.

The overhead involved by the *minimum* value calculation is not negligible, especially for large sized virtual warps. Thus, a further knob, K_7, allows the user to set a threshold for switching the writing mode between induced coalescence and standard non coalesced (prefix-sum). The K_7 value depends on the GPU characteristics (warp shuffle efficiency). In the experimental results, we heuristically set $K_7 = 10$.

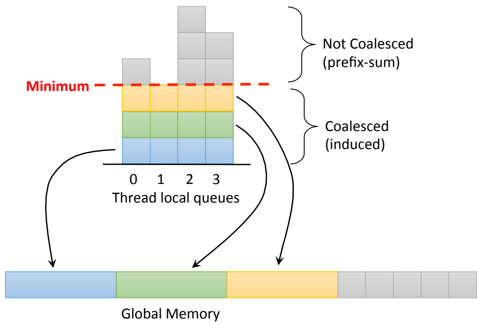


FIG. 6.8: *Example of induced coalescence in memory accesses*

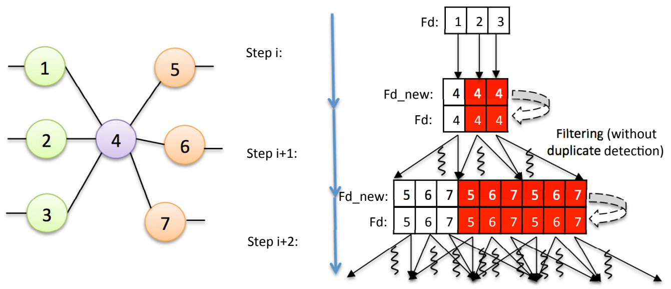


FIG. 6.9: *Example of duplicates exponential growth*

6.4 Duplicate Detection and Correction

Duplicate vertices are a relevant problem in the parallel BFS visit of graphs. Duplicate vertices are generated whenever two or more threads visit the same vertex at the same time and, as a consequence, they cause redundant work among threads during frontier propagation. Figure 6.9 shows an example that underlines how such a redundant work grows exponentially through the frontier propagation steps.

BFS-4K implements a hash table in shared memory (i.e., one per streaming multiprocessor) to detect and correct duplicates, and takes advantage of the 8-bank shared memory mode of Kepler to guarantee high performance of the table accesses. At each propagation step, each frontier thread invokes the `hash64` procedure depicted in Figure 6.10 to update the hash table with the visited vertex (v).

Given the size of the hash table $(Hash\_Table\_Size)$, each thread of a block calculates the address (h) in the table for v (row 2). The thread identifier $(thread_{id})$ and the visited vertex identifier (v) are merged into a single 64-bit word, to be then saved in the calculated address (row 3). The merge operation (as well as the consequent split in row 5) is efficiently implemented through bitwise instructions. A duplicate vertex causes the update of the hash table in the same address by more threads. Thus, each thread recovers the two values in the corresponding address (rows 4, 5) and checks whether they have been updated (row 6) to notify a duplicate. In particular, the recovered information classifies a vertex v as follows:

hash64

```
1:    H_SZ : Hash_Table_Size
2:    $h = \textbf{hash}(v)$        $\rightarrow h \in [0, \text{H\_SZ}]$
3:*    HashTable$[h]$ = $\textbf{merge}(v, thread_{id})$
4:    $recover$ = HashTable$[h]$;
5:*    $(v_R, thread_{idR})$ = $\textbf{split}(recover)$
6:    return $thread_{id} \neq thread_{idR} \wedge v = v_R$
```

*volatile int2 are not supported in CUDA

FIG. 6.10: *Main steps of the hash table managing algorithm*

- If $v = v_R$ and $thread_{id} = thread_{idR}$: the vertex is valid (not a duplicate).
- If $v = v_R$ and $thread_{id} \neq thread_{idR}$: the vertex is a duplicate.
- If $v \neq v_R$: there has been a conflict, that is, different threads wrote in the same hash table address (i.e., $hash(v) = hash(v_R)$). Since it is not possible to know whether the conflict hides a valid or a duplicate vertex, v is conservatively maintained in the frontier.

Conflicts are proportionally related to the size of the hash table and, thus, to the size of shared memory allocated for the hash table. As explained in Section 6.3.5 and shown in Figure 6.6, the setting of the *FrontierLimit* knob to run a single block rather than a multi-block kernel directly impacts on the hash table size and, thus, to the capability of *BFS-4K* of detecting duplicate vertices rather than conflicts.

The vertex classification is feasible for threads of the same warp, since they are synchronized at each instruction of the procedure and each access to the hash table is atomic. When duplicates are generated by threads of different warps of the same block, the procedure detects the duplicate whenever the warp scheduling does not generate a race condition. For example, the sequence:

$HashTable[h] = merge(v_x, thread_1)$
$HashTable[h] = merge(v_x, thread_2)$
$recover = HashTable[h]$ // by $thread_1$
$recover = HashTable[h]$ // by $thread_2$

allows the procedure to detect the duplicate, while the sequence:

$HashTable[h] = merge(v_x, thread_1)$
$recover = HashTable[h]$ // by $thread_1$
$HashTable[h] = merge(v_x, thread_2)$
$recover = HashTable[h]$ // by $thread_2$

does not allow the procedure to detect the duplicate, which is conservatively main-

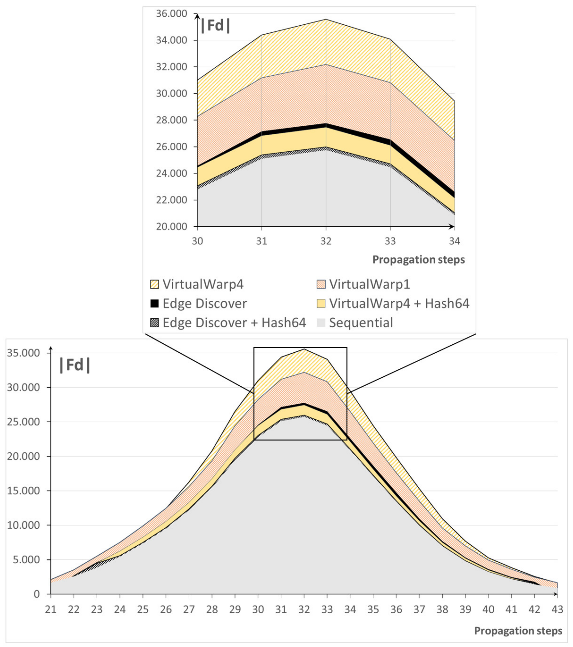


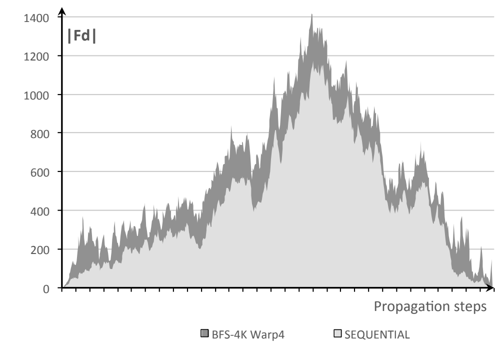
FIG. 6.11: *Example of duplicates caused by different visiting techniques and the effect of the proposed detection strategy*

tained in the frontier. Duplicates generated by threads of different blocks are not detectable.

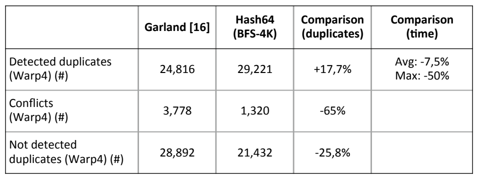
Since the duplicate issue occurs mainly among threads of the same warp, the problem affects more the visit by virtual warps than by edge-discover. Indeed, since in edge-discover the exploration is performed on edges, the chances to visit the same vertex more times is considerably small. Figure 6.11 shows the problem with the different visit strategies and the efficiency of the implemented technique of duplicate detection.

The virtual warp size (1 and 4 in the figure) is proportionally related to the number of duplicates. The sequential visit does not suffers from duplicates. Plots *Edge Discover + Hash64* and *VirtualWarp4 + Hash64* represent the frontier sizes obtained by combining the duplicate detection technique to the dynamic virtual warps and edge-discover, respectively. *VirtualWarp1 + Hash64* overlaps *VirtualWarp4 + Hash64* and has not been reported in the figure for the sake of clarity.

For the best of my knowledge, the duplicate detection and correction problem has been addressed in literature only in [107]. Differently from the proposed solution, [107] implements a hash table per warp and a procedure that writes and reads $lane_{id}$ (instead of $thread_{id}$) and v non atomically in the hash table. This involves more overhead due to the number of memory accesses and, by implementing disjointed hash tables, it suffers more from conflicts and non detectable duplicates. Figure 6.12 shows a representative example in which the techniques implemented in [107] and in *BFS-4K* are compared. In particular, Figure 6.12(a) shows the duplicates generated by adopting virtual warp of size 4 over the prop-



(a)



	Garland [16]	Hash64 (BFS-4K)	Comparison (duplicates)	Comparison (time)
Detected duplicates (Warp4) (#)	24,816	29,221	+17,7%	Avg: -7,5% Max: -50%
Conflicts (Warp4) (#)	3,778	1,320	-65%	
Not detected duplicates (Warp4) (#)	28,892	21,432	-25,8%	

(b)

FIG. 6.12: *Comparison between the duplicate detection techniques implemented in [107] and in BFS-4K*

agation steps. Figure 6.12(b) reports the total number of detected duplicates for both the solutions and the corresponding improvement on the overall performance (average and maximum improvement). The figure also reports the total number of conflicts and non detected duplicates.

6.5 Experimental results

Figure 6.13 summarizes the differences between the most representative BFS implementations at the state of the art and *BFS-4K*.

BFS-4K has been run on two main sets of graphs. The first set is from Stanford Network Analysis Platform (SNAP) [252]. It includes graphs from different contexts, such as, product co-purchasing networks, web page hyperlink graphs, network with ground-truth communities, road networks, social networks, time-evolving graphs and small-word phenomenon graphs. The second set is from the 10th DIMACS Implementation Challenge [1]. The $random.2Mv.128Me$ and $rmat.2Mv.128Me$ datasets have been generated by using GTGraph [3]. Table 6.1 shows each graph characteristics in terms of number of vertices (V, in millions), edges (E, in millions), size of graph diameter, average degree, standard deviation, and mode. *BFS-4K* has been run on a NVIDIA GEFORCE GTX 780 device [7] with CUDA Toolkit 5.0, with AMD Phenom II X6 1055T (3GHz) host processor (Ubuntu 10.04 operating system).

	Harish [117]	Virtual Warps [125]	Edge Parallelism [134]	Luo [172]	Garland [107]	*BFS-4K*
Work complexity	$O(VD+E)$	$O(VD+E)$	$O(ED)$	$O(V+E)$	$O(V+E)$	$O(V+E)$
Space complexity	$O(3V+E)$	$O(2V+E)$	$O(2E)$	N/A	$\Omega(4V+2E)$	$\Omega(4V+E)$
Type of parallelismn	Vertices	Virtual Warp	Edges	Vertices	Vertices, Edges, CTA	Vertices, Edges, *Dynamic* Virtual Warp, Dynamic Parallelism
High-degree vertex management	no	yes	indifferent	no	yes	yes
Duplicate detection	no	no	no	no	yes	yes
Type of synchronization	Host-Device	Host-Device	Host-Device	Host-Device, Inter-block [281], Thread barriers	Host-Device Inter-block [281]	Host-Device, Inter-block [281], Thread barriers

FIG. 6.13: *Comparison of the most representative BFS implementations at the state of the art with BFS-4K*

Figure 6.14 shows an example of the impact of each *BFS-4K* feature (Section 6.3) and the duplicate detection and correction technique (Section 6.4) on the overall speedup. The feature contributions are shown for a sample graph (*as-skitter*), by taking the speedup of the parallel BFS implementations in [117] and [134] versus the sequential implementation as reference point. The figure underlines that the best speedup is achieved by the combination of these features. In particular, the best feature configuration and combination can be obtained by properly setting the presented *knobs*. As explained in the follows, such a setting is correlated to the characteristics of the visited graphs and the characteristics of the GPU device.

Table 6.2 and Figure 6.15 report the performance comparison of *BFS-4K* with the most representative implementations at the state of the art in terms of visiting time and speedup, respectively. The performance of the state-of-the-art implementations are the best ones we obtained by tuning the kernel configurations (in terms of number of threads per block and number of blocks per grid) for the GPU device used. For the static virtual warp technique [125], Table 6.2 reports the size of

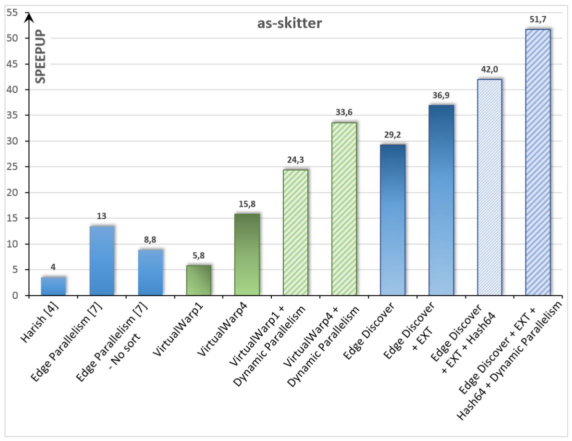


FIG. 6.14: *Impact of virtual warp, edge discover, dynamic parallelism and duplicate detection*

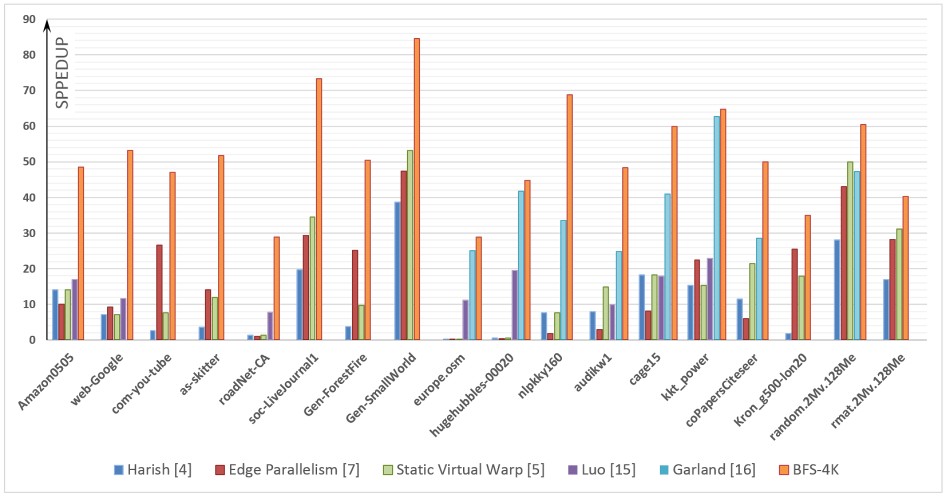


FIG. 6.15: *Performance comparison (speedup) of BFS-4K with the most representative implementations at the state of the art*

virtual warp statically set to obtain the best performance results. The Garland's implementation [107] does not support the template representation of the first set of graphs.

The results show how *BFS-4K* outperforms all the other implementations in every graph. This is due to the fact that *BFS-4K* exploits the more advanced architecture characteristics (in particular, Kepler features) and that it allows the user to optimize the visiting strategy through the knobs ($K_1 - K_7$).

We observed that K_1 is strictly related to the graph standard deviation and average degree. In particular, we measured the best speedups by increasing this

		V (10^6)	E (10^6)	Approx. Diameter [279]	Avg. Degree	Std. Deviation	Mode
Set 1	Amazon0505	0.4	3.4	40	8.2	3.1	10
	web-Google	0.9	5.1	34	5.6	6.5	456
	com-youtube	1.2	6.0	24	5.2	50.2	28,754
	as-skitter	1.7	22.2	31	12.1	136.9	35,455
	roadNet-CA	2.0	5.5	865	2.8	1.0	12
	soc-LiveJournal1	4.8	69.0	19	14.2	36.1	20,293
	Gen-ForestFire (f:0.35,b:0.32,s:1)	1.0	7.3	19	7.3	38.3	2,416
	Gen-SmallWorld (k:10 ,p: 0.3)	2.0	40.0	7	20.0	2.3	32
Set 2	europe.osm	50.9	108.1	30,102	2.1	0.5	13
	hugehubbles-00020	21.2	63.6	7,905	3.0	0.0	3
	nlpkkt160	8.3	221.2	162	26.5	2.7	27
	audikw1	0.9	76.7	81	81.3	42.4	344
	cage15	5.2	94.0	56	18.2	5.7	46
	kkt_power	2.1	13.0	49	6.3	7.5	95
	coPapersCiteseer	0.4	32.1	34	73.9	101.3	1,188
	kron_g500-lon20	1.0	100.7	7	96.0	1,033.2	413,378
	random.2M.128M	2.0	128.0	5	64.0	10.6	183
	rmat.2M.128M	2.0	128.0	5	64.0	136.8	8,785

TABLE 6.1: Characteristics of the graph datasets on which *BFS-4K* has been evaluated

knob value proportionally to the graph deviation and degree, starting from the lowest value ($K_1 = 4$) for graphs with low deviation and degree (e.g., road networks in general, *web-Google*, *kkt_power*, etc.), to the highest value ($K_1 = 32$) in graphs with high average (e.g., *random.2Mv.128Me*) or high standard deviation (e.g., *com-youtube*).

K_2 controls the use of dynamic parallelism, which achieves the best results with very high mode networks (e.g., mode greater than 2048 such as in *as-skitter*, *rmat.2Mv.128Me*, etc.) to deal with the sporadic high workloads. K_2, which maximum value is 0.15% in the experiments, should be higher for graphs with low average and inversely proportional to K_1. As explained in Section 6.3.3, the larger is the minimum warp size, the smaller is the sub-set of vertices that can be managed by dynamic kernels to improve the BFS performance. This is due to the fact

		Harish [117] (ms)	Edge Parall. [134] (ms)	Static Virtual Warp [125] (ms)	Luo [172] (ms)	Garland [107] (ms)	BFS-4K (ms)
Set 1	Amazon0505	5.2	7.2	5.2 (W1)	4.3	–	1.5
	web-Google	12.0	9.2	12.0 (W1)	7.3	–	1.6
	com-youtube	57.0	5.5	19.0 (W4)	out-of-time	–	3.1
	as-skitter	95.0	24.0	28.0 (W4)	out-of-time	–	6.5
	roadNet-CA	120.7	154.4	120.7 (W1)	20.2	–	5.5
	soc-LiveJournal1	91.0	61.0	52.0 (W2)	out-of-time	–	24.4
	Gen-ForestFire	37.0	5.4	14.0 (W4)	out-of-time	–	2.7
	Gen-SmallWorld	33.0	27.0	24.0 (W2)	out-of-time	–	15.1
Set 2	europe.osm	59,620.0	78,422.0	59,620.0 (W1)	684.0	305	264.8
	hugehubbles-00020	8,123.0	11,922.0	8,123.0 (W1)	220.0	103	95.9
	nlpkkt160	351.0	1486.0	351.0 (W1)	out-of-memory	80.4	39.2
	audikw1	68.0	185.0	36.0 (W4)	54	21.5	11.1
	cage15	95.0	213.0	95.0 (W1)	96	42.2	28.8
	kkt_power	36.0	24.5	36.0 (W1)	24	8.8	8.5
	coPapersCiteseer	21.2	40.6	11.4 (W4)	out-of-memory	8.6	4.9
	kron_g500-lon20	675.0	47.4	67.0 (W32)	out-of-time	out-of-memory	34.4
	random.2Mv.128Me	112.0	73.0	63.0 (W16)	out-of-time	66.5	52.0
	rmat.2Mv.128Me	103.0	62.0	56.0 (W4)	out-of-time	out-of-memory	43.4

Table 6.2: Performance comparison (BFS visiting time) of *BFS-4K* with the most representative implementations at the state of the art

that large virtual warps can handle the workload imbalance more efficiently (i.e, with less overhead) than dynamic parallelism.

K_3 impacts on the block size of *child kernels* when applying dynamic parallelism. The right value is more related to the GPU device characteristics and should be optimized heuristically. In the experimental results, $K_3 = 16$ provides the best BFS performance for all the analysed graphs.

K_4 controls the edge-discover technique to contrast the workload imbalance and it is strongly related to the standard deviation and average. We set $K_4 = 2$ for graphs with low average and high standard deviation (e.g., *com-youtube*, *ForestFire*, etc.). We decreased K_4 to 1 for graphs with medium standard deviation (e.g., *kkt_power*, *web-Google*, etc.). The edge-discover technique should not be used ($K_4 = 0$) with graphs with both high average degree and high standard deviation since, in these cases, the virtual warp size is expected to be high. This is due to the fact that the assignment of edges (rather than vertices) to threads is more efficient for high degree vertices.

The use of the single-block rather than the multi-block kernel is ruled by K_5, which is strictly related to the average degree and, though to a lesser extent, to the standard deviation. The single-block kernel should not be used ($K_5 = 0$) in graphs with high average since they provide enough parallelism for the multi-block kernel. In the experiments, we mainly set $K_5 = 1$ to provide a good trade-off between parallelism and synchronization.

Finally, K_6 and K_7 sets the block size in the multi-block kernel and the threshold for switching the writing mode in global memory, respectively. They best values depend on the GPU device characteristics. In the experiments, we heuristically set $K_6 = 128$ and $K_7 = 10$.

6.6 Concluding remarks

This Section presented *BFS-4K*, a parallel implementation of BFS for Kepler GPU architectures. *BFS-4K* implements different techniques to deal with the potential workload imbalance and thread divergence caused by any actual graph non-homogeneity. The Section presented an analysis of the advantages and limits of each proposed technique to understand how and when they can be applied and combined to improve the performance of the BFS visits. The Section also showed how such techniques can be calibrated through several knobs to customize *BFS-4K* depending on both the GPU device characteristics and the graphs to be visited. Finally, a comparison between the most efficient BFS implementations for GPUs at the state of the art and *BFS-4K* is reported to underline the efficiency of the proposed solution.

7

Breadth-First Search - Helix

Breadth-first search (BFS) is a core primitive for graph traversal and a basis for many higher-level graph processing algorithms. It is also representative of a class of algorithms whose implementation for graphic processing units (GPUs) is a very challenging task. Indeed, the irregularity of the input datasets makes each of the many parallel solutions proposed in literature suitable only for specific graph characteristics. This Section presents *Helix*, a fully configurable BFS for GPUs. It relies on a flexible and expressive programming model that allows selecting, for each BFS feature (e.g., frontier handling, load balancing, duplicate removing, etc.) and among different implementation strategies of them, the best combination to address the graph characteristics. The Section presents the analysis conducted on a large set of representative real-world and synthetic graphs to understand the correlation between graph characteristics and BFS configurations. The results show that *Helix* allows reaching throughput up to 14,000 MTEPS on single GPU devices, with speedups ranging from 1.2x to 18.5x with regard to the best parallel BFS solutions for GPUs at the state of the art.

7.1 Introduction

Breadth-first search (BFS) is a fundamental technique for graph exploration and analysis, and it is used as core primitive in a great variety of graph algorithms. The irregular nature of the problem and its high variability over multiple dimensions such as, graph size, diameter, and degree distribution, make the parallel implementation of BFS for graphics processing units (GPUs) a very challenging task.

Different parallel BFSs for GPUs have been proposed to efficiently deal with such issues during graph traversal [118, 125, 190, 275]. Although they provide good results for specific graph characteristics, no one of them is flexible enough to be considered the most efficient for any input dataset. This makes each of these solutions, and in turn the higher level algorithm in which they are included, not efficient in several circumstances (in some cases, less efficient than the sequential implementation [172]).

This Section presents *Helix*, a fully configurable BFS for GPUs. It includes many strategies that can be adopted to deal with any implementation issue (e.g., node-based mapping, scan-based, binary search for *load balancing* and vertex, edge, hybrid queues for the *frontier implementation*). Some of them have been collected from the literature and properly re-implemented to fully take advantage of the most recent programming and architectural characteristics of GPUs. Other strategies have been defined and implemented ex-novo to complete the *Helix* flexibility to cover the large class of graph characteristics. Thanks to a flexible and expressive programming model, *Helix* allows selecting the best implementation strategy for each BFS issue by considering the characteristics of the given graph.

The Section presents an analysis conducted on a large set of representative real-world and synthetic graphs to understand the correlation between graph characteristics and BFS configurations. The results show that some widespread strategies adopted to deal with specific implementation issues (e.g., binary search for load balancing) are the most efficient when considered singularly while not the best when combined to the others for building a complete BFS solution.

Finally, the analysis results show that average degree, Gini coefficient, and maximum degree of a graph are the necessary and sufficient information to correctly customize the BFS for any input dataset. The results show that *Helix*, which is available for download from *https://profs.scienze.univr.it/bombieri/Helix*, allows reaching throughput up to 14,000 MTEPS on single GPU device, with speedups from 1.2x to 18.5x with regard to the best parallel BFS solutions at the state of the art in all the analysed graphs.

The Section is organized as follows. Section 7.2 presents some background. Section 7.3 presents *Helix* in details. Section 7.4 presents the experimental results, while Section 7.5 is devoted to the concluding remarks.

7.2 Parallel graph traversal through BFS

Graph traversal consists of visiting each reachable vertex in a graph from a given set of root vertices. Given a graph $G = (V, E)$ with a set V of vertices, a set E of edges, and a source vertex s, the parallel graph traversal through BFS explores the reachable vertices level-by-level starting from s. Algorithm 4 illustrates the main structure of such an algorithm, which, in the parallel version, is commonly based on *frontiers*. The algorithm expresses the parallelism in two in two `for` loops (lines 3 and 4). The first loop provides trivial parallelism by iterating over the frontier vertices. In contrast, the second (nested) loop requires advanced parallelization techniques. This is mainly due to the fact that the loop aims at exploring the immediate neighbors of the frontier vertices, each of them requires a different number of iterations, and such a number depends on the out-degree of the vertices. As the corresponding sequential algorithm, the frontier-based BFS algorithm shows linear work-complexity $\mathcal{O}(V + E)$.

Algorithm 4 FRONTIER-BASED PARALLEL GRAPH TRAVERSAL

```
    Input: $G(V, E)$ graph, $s$ source vertex
    $\forall v \in V \backslash s : \ v.dist = \infty$
    $s.dist = 0$
    $Frontier_1 = s$
    $Frontier_2 = \varnothing$
 1: level = 1
 2: while $Frontier_1 \neq \varnothing$ do
 3:     Parallel for $v \in Frontier_1$ do
 4:         Parallel for $u \in neighbor(v)$ do
 5:             if $u.dist = \infty$ then
 6:                 $u.dist = level$
 7:                 INSERT($Frontier_2, u$)
 8:
 9:         end
10:     end
11:     BARRIER
12:     level = level + 1
13:     SWAP($Frontier_1, Frontier_2$)
14:     $Frontier_2 = \varnothing$
15: end
```

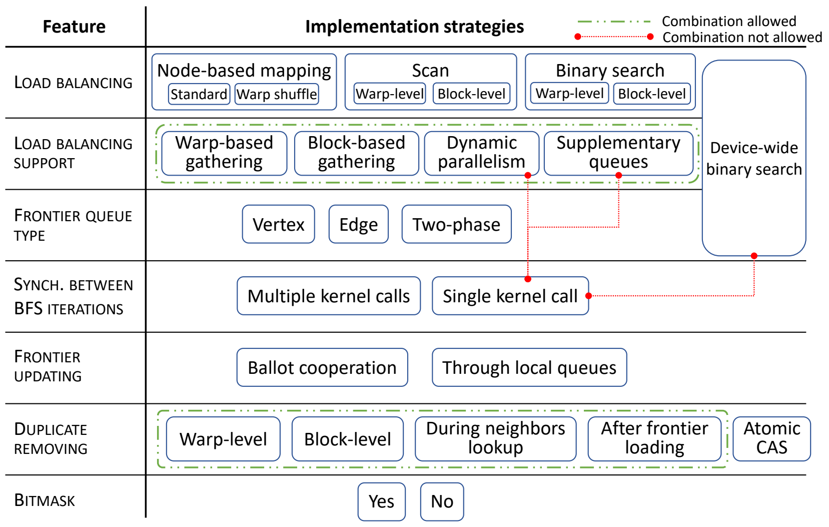


FIG. 7.1: *Helix framework overview.*

7.3 The *Helix* framework

Helix allows selecting, combining, and evaluating different implementation strategies for each basic feature of the BFS algorithm. Figure 7.1 shows the framework overview, which lists the features of the parallel BFS and the corresponding possible implementation strategies for GPUs. Some strategies have been collected from

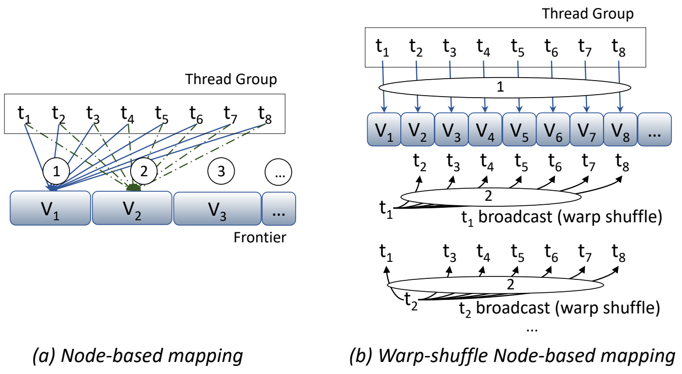

Thread Group
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8
1
2
3
...
V_1 V_2 V_3 ...
Frontier
(a) Node-based mapping
Thread Group
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8
V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8 ...
t_1 broadcast (warp shuffle)
t_2 broadcast (warp shuffle)
(b) Warp-shuffle Node-based mapping

FIG. 7.2: *Node-based mapping strategies.*

the literature and properly re-implemented to fully take advantage of the most recent programming and architectural characteristics of GPUs. Other strategies have been added ex-novo. The following sections present and compare the strategies in details as well as the programming model. The framework relies on a smart programming model to reduce the code size and to simplify the kernel procedure implementations. It also relies on a novel graph representation and provides an optimized prefix-sum implementation as described in the following sections.

7.3.1 Load balancing

Helix allows addressing the load balancing issue of the parallel BFS visit in four different ways. They rely on *node-based mapping*, *scan-based procedures*, *binary search*, or *device-wide binary search.*

The *node-based* technique partitions the workload by directly mapping groups of threads to the edges of each frontier vertex. The left-most side of Figure 7.2 shows an example, in which the 8 threads of a thread group first access to the vertex V_1 identifier in parallel and, then, each thread calculates the corresponding edge to be processed. Then, in sequence, the whole thread group moves to the other frontier vertices. The thread group size can be set depending on the average degree of the graphs (smaller warp sizes for graphs with lower average degrees). Nevertheless, in case of large thread group sizes, it may lead to many non-coalesced memory accesses during the frontier loading (8 accesses in the example), which in turn cause a strong loss of performance.

In these cases (i.e., large thread group sizes), *Helix* allows adopting an improved version of the node-based technique, which combines *warp shuffle* instructions to the direct thread-to-edge mapping to guarantee high performance. The right-most side of Figure 7.2 shows an idea of the strategy. Each thread accesses to a different frontier vertex and broadcasts the vertex identifier to the threads through warp shuffle (the frontier loading in the example requires 1 coalesced memory access at the cost of a minimum overhead involved by the warp-shuffle instructions).

The *scan-based* load balancing strategy is an alternative of node-based mapping. Instead of directly mapping threads to edges, each thread organizes the own edge offsets in shared memory through scan operations. *Helix* provides a *warp-level*

Algorithm 5 Optimized warp-level binary search

```
Input: Sequence of values represented by the variable val
       of each thread; value to search: searched
Output: lower bound of searched
low = 0;
#pragma unroll
for ( $i = 1$;  $i \leqslant \log_2$(WarpSize);  $i$++ ) do
    pos = low + (WarpSize $\gg i$); //$\gg$: compile time evaluated
    if (searched $\geqslant$ __shfl(val, pos)) then
        low = pos;
end
return low;
```

scan strategy that exploits an optimized prefix-sum procedure (see Section 7.3.10), that exploits the whole shared memory during the *frontier expansion* phase, and that adopts the warp-synchronous paradigm to avoid any kind of explicit synchronization. The *block-level scan* strategy follows the same steps, even though each iteration involves additional overhead for the mandatory thread synchronization through barriers.

The *binary search* technique provides the best load balancing among all threads at *warp level*, at the cost of additional computation. With even more computation, such a balancing can be guaranteed at *block level*. *Helix* implements an efficient version of the binary search that fully exploits the GPU shared memory. The algorithm consists of four steps:

(1) It computes the prefix-sum of the out-degrees of the frontier vertices.
(2) It executes an *optimized binary search* to equally partitioning the workload among threads. Algorithm 5 shows the pseudo-code of such an optimized binary search at warp-level (the binary search at block level is similarly implemented).
(3) It stores and reorganizes the edge offsets in shared memory.
(4) It processes the shared memory elements in parallel.

Thanks to the organization of the frontier information into shared memory, the binary search allows the following operations on the edge offsets to be performed through coalesced memory accesses. As for the *scan-based* technique, this strategy has been implemented by adopting the warp-synchronous paradigm to avoid barriers among warps of the same block. In general, the warp-level binary search provides perfect load balancing only among warp threads, while the block-level technique guarantees uniform workload among warps of the same block. For this reason, the block-level binary search best applies to highly irregular graphs.

The *device-wide binary search* strategy is a special case of the previous solution and guarantees equal workload among all threads of the GPU device. It implements a revisited algorithm of the *merge-path strategy* proposed by Green et al. [113] in the context of merge-sort, which is strongly oriented to graph traversal. Given the prefix-sum of the out-degrees and the edge offsets of the frontier vertices, the algorithm consists of three main steps:

(1) A simple and fast kernel computes the binary search over the whole workload to uniformly partition the frontier edges among the grid blocks (Figure 7.3(a) shows an example, where p_i are the prefix-sum elements and c_i are the equally sized chunks of elements). The size of a workload chunk (i.e., the number of edges per chunk) is equal to the available shared memory per block.

(2) A second kernel applies a block-level load partition by following the steps of the binary search. Each block identifies the corresponding workload chunk by using the offsets calculated by the first kernel. This step generates the neighbor frontier starting from the edge offsets (Figure 7.3(b)).

(3) A third kernel generates all the information to build the new frontier (Figure 7.3(c)). The kernel procedure executes the status lookup and update of the frontier elements, it removes previously visited vertices, and it computes an *online unordered prefix-sum*. In this particular case, the online procedure computes the prefix-sum of the out-degrees and of the number of warp elements at the same time. This allows avoiding double memory accesses to compute the prefix-sum offline through a specialized kernel procedure, which must load and store the degrees of the frontier vertices. The two information are merged into a single value through the 64-bit `atomicAdd` instructions. A *second optimization* has been implemented to discard vertices with out-degree equal to zero (for directed graphs) and equal to one (for undirected graphs) since they never contribute to the new frontier generation. In general, such an optimization is useful in power-law graphs since they present a high number of *leaf* vertices (up to 20% in some instances).

The basic implementation of the device-wide binary search sets the workload chunk size proportional to the available shared memory per block. It is suitable for large frontiers, but it involves inactive threads in case of small frontiers. Such a search strategy has been implemented in *Helix* with a *third optimization*, which allows dynamically configuring the workload chunk size between BFS iterations as follows:

$$\min \begin{cases} \left\lceil \dfrac{\textit{sum of out-degrees}}{\textit{\#resident threads}} \right\rceil \cdot \textit{block\_size} \\ \\ \textit{shared\_mem\_per\_block} \end{cases}$$

The device-wide binary search is an atomic strategy. Because of its radical structure, it cannot be combined with any of the other frontier queue or load balancing support techniques.

7.3.2 Load balancing support techniques

Helix provides a set of strategies to support load balancing. They can be singularly applied or combined to address the irregularity of the graph to be visited.

The *warp-based gathering* consists of collecting frontier vertices with out-degree greater than a threshold (*warp_size*) and, then, cooperatively processing their adjacency lists among warp threads.

Helix implements such a gathering through a low latency *binary prefix-sum* (see Section 7.3.10). The prefix-sum is computed on the condition upon the threshold

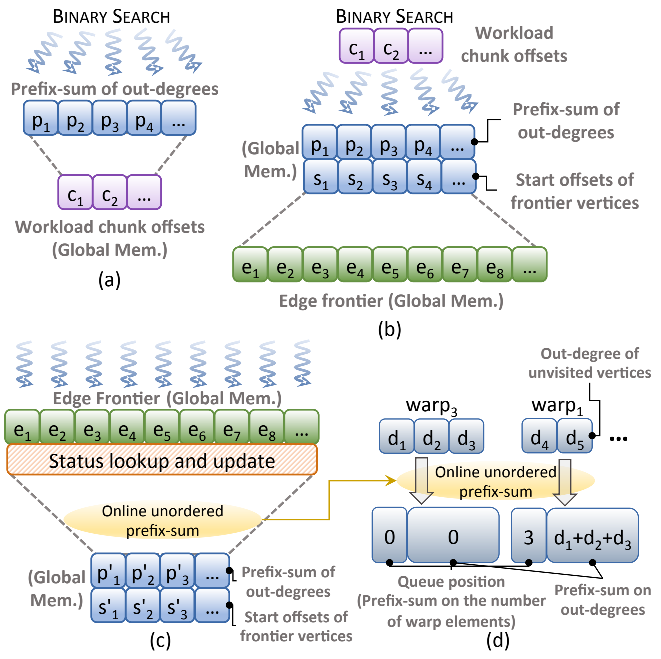

BINARY SEARCH
Prefix-sum of out-degrees
p1 p2 p3 p4 ...
c1 c2 ...
Workload chunk offsets (Global Mem.)
(a)
BINARY SEARCH
c1 c2 ...
Workload chunk offsets
Prefix-sum of out-degrees
(Global Mem.)
p1 p2 p3 p4 ...
s1 s2 s3 s4 ...
Start offsets of frontier vertices
e1 e2 e3 e4 e5 e6 e7 e8 ...
Edge frontier (Global Mem.)
(b)
Edge Frontier (Global Mem.)
e1 e2 e3 e4 e5 e6 e7 e8 ...
Status lookup and update
Online unordered prefix-sum
(Global Mem.)
p'1 p'2 p'3 ...
s'1 s'2 s'3 ...
Prefix-sum of out-degrees
Start offsets of frontier vertices
(c)
Out-degree of unvisited vertices
warp3
warp1
d1 d2 d3
d4 d5 ...
Online unordered prefix-sum
0
0
3
d1+d2+d3
Queue position (Prefix-sum on the number of warp elements)
Prefix-sum on out-degrees
(d)

FIG. 7.3: *Overview of the device-wide binary search.*

of each thread (1 if *out-degree* > *warp_size*, 0 *otherwise*). The prefix-sum result allows storing the vertices to be processed in consecutive locations of the shared memory. This solution requires only a single iteration and one shared memory access for the whole computation.

The *block-level gathering* follows the same idea of the previous procedure. *Helix* implements the block-level gathering through a *single step* of *unordered binary prefix-sum*, which significantly reduces the technique complexity and improves its applicability to very irregular graphs.

Helix includes the *dynamic parallelism* presented by Busato et al. [60] to process vertices with *very* large out-degrees. It provides benefits only with a fine tuning of the out-degree threshold. A wrong setting of such a value leads to a sensible overhead (caused by the dynamic kernel) that may compromise the overall performance of the application.

Helix introduces a different strategy, which relies on *supplementary queues* to organize the high degree vertices in different *bins*. Each bin holds vertices with sizes of the same (approximate) power of two (see the example of Figure 7.4). More in detail, the i-th bin holds vertices with out-degree in the range $[2^{(b+i)}, 2^{(b+i+1)}]$, where 2^b identifies the base threshold, and b is tuned by the user. Such a classification allows running a single kernel for the different bins, properly configured for the bin characteristics. In *Helix*, the total number of grid threads has been set equal to the lower bound of the bin ($2^{(b+i)}$) times the number of bin elements. In this way, the worst case involves at most two memory accesses among elements in consecutive queues. Finally, the threshold of the last bin is limited to the maxi-

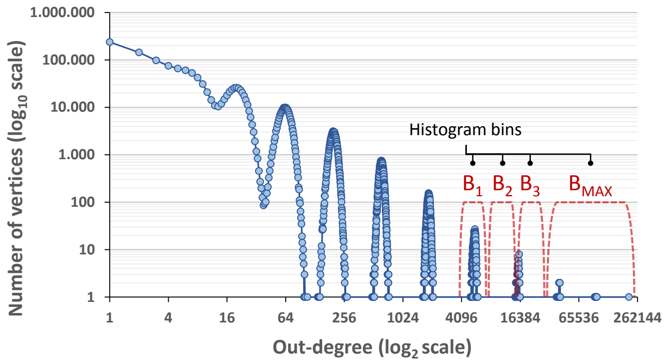


FIG. 7.4: *Example of the supplementary queues applied to the kron_g500-logn21 graph.*

mum number of resident device threads, since no more parallelism is possible for greater values.

7.3.3 Frontier queue types

The frontier queue represents the fundamental data structure of any work-efficient BFS. Such a queue stores either the vertices or the edges of the frontier, it allows an efficient neighbour exploration, and updates at each BFS iteration. Many approaches in literature store the *vertices* to be visited at the next iteration in the frontier queue [42, 60, 172, 190]. The amount of global memory accesses involved to maintain such a vertex queue is in the order of $2V$.

Two possible alternatives are the *edge queue* and the *two-phase queue* [190]. The *edge queue* stores the neighbors of the vertex frontier, and it involves $2E$ memory accesses. The *two-phase queue* alternates the two representations, by involving $2V + 2E$ memory accesses.

In general, the *vertex queue* allows minimizing the global memory data movement but suffers from thread inactivity during the status lookup[1]. On the other hand, the *edge queue* involves more memory accesses, but it provides better performance for the status lookup phase. The *two-phase queue* guarantees high thread utilization at the cost of a high number of memory accesses.

7.3.4 Synchronization between BFS iterations

Thread synchronization between BFS iterations strongly impacts on performance in the exploration of large diameter graphs. Implementing the whole graph traversal in one single kernel call requires synchronizing all threads at device level without returning to the host. This procedures can easily lead to low SM utilization or significant overhead at each iteration if not properly implemented.

Helix implements a strategy that guarantees a sound trade-off between SM occupancy and synchronization overhead. It restricts the number of blocks per SM

[1] The status lookup (e.g., updating of vertex distance) is the most expensive step of the algorithm due to the sparse memory accesses involved by the graph data structure.

to the maximum number of *resident* blocks (i.e., max blocks with resident threads), in which each block has size always greater than the number of grid blocks.

Helix also provides thread synchronization through explicit *multiple kernel calls* (i.e., one call per BFS iteration). Even though it introduces overhead at each kernel call, which strongly impacts on the overall performance especially in graphs with large diameter[2], it has three advantages: (i) it allows flushing the GPU caches between kernel calls, improving the hit rate for status lookups; (ii) it provides more flexibility in the kernel configuration; (iii) it improves the device occupancy for each kernel. Indeed, differently from the global synchronization, it does not require including many different synchronization functions into a single kernel, thus limiting the usage of thread registers.

7.3.5 Frontier updating

The frontier updating through write instructions is a key aspect to consider when implementing the linear-work graph traversal. *Helix* provides two alternatives: *Ballot cooperation* and *through local queues*. The first stores unvisited vertices at each step in global memory by using a fast *binary* prefix-sum. This approach allows for coalesced memory accesses and low-latency prefix-sum, but it involves a higher number of atomic operations. The second strategy relies on a *local register queue* for each thread, which holds unvisited vertices for the frontier generation. The warp threads coordinate each other through fast *voting* functions (see Section 2.2) to know when a thread fills his local queue. Then, each warp performs a prefix-sum and an atomic operation with the total sum of thread elements to reverse a chunk of frontier, and stores the concatenation of local queues. The strategy minimizes the atomic operations but involves sparse accesses due to non-consecutive memory addresses among warp threads.

7.3.6 Duplicate removing

Linear-work graph traversal can cause concurrent visits of the same vertex neighbors by different threads. This involves duplicate vertices in the frontier queue. *Helix* addressed this issue by hashing vertex identifiers in shared memory. The duplicate removing is implemented by merging vertex and thread id in a *single vector data*, store the result, and then recover the data. The duplicate vertices are progressively eliminated through multiple iterations and different hash functions in both warp and block memory spaces. *Helix* allows applying this process at different phases of the BFS iteration, including *during the neighbor lookup* and *after the frontier loading*.

Helix also allows applying the status lookup through atomic operations (`atomicCAS`), which completely eliminate duplicate vertices. On the other hand, such operations introduce more overhead since they are not as efficient as standard memory accesses.

[2] In the tested GPU devices, we experimentally measured such an overhead equal to $15\mu s$ per kernel call, which leads to a 15 ms overhead in graphs with one thousand depth. This overhead is consistent with other recent GPU devices [189, 260].

Graph	Category	U/D	V (M)	E (M)	Avg. degree	Std. deviation	Gini coeff.	Max degree	Avg. eccentricity
asia_osm	Road Network	U	12.0	25.4	2.1	0.5	0.08	9	36,626.7
europe_osm	Road Network	U	50.9	108.1	2.1	0.5	0.09	13	19,738.2
USA-road-d.USA	Road Network	U	23.9	58.3	2.4	0.9	0.21	9	6,418.6
hugebubbles-00020	Num. simulation	U	21.2	63.6	3.0	0.0	0.00	3	6,205.9
rgg_n_2_23_s0	Random Geometric	U	8.4	127.0	15.1	3.9	0.14	40	1,715.7
delaunay_n24	Structural	U	16.8	100.7	6.0	1.3	0.12	26	1,588.3
channel-500x100x100	Num. simulation	U	4.8	85.4	17.8	1.0	0.01	18	381.6
ldoor	Structural	U	1.0	47.5	49.9	11.9	0.13	78	161.4
nlpkkt160	Num. simulation	U	8.3	237.9	28.5	2.7	0.02	29	145.2
audikw_1	Structural	U	0.9	78.6	83.3	42.4	0.23	346	61.8
circuit5M	Circuit simulation	D	5.6	59.5	10.7	772.6	0.52	1,290,501	58.0
FullChip	Circuit simulation	D	3.0	26.6	8.9	23.1	0.35	2,312,481	38.3
cage15	DNA electrophoresis	D	5.2	99.2	19.2	5.7	0.17	47	37.3
indochina-2004	Social Network	D	7.4	194.1	26.2	215.8	0.74	6,985	31.0
soc-LiveJournal1	Social Network	D	4.8	69.0	14.2	36.1	0.72	20,293	14.3
soc-pokec-relationships	Social Network	U	1.6	61.2	37.5	59.5	0.62	20,518	10.2
er-fact1.5-scale23	Erdös-Rényi	U	8.4	200.6	23.9	4.9	0.12	53	7.8
hollywood-2009	Social Network	U	1.1	115.0	100.9	271.9	0.73	11,469	7.6
kron_g500-logn21	Kronecker	U	2.1	182.1	86.8	680.1	0.92	213,906	5.1

Table 7.1: Graph dataset.

7.3.7 Bitmask status lookup

The bitmap structure aims at improving the efficiency of the status lookup phase. Thanks to its compact size, it generally fits in the L2 cache and allows the threads to retrieve visited/unvisited information of frontier vertices without accessing the global memory. On the other hand, the massive parallelism of GPU threads often leads to *false negatives* (unvisited vertices that actually have been already visited). This is due to the concurrent accesses of threads to the same memory locations.

Helix implements a bitmap structure based on 8-bit data types (the smallest possible), which reduces the access overlapping and, as a consequence, the number of false negatives.

7.3.8 Programming model

Recent CUDA implementations provide full support of advanced C++11 language features. This allows reducing the code size and simplifying common kernel procedures. Figure 7.5 shows an example on how *Helix* organizes kernel code by exploiting such features. The commented lines represent possible configuration alternative.

The model relies on *C++ forward iterators* to coordinate the device threads during the frontier processing. It encapsulates the frontiers in a specialized struc-

```
1: auto it_item = vertex_item(Frontier);                          //vertex frontier
2: //auto it_node = edge_item(Frontier);                           //edge frontier
3: auto F = frontier<decltype(node)>(frontier_size, it_item);

4: load_balancing::BinarySearch LB;
5: //load_balancing::Scan LB;

6: auto lambda = [&](int limit) {
7:     writeOnFrontier(LB.SharedMem, limit); };

8: for (auto it : F)
9:     LB.exec(lambda, it);
```

FIG. 7.5: *Example of kernel function. The red text represents possible alternatives.*

ture (line 3) and defines the functionality to extract an element from the frontier in a separated procedure (line 1). Such iterators are extended to support *range-based loop* (line 8) to simplify and to make clearer the code.

All load balancing functions (lines 4-5) are defined in terms of input/output. They take in input the start and the end offset of a vertex provided by an iterator and generate the corresponding list of neighbors. Given the neighbor list, the functions allow applying any operation on them through *lambda expressions* (lines 6-7). The main advantage of the lambda construct is an easy mechanism to capture variables from the context without modifying the rest of the code.

The PTX code (intermediate representation) generated from the C++11 implementation has been compared to that generated by the standard implementation approach to ensure they produce the same PTX instruction sequence.

The framework strongly relies on C++ templates and constant expressions to save the register usage and to avoid unnecessary operations.

Thanks to the adopted programming model, *Helix* allows generating up to 45,824 different configurations, counting only ≈4,000 lines of code for the core functionality (i.e., without considering the input parsing). It does not require any other external sequential or parallel library (e.g., Boost, CUB [187], ModernGPU [31], etc.).

7.3.9 Modified Graph Representation

Most sequential and parallel graph applications adopt the *Compressed Sparse Row* (CSR) format to store graphs. The CSR format, commonly used in sparse linear algebra applications, allows storing graphs in the adjacency list representation with two arrays without using pointers. The *edges array* consists of the concatenation of adjacency lists, while the row-offset array is formed by the prefix-sum values of vertex degrees that correspond to adjacency list offsets in the edges array. The CSR graph representation shows great space efficiency, able to store graphs with billions of vertices/edges in few GBs. In contrast, it requires advanced load-balancing techniques to efficiently traverse the data structure.

All graph operations on the basic CSR representation involves two memory accesses for each thread to obtain the start and the end offsets of vertices. This leads to non-coalesced memory accesses through continuous addresses with a consequent performance penalty. To face such a problem, *Helix* adopts a modified graph representation, called *CSR-M*, which replaces the standard CSR row-offsets with an array of *two-component vector* data type. Each array element stores, in a single vector, the start and the end offset of the adjacency lists in the edge array. The *CSR-M* representation overcomes non-coalesced accesses by forcing vectorized accesses. The modified representation improves the performance by 10/15% for all graphs ad the cost of negligible additional memory space. The observation relies on the fact that, in general, graphs show a number of edges one or two orders of magnitude greater than the number of vertices.

7.3.10 Optimized Prefix-sum

Prefix-sum is a fundamental parallel primitive, used as a building block for designing a wide range of parallel algorithms and applications. In the context of graph traversal and load balancing we make extensive use of different prefix-sum algorithms and at different thread hierarchy levels for workload partition, reorganize sparse data, and coordinate threads. How such parallel primitive is implemented has a significant impact on the performance of the overall application.

The prefix-sum algorithms can be classified on the ordering of involved operations and on the produced output. Given an input sequence $a_1, a_2, \ldots, a_n$ the *ordered prefix-sum* algorithm computes the output result $a_1, (a_1 + a_2), \ldots, (a_1 + \ldots + a_n)$. We implement the warp prefix-sum by using intrinsic warp shuffle instructions combined with PTX assembly to elide wasted operations of inactive lanes during the computation.

A variant of the common prefix-sum algorithm, called *unordered prefix-sum*, does not guarantee a strict ordering of the output while maintaining monotonic increasing values in the resulting sequence. A possible output of unordered variant is $a_3, (a_3 + a_5), (a_3 + a_5 + a_1), \ldots, (a_1 + \ldots + a_n)$. The parallel implementation of such algorithm at global and block-level can take advantage of loose ordering to accelerate the computation. The unordered prefix-sum applies the same procedure of ordered variant at warp level but relies on atomic operations among different warps. In particular, each warp atomically updates a single value in shared memory for block-wide computation and in global memory for device-wide computation with the total sum of its values and getting back the previously stored value. Thanks to *hardware-implemented* atomic operations in both shared and global memory, Maxwell and Pascal architectures achieve better execution times for the unordered variant compared to the conventional ordered scan-then-fan algorithm [277].

Finally, a special case of the general prefix-sum, called *binary prefix-sum*, involves only boolean values. We implement the binary prefix-sum at warp-level with only three instructions by improving the procedure described in [121]. First, we evaluate the predicate for each lane with the `ballot` instruction, then we compute the bitwise AND between the ballot result and the lane lower mask (e.g. the lower mask of the lane 3 is `111`), and finally each lane counts the number of true values of preceding lanes with the `popc` instruction. The lane lower mask is obtained by reading a special register via PTX instruction instead computing multiple instructions.

7.4 Experimental Results

7.4.1 Graph dataset and system setup

We conducted the analysis and the performance evaluation of *Helix* on a dataset of 20 graphs, which includes both real-world and synthetic graphs from different application domains. Table 7.1 presents the graphs and their characteristics in terms of structure (directed/undirected), number of vertices (V, in millions), edges (E, in millions), average degree, standard deviation, Gini coefficient, maximum degree, and average eccentricity (or BFS depth). The Gini coefficient [151] measures the inequality among vertex degrees. It is complementary to the standard deviation information to express the graph irregularity and it is considered a clear and reliable alternative to the power-law. It is expressed in the range $[0, 1]$, where 0 indicates the maximum regularity among vertex degree (i.e., all vertices with equal degree), while 1 indicates the maximum inequality among vertex degree (i.e., there exists one vertex with degree equal to the total number of edges).

The graphs have been selected to be representative of a wide range of characteristics, including size, diameter, degree distribution (from regular to power-law).The variability of the dataset is essential to fully stress the proposed solution and the state-of-the-art counterparts under very different input types. The graphs have been selected from the University of Florida Sparse Matrix Collection [85], the 10th DIMACS Challenge [23], and the SNAP dataset [160].

We ran the experiments on a NVIDIA Maxwell GeForce GTX 980 device with CUDA Toolkit 7.5, AMD Phenom II X6 1055T 3GHz host processor, Ubuntu 14.04 O.S., and clang 3.6.2 host compiler with the -O3 flag. The GPU device consists of 16 SMs (2,048 SPs) capable of concurrently executing 32,768 threads. For a fair comparison, we did not perform any modification to the input graphs and we adopted the Matrix Market format (.mtx) representation for all the evaluated tools. We ran all tests 100 times from random sources to obtain the average execution time t_{avg}. The traversal throughput is computed as E/t_{avg} for all tools and is expressed in MTEPS (million traversed edges per second). For all the state-of-the-art BFS implementations, we considered the best performance results obtained by trying any possible configuration (if available).

7.4.2 The configurability analysis

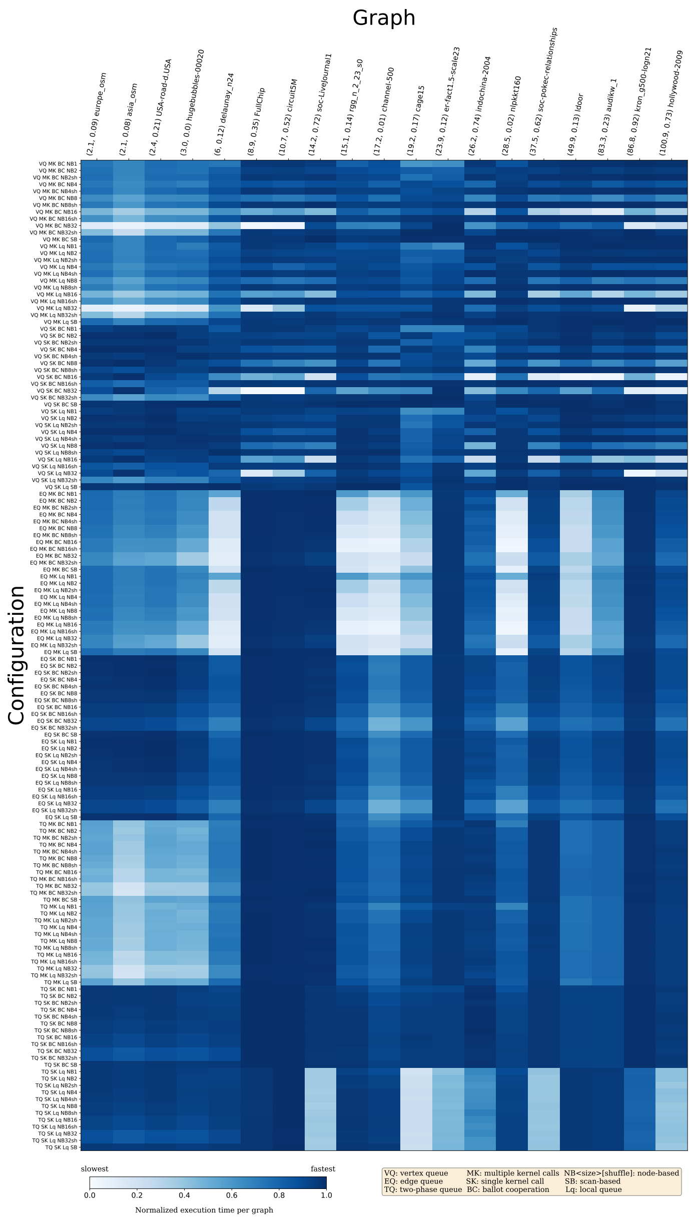


FIG. 7.6: *Configuration heatmap. The colour gradations represent the normalized performance of the Helix configurations*

We ran the analysis of *Helix* on the whole graph dataset to understand the correlation between graph characteristics and BFS configurations.

Figure 7.6 shows the results, which are represented through a heatmap. Each row represents a *Helix* configuration (i.e., a combination of the BFS features of Fig. 7.1). For the sake of clarity, the figure reports the subset of the most relevant (i.e., in terms of performance incisiveness) configuration features (i.e., load balancing, frontier queue type, synchronization, and frontier updating). Each column represents a graph, whose most significant characteristics (average degree and Gini coefficient) are reported in brackets for the following analysis. The performance (i.e., execution time) of each configuration for each graph has been normalized over the *Helix* best performance and represented by colour gradations (the darker the better).

The results show that, for load balancing, the *node-based mapping* technique is well-suited to graphs with regular degree distribution. This is due to the fact that the direct mapping of threads to work items implemented through node-based mapping, differently from the other balancing techniques, does not uselessly introduce overhead of complex mapping computations.

Node-based mapping with *warp shuffle* provide the best throughput when such instructions involve at least half warp threads (16) and if there exists *enough parallelism* (double thread group size) during the exploration of vertex neighbours

The *scan-based* load balancing is well-suited to irregular graphs thanks to the thread cooperation implemented by such technique in shared memory.

We found that both *binary search* and *device-wide binary search* are never better than the other balancing approaches for the BFS visit of any graph. This is due to the significant overhead they introduce in the workload partition procedures. In particular, even though they are the most efficient techniques from the balancing point of view for very irregular graphs and very popular in literature [31,42,83,275], we found that the best BFS configurations for such graphs do not include them. It is worth noting that the *binary search* load balancing implemented in *Helix* has speedups from 1.1x (in regular graphs) to 9.7x (in irregular graphs) with respect to the same technique at the state of the art that does not rely on edge reorganization in shared memory and on the unordered prefix-sum. The optimized *device-wide binary search* shows speedups from 1.9x to 4.28x compared to the Gunrock [275] implementation.

Warp-level and *block-level gathering* provide performance benefits when applied to graphs in which the maximum degree is greater than their corresponding work sizes (warp and block size, respectively). Below these thresholds, the efficiency of the techniques are affected by thread inactivity by construction. The *supplementary queues* provide the best efficiency in graphs with maximum degree greater than half of the available device threads, which corresponds to a threshold of 16,384 on the GeForce 980 GTX. Such a threshold is motived by the fact that the device is well exploited when at least half threads are active.

We found that *dynamic parallelism* never performs better than the supplementary queues for any graph of the dataset. For this reason, it does not belong to the best BFS configuration we identified for the analysed dataset.

The *vertex queue* is the most efficient solution to represent the frontiers thanks to the low number of memory accesses. The *edge queue* well applies only when the available parallelism is low and in case of irregular workload as it reduces the thread inactivity. For this reason, *Helix* adopts the *edge queue* combined with *node-based mapping 1* on graphs with low average degree and that present not uniform distribution.

Differently to the approach proposed by Merrill et al. [190], the two-phase queue does not provide the best performance. The two-phase queue can alleviate the workload unbalancing combined with simple techniques but, on the other hand, the high number of memory accesses strongly affect the performance. For these reasons, the two-phase queue does not belong to the best BFS configuration.

The *multiple kernel calls* strategy provides the best performance in graphs with average eccentricity less than 100. After such a value, the amount of overhead generated by each single kernel invocation eludes all the advantages provided by the strategy and makes the *single kernel call* a better alternative. Since the average eccentricity is not known a priori, we consider *average degree greater than 20 or Gini coefficient greater than 0.6* as a good heuristic for identifying graphs with average eccentricity less than 100.

The *ballot cooperation* strategy for frontier updating strongly relies on an efficient *single-step* warp-level procedure. For this reason, it provides the best performance in graphs with average degree less than the warp size. The technique requires more atomic operations than the *local queue* strategy, but the number of such operations does not affect the execution time in low-sized workloads. The *local queue* strategy performs less atomic operations but involves non-coalesced memory accesses. We also found that the *ballot cooperation* works well only in graphs with Gini coeff. greater than 0.9 as, in these cases, it allows preventing the workload (average degree) to be distorted by the distribution of the edges. Indeed, a Gini coeff. greater than 0.9 clearly indicates a strong power-law distribution in which most vertices have low degree [151].

We also found that the choice of the *duplicate removing* technique can be associated to the chosen load balancing technique. It reduces the redundant work in the frontier exploration by introducing additional computation. Removing duplicates *during the neighbor lookup* allows eliminating a high number of vertices, but the amount of introduced overhead completely eludes the advantages of the technique. For this reason, duplicate removing during the neighbour lookup does not belong to any *best* BFS configuration. In contrast, duplicate removing *after the frontier loading* provides more efficiency at the cost of a lower number of eliminated vertices. In the same way, *block-level* removing is more effective than the *warp-level* procedure, but it requires synchronization barriers if used in conjunction with load balancing techniques that rely on shared memory. For this reason, the best combination is removing *after frontier loading* at *warp-level* with *scan* load balancing, while removing *after frontier loading* at block-level with *node-based* load balancing. Duplicate removing through *atomicCAS* operations completely eliminates duplicate vertices at the cost of a considerable overhead. It never performs better than the other duplicate removing strategies for any graph of the dataset.

The choice of the *bitmask* application depends on how the input graph has been built. We observed a 70%-99% usage of the whole bitmask in synthetic graphs after the whole graph exploration. This is due to the fact that such graph vertices are randomly enumerated and, as a consequence, neighbor vertices have not close identifiers. This prevents concurrent accesses of threads to the same memory locations during the frontier exploration. In contrast, real-world graphs, for which

Feature	Rules
Load balancing	- *Node-based 1*: avg. degree $<$ 5 and Gini coeff. $\geqslant$ 0.2 (edge queue), - *Node-based 4*: avg. degree $<$ 5 and Gini coeff. $<$ 0.2, - *Node-based 16 (with shuffle)*: avg. degree $>$ Warp size and Gini coeff. $<$ 0.3, - *Node-based 32 (with shuffle)*: avg. degree $>$ Warp size * 2 and Gini coeff. $<$ 0.3, - *Scan-based (warp-level)*: otherwise
Load balancing support	- *Warp-based gathering*: max. degree $>$ Warp size, - *Block-based gathering*: max. degree $>$ Block size, - *Supplementary queues*: max. degree $>$ Half device threads
Frontier queue type	- *edge queue*: avg. degree $<$ 5 and Gini coeff. $\geqslant$ 0.2 (Node-based 1), - *vertex queue*: otherwise
Synchronization between BFS iterations	- *Multiple kernel calls.*: eccentricity $<$ 100 (avg. deg. $>$ 20 or Gini coeff. $\geqslant$ 0.6, heuristic) - *Single kernel call*: otherwise
Frontier updating	- *Local queue*: avg. degree $>$ Warp size and Gini coeff. $<$ 0.9 - *Ballot cooperation*: otherwise
Duplicate removing	- *After frontier load* at warp-level with *Scan-based* - *After frontier load* at block-level with *Node-based*
Bitmask	- Synthetic graphs and DNA electrophoresis

TABLE 7.2: Configuration table.

the vertex enumeration is implicitly done during the graph generation, present neighbor vertices with close identifiers. We measured a 20%-30% usage of the whole bitmask and a high number of *conflicts* during the frontier exploration in such non-randomly enumerated graphs. We claim that the bitmask technique is not suited to all graphs in which the vertex labeling follows the topological structure of the graph. In contrast, the bitmask provides positive speedups in synthetic graphs generated with random vertex labeling

Finally, from the large set of obtained data, we derived a *configuration table*, with the aim of matching the best (near-optimal) *Helix* configurations and the characteristics of the graphs to be visited. To do that, we applied a *decision tree* for multiclass classification [5]. Table 7.2 summarizes the result of such a regression, which reports, for each BFS feature, the graph characteristics considered to select the best implementation strategies. It is worth noting that average degree, Gini coefficient, and maximum degree are the necessary and sufficient information to correctly customize the BFS for every analysed graphs.

Many rules are expressed in function of GPU device parameters (warp size, block size, etc.) or according to architecture-independent graph properties, such as the Gini coefficient. A fine-tuning of *Helix* by considering also the characteristics of the specific GPU device is still possible, even though, by considering the preliminary results with different GPU architectures, it plays a minor role. It is part of current and future work.

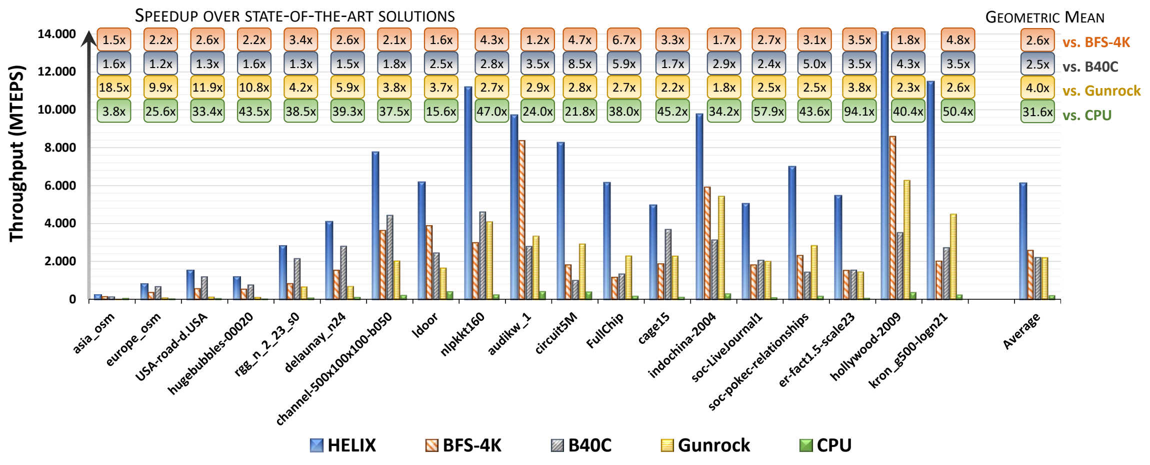


FIG. 7.7: *Performance comparison of Helix with the most representative implementations at the state of the art.*

7.4.3 Performance evaluation

Figure 7.7 shows the performance comparison of *Helix* (which has been set with the configurations reported in Table 7.2) with the best and most representative GPU implementations at the state of the art (B40C [190], Gunrock [275], BFS-4K [60]) and with a sequential CPU implementation for completeness. The figure reports the throughput (in MTEPS) and the speedup of *Helix* over the different solutions.

The results show that *Helix* is significantly faster than all the other implementations in all graphs. We observed that, in general, the throughput of the GPU implementations is strongly related to the average degree and the graph size.

All the GPU solutions provide lower throughput in the left-most graphs as these graphs do not allow for much parallelism during exploration. Despite the low average degree and the high eccentricity, *Helix* provides speedups from 3.8x to 43.5x with regard to the CPU implementation for the first four graphs of the dataset. This is due to the combination of the global synchronization strategy with an efficient load balancing technique. *Helix* has speedups from 1.2x to 1.6x w.r.t. B40C for these graphs since this last implements an edge frontier queue and a slightly less efficient global synchronization. *Helix* has speedups up to 18.5x w.r.t. Gunrock as this last supports only host-device synchronization. It is worth to note that the lack of global synchronization in Gunrock translates into an execution time higher than the sequential CPU implementation for the *asia_osm* graph.

The results show a significant throughput improvement of *Helix* compared to the other GPU solutions in graphs with a very high maximum degree (*circuit5M, FullChip, kron_g500-logn21, soc-LiveJournal1, soc-pokec-relationships*). This is due to the efficient *block-level gathering* and to the *supplementary queue* technique. Indeed, B40C provides a technique to process such vertices only at block-level, BFS-4K applies the dynamic parallelism which has been proved to be less efficient than the supplementary queues, while the device-wide binary search technique implemented in Gunrock suffers from high overhead despite the perfect load balancing.

Finally, the 8-bit bitmask implemented in *Helix* significantly improves the performance for synthetic graphs (*kron_g500logn21* and *er-fact1.5-scale23*), which present full-random labeling.

7.5 Conclusions

This Section presented *Helix*, a fully configurable BFS for GPUs. *Helix*, thanks to a flexible and expressive programming model, allows selecting the most efficient combination of features and their implementation strategy for a BFS visit (i.e., frontier handling, load balancing, duplicate removing, and other optimized supporting techniques). The Section presented the analysis conducted on a large set of representative real-world and synthetic graphs to understand the correlation between graph characteristics and BFS configurations. The experimental results showed that Helix provides high-performance and customized BFSs with speedups ranging from 1.2x to 18.5x with regard to the best parallel BFS solutions for GPUs at the state of the art, with peak throughput of 14,000 MTEPS on a single GPU device.

7.6 Appendix

This appendix describes how to download (from online repository), compile, and run *Helix*, the performance-oriented framework for graph traversal. It also describes how to download the datasets and to reproduce the results obtained in the article *"Helix: A Fully Configurable Breadth-first Search for GPUs"*.

Description

Check-list (SW meta information)

- **Algorithm:** Breadth-first Search, Graph Traversal.
- **Program:** CUDA and C++11 code.
- **Compilation:** NVIDIA nvcc (v7.5 or higher) CUDA compiler, GNU g++/gcc (v4.8.4 or higher) or LLVM clang++ (v3.6.2 or higher) host compiler, KitWare CMake (v3.5 or higher), and GNU make. All the tool versions listed above have been successfully tested.
- **Binary:** CUDA executable.
- **Data set:** Publicly available matrix market files (.mtx); SNAP dataset (.txt); DIMACS 9th dataset (.gr), DIMACS 10th dataset (.graph), Koblenz Network Collection (.out), Network Data Repository (.edges).
- **Run-time environment:** Ubuntu 14.04 64-bit O.S. with CUDA toolkit/driver v7.5.
- **Hardware:** Any CUDA GPU device with compute capability 3.5 or higher (i.e., with dynamic parallelism support). Tested on NVIDIA GeForce 980 GTX.
- **Execution:** Command line
- **Output:** Graph dataset statistics, elapsed running time, MTEPS throughput.
- **Experiment workflow:** download the project; download the datasets (see Section 7.6); run the tests; observe the results.
- **Experiment customization:** run-time parameters.
- **Publicly available?:** Yes.

How the Software can be obtained

The source code of *Helix* is available here:

`https://profs.scienze.univr.it/bombieri/Helix`

Hardware dependencies

For adequate reproducibility, we suggest an NVIDIA GPU device with compute capability 3.5 or higher (Kepler GPU architecture with support for dynamic parallelism). To fully exploit all features of *Helix*, more recent GPU architectures are necessary (Maxwell, Pascal, or more recent).

Software dependencies

The GPU graph traversal evaluation requires the CUDA GPU driver, nvcc CUDA compiler (v7.5 required, v8.0 or higher recommended), and KitWare CMake. No other additional libraries are required. The software has been tested on Ubuntu 14.04 64-bit O.S., but it has been implemented to run correctly under other Linux distributions.

Datasets

Helix supports six different input formats, for which several datasets are publicly available. Many of them, which are listed in the following, have been used for the experimental results reported in the Section:

1. `matrix market (.mtx)` :
   `www.cise.ufl.edu/research/sparse/matrices/`
2. `SNAP graph (.txt)` :
   `http://snap.stanford.edu/`
3. `DIMACS 9th (.gr)` :
   `http://www.dis.uniroma1.it/challenge9/`
4. `DIMACS 10th (Metis format)(.graph)` :
   `http://www.cc.gatech.edu/dimacs10/`
5. `Koblenz (.out)` :
   `http://konect.uni-koblenz.de/`
6. `Network Data Repository (.edges)`
   `http://networkrepository.com/index.php`

Installation

After cloning the repository, the following steps are required:

```
$ cd helix/build
$ cmake ..
$ make
```

Note: the code will be automatically configured for the first GPU device founded in the system, with no need to specify the compute capability.

Experiment workflow

To run the executable on the selected dataset:

```
$ Helix <input_graph> <options>
```

Note: use the `--help` commandline option to see the extended command line usage, including all options to configure the Helix framework (e.g., load balancing, bitmask, duplicate removing, etc.)

Evaluation and expected result

The expected results include graph statistics (number of vertices, number of edges, average degree, etc.), elapsed running time for the graph traversal, MTEPS throughput.

Experiment customization

It is possible to customize the graph traversal framework (e.g., load balancing, bitmask, duplicate removing, etc.) through run-time command line options or at compile-time by following the instructions included in the `CMakeList.txt` file in the root directory.

Notes

For up-to-date information, please visit the Helix project's page:
https://profs.scienze.univr.it/bombieri/Helix.

8

Single-Source Shortest Path - H-BF

Finding the shortest paths from a single source to all other vertices is a common problem in graph analysis. The Bellman-Ford's algorithm is the solution that solves such a single-source shortest path (SSSP) problem and better applies to be parallelized for many-core architectures. Nevertheless, the high degree of parallelism is guaranteed at the cost of low work efficiency, which, compared to similar algorithms in literature (e.g., Dijkstra's) involves much more redundant work and a consequent waste of power consumption. This Section presents a parallel implementation of the Bellman-Ford algorithm that exploits the architectural characteristics of recent GPU architectures (i.e., NVIDIA Kepler, Maxwell) to improve both performance and work efficiency. The Section presents different optimizations to the implementation, which are oriented both to the algorithm and to the architecture. The experimental results show that the proposed implementation provides an average speedup of 5x higher than the existing most efficient parallel implementations for SSSP, that it works on graphs where those implementations cannot work or are inefficient (e.g., graphs with negative weight edges, sparse graphs), and that it sensibly reduces the redundant work caused by the parallelization process.

8.1 Introduction

Given a weighted graph $G = (V, E)$, where V is the set of vertices and $E \subseteq (V \times V)$ is the set of edges, the single-source shortest paths (SSSP) problem consists of finding the shortest paths from a single source vertex to all other vertices [74]. Such a well-known and long-studied problem arises in many different domains, such as, road networks, routing protocols, artificial intelligence, social networks, data mining, and VLSI chip layout.

The de-facto reference approaches to SSSP are the Dijkstra's [91] and Bellman-Ford's [38, 102] algorithms. The Dijkstra's algorithm, by utilizing a priority queue where one vertex is processed at a time, is the most efficient, with a computational complexity almost linear to the number of vertices ($O(|V| \log |V| + |E|)$).

Nevertheless, in several application domains, where the modelled data maps to very large graphs involving millions of vertices, any Dijkstra's sequential implementation becomes impractical. In addition, since the algorithm requires many iterations and each iteration is based on the ordering of previously computed results, it is poorly suited for parallelization. Indeed, the parallel solutions proposed in literature for graphics processing units (GPUs) [180,215] are asymptotically less efficient than the fastest CPU implementations.

On the other hand, the Bellman-Ford's algorithm relies on an iterative process over all edge connections, which updates the vertices continuously until final distances converge. Even though it is less efficient than Dijkstra's ($O(|V||E|)$), it is well suited to parallelization [56].

In the context of parallel implementations for GPUs, where the energy and power consumption is becoming a constraint in addition to performance [128], an ideal solution to SSSP would provide both the performance of the Bellman-Ford's and the work efficiency of the Dijkstra's algorithms. Some work has been recently done to analyse the spectrum between massive parallelism and efficiency, and different parallel solutions for GPUs have been proposed to implement parallel-friendly and work-efficient methods to solve SSSP [82]. Experimental results confirmed that these trade-off methods provide a fair speedup by doing much less work than traditional Bellman-Ford methods while adding only a modest amount of extra work over serial methods.

On the other hand, all these solutions as well as Dijkstra's implementations, do not work in graphs with negative weights [74]. Indeed, the Bellman-Ford algorithm is the only solution that can be also applied in application domains where the modeled data maps on graphs with negative weights, such as, power allocation in wireless sensor networks [229,291], systems biology [146], and regenerative braking energy for railway vehicles [168].

In addition, the most recent GPU architectures (e.g., NVIDIA Kepler GK110 [13] and Maxwell [204]), not only offer much higher processing power than the prior GPU generations, but, also, they provide new programming capability that allows improving the efficiency of the parallel implementations.

This Section presents *H-BF*, a high-performance implementation of the Bellman-Ford algorithm for GPUs, which exploits the more advanced features of GPU architectures to improve the execution speedup with respect to any implementation at the state of the art for solving the SSSP problem. In particular, *H-BF* implements a parallel version of the Bellman-Ford algorithm based on *frontiers* [74] and *active vertices* [237] with the aim of optimizing, besides the performance, the algorithm work efficiency. The Section presents different optimizations implemented in *H-BF*, which are oriented both to the algorithm and to the architecture to underline the synergy of parallelism in the algorithm, programming and architecture.

Experimental results have been conducted on graphs of different sizes and characteristics to compare, firstly in terms of performance and, then, in terms of work efficiency, the *H-BF* implementation (which is available for download in *http://profs.sci.univr.it/~bombieri/H-BF/index.html*) with the most efficient sequential and parallel implementations at the state of the art of both Dijkstra's and Bellman-Ford's algorithms.

The work is organized as follows. Section 8.2 summarizes the Bellman-Ford's algorithm.Sections 8.3 and 8.4 present the optimizations of the proposed approach oriented to the algorithm and GPU architecture, respectively. Section 8.5 reports the experimental results, while Section 8.6 is devoted to concluding remarks.

8.2 The Bellman-Ford's algorithm

Given a graph $G(V, E)$ (directed or undirected), a source vertex s and a weight function $w : E \to \mathbb{R}$, the Bellman-Ford algorithm visits G and finds the shortest path to reach every vertex of V from source s. The pseudocode of the original sequential algorithm is the following:

Algorithm 6 Bellman-Ford's algorithm

```
for all vertices $u \in V(G)$ do
    $d(u) = \infty$
d(s) = 0
while edges $(u, v) \in E(G)$ do
    Relax $(u, v, w)$
end
```

where the *Relax* procedure of an edge (u, v) with weight w verifies whether, starting from u, it is possible to improve the approximate (*tentative*) distance to v (which we call $d(v)$) found in any previous algorithm iteration. The relax procedure can be summarized as follows:

Algorithm 7 Relax Procedure

```
Relax(u, v, w)
if $d(u) + w < d(v)$ then
    $d(v) = d(u) + w$
```

The algorithm, whose asymptotic time complexity is $O(|V||E|)$, updates the distance value of each vertex continuously until final distances converge.

8.3 The frontier-based algorithm and its optimizations

The complexity of a SSSP algorithm is strictly related to the number of *relax* operations. The Bellman-Ford algorithm performs a number of *relax* operations higher than the Dijkstra or Δ-stepping algorithms while, on the other hand, it has a simple and lightweight management of the data structures. The *relax* operation is the most expensive in the Bellman-Ford algorithm and, in particular, in a parallel implementation, each *relax* involves an atomic instruction for handling race conditions, which takes much more time than a common memory access.

To optimize the number of relax operations, *H-BF* implements the graph visiting by adopting the idea proposed in the sequential queue-based Bellman-Ford of

Sedgewick et al. [237]. Such a sequential algorithm uses a FIFO data structure to keep track of *active vertices*, that is, all and only vertices whose tentative distance has been modified and, thus, that must be considered for the relax procedure at the next iteration. If $d(v)$ does not change during iteration i, there is no need to relax any edge outgoing from v in iteration $i + 1$. As a consequence, v is not inserted in the queue to avoid useless computation.

Differently from Dijkstra's, the queue-based Bellman-Ford's algorithm does not rely on a priority queue and the vertex processing can be performed in any order. The parallel algorithm implemented in *H-BF* exploits the concept of *frontier* [74] rather than FIFO queue to visit the vertices concurrently. Given a graph G and a source vertex s, the parallel algorithm can be summarized as in Algorithm 8.

Algorithm 8 Frontier-based Bellman-Ford's algorithm

```
for all vertices $u \in V(G)$ do
    $d(u) = \infty$
$d(s) = 0$
$F_1 \leftarrow \{s\}$
$F_2 \leftarrow \varnothing$
while $F_1 \neq \varnothing$ do
    Parallel for vertices $u \in F_1$ do
        $u \leftarrow$ DEQUEUE($F_1$)
        Parallel for vertices $v \in adj[u]$ do
            if $d(u) + w < d(v)$ then
                $d(v) = d(u) + w$
                ENQUEUE($F_2, v$)
        end
    end
    SWAP($F_1, F_2$)
end
```

Considering two frontier data structures, F_1 and F_2, at each iteration i of the while loop, the algorithm concurrently extracts the vertices from F_1 and inserts all the *active* neighbors in F_2 for the next iteration step. Each iteration step concludes by swapping the F_2 contents (which will be the actual frontier at the next iteration step) in F_1. Fig. 8.1 shows an example of the basic algorithm iterations starting from vertex "0". For the sake of clarity, the figure only reports the actual frontier (F_1 of Algorithm 8, reported as F in Fig. 8.1) at each iteration step, and D as the corresponding data structure containing the tentative distances. The example shows, for each algorithm iteration, the dequeue of each vertex form the frontier, the corresponding relax operations, i.e., the distance updating for each vertex (if necessary), and the vertex enqueues in the new frontier. In the example, the algorithm converges in a total of 23 relax operations over six iterations.

The parallel frontier-based Bellman-Ford's algorithm (Algorithm 8) preserves the semantics of the original Bellman-Ford's algorithm (Algorithm 6). The only difference between the sequential and parallel algorithms is that the first adopts

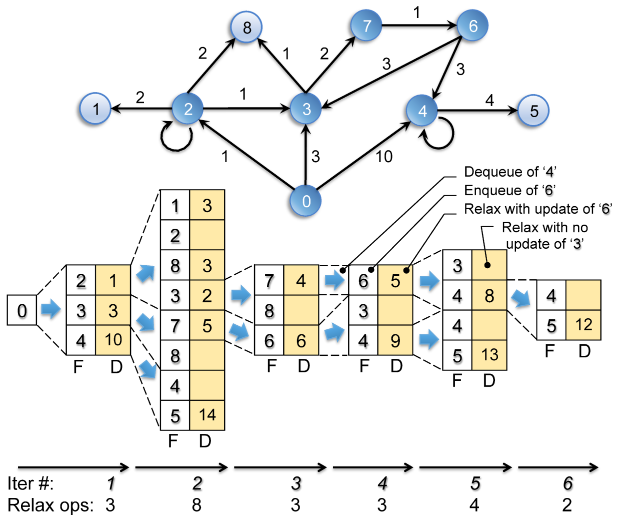


FIG. 8.1: *Example of the basic algorithm iterations starting from vertex "0"*

a *queue structure* in which all nodes are stored and processed sequentially. The second adopts a *frontier structure* (as proposed by Cormen et al. [74]), in which all and only active nodes are processed in parallel. The parallel processing of active nodes does preserve the semantics of the algorithm. This is due to the fact that (i) each node processing is independent from the others, and (ii) including non-active nodes in the processing phase of any propagation step does not change the result (next frontier), as proved by Sedgewick et al. [237] and by Pape [217].

Frontier-based data structures have been similarly applied in literature for implementing parallel breadth-first search (BFS) visits [59, 107]. The main difference from BFS is the number of times a vertex can be inserted in the queue. In BFS, a vertex can be inserted in such a queue only once, while, in the proposed Bellman-Ford implementation, a vertex can be inserted $O(|E|)$ times in the worst case.

H-BF implements three main optimizations to improve both the performance and the work efficiency:

1. The *edge classification*. During each frontier propagation step, the edge outgoing the vertices of the frontier are classified and processed differently to simplify as much as possible the *relax* operations, as explained in Section 8.3.1.
2. The *duplicate removal*. It allows avoiding redundancy when processing duplicate vertices in the frontier propagation steps, as explained in Section 8.3.2.

8.3.1 The edge classification optimization

During the graph visit, the number of relax operations can be significantly reduced by observing the properties of the edges outgoing the active vertices in the frontier. In particular, given a vertex u and an outgoing edge $(u, adj[u])$, we identify four different classes to which the edge may belong. Depending on the class, the edge may involve operations lighter than the standard *relax*, with a consequent impact on performance:

Algorithm 9 EDGE CLASSIFICATION OPTIMIZATION

```
Relax_Opt(u, v, w)
if $u = v$ OR out-degree(v) = 0 then
    skip
else if $u = s$ OR in-degree(v) = 1 then
    $d(v) = d(u) + w$

else
        AtomicMin($d(v)$, $d(u) + w$)
end
```

1. *Self-loop class* ($(u, adj[u])$ edges where $u = adj[u]$). Since a self-loop cannot change the tentative distance of u, the relax operation can be avoided (see, for example, edges $(2, 2)$ and $(4, 4)$ in Figure 8.2). The efficiency improvement provided by the self-loop identification is proportional to the number of self-loops in the graph. It best applies to graphs where each vertex includes a self-loop (e.g., *msdoor* and *circuit* in the experimental results).
2. *Source edge class* ($(u, adj[u])$ edges where $u = s$). The relax operation for the source vertex is substituted with a direct update of the tentative distance for each source neighbour ($v \in adj[s]$) since, certainly, they have not been set previously. This optimization best applies to graphs with small diameter and, even more, when the source in such graphs is a high-degree vertex.
3. *In-degree edge class* ($(u, adj[u])$ edges where in-degree of $adj[u]$ is equal to 1). The vertices with in-degree equal to one (e.g., $(0, 2)$, $(2, 1)$, $(4, 5)$, $(7, 6)$, and $(3, 7)$ in Figure 8.2) are never visited concurrently and, thus, the atomic operations are avoided and replaced with a direct distance update.
4. *Out-degree edge class* ($(u, adj[u])$ edges where out-degree of $adj[u]$ is equal to 0). The vertices with out-degree equal to zero in directed graphs (e.g., 1, 5, and 8 in Figure 8.2) and equal to one in undirected graphs are ignored during the algorithm iterations (the relax operation and the enqueue into the next frontier are skipped). The correct distance is assigned at the end of the algorithm iterations without using atomic instructions. *H-BF* implements this optimization through an extra kernel, which is invoked after the main algorithm procedure. Such a kernel involves a negligible amount of computational work with respect to the (useless) relax operations performed for these edges in the standard approach.

Algorithm 9 summarizes the main important steps of the *edge classification*, while Figure 8.2 shows such an optimization applied to the example of Figure 8.1, by underlining how it reduces the number of relax operations of about five times with respect to the standard approach. The example in Figure 8.2 converges in a total of 5 relax operations over five iterations.

The edge classification optimization has been implemented, in *H-BF*, through a *marking* phase, in which each edge is classified by two bits. The marking bits are added to the bits encoding the edge. In particular, in the adopted adjacency list data structure, each edge is encoded with the *id* of the target vertex. In common

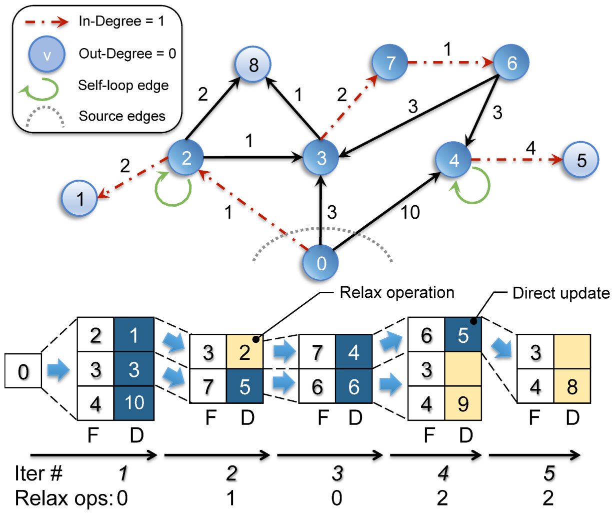


FIG. 8.2: *Example of Edge-Classification optimization*

GPU architectures, where the global DRAM memory is on the order of 4GBytes, such an *id* may require at most 30 bits, since, considering 4Bytes-sized *ids*, 2^{30} is the maximum number of vertices that we can handle[1]. The two most significant bits in a 32 integer id of a vertex, which are, thus, always available, are exploited for such a classification. Reading those bits to identify the edge class does not involve overhead since it is included in the memory access for reading the vertex *id*.

8.3.2 Duplicate removal with Kepler 64-bit atomic instructions

In the execution of a parallel Bellman-Ford implementation, *duplicate* vertices are generated when, during an algorithm iteration, more threads concurrently access the same vertex for the relax operation. This causes a vertex to be redundantly considered for relax and enqueued more times in the next frontier. Figure 8.3 shows an example. Initially, the frontier queue consists of vertices 1, 2, and 3. In the first iteration, the algorithm dequeues the three vertices and performs, in parallel, the relax operation over edges $(1, 4)$, $(2, 4)$, and $(3, 4)$. The memory accesses for updating the tentative distance of 4 are serialized through atomic operations to handle race conditions, and the next frontier is generated by en-queuing duplicates of vertex 4 (see iteration # 1 in the example). In turn, the duplicates are redundantly evaluated for the successive iteration and relax operations. In the example, the duplicate problem of the parallel implementation, by considering the atomic operation order shown in the figure, causes 9 relax operations instead of the three of any serial implementation.

[1] Actually, the available global memory for storing the vertex *ids* is much less since the implementation requires additional data structures for storing frontiers, edges, weights, vertex offsets, and vertex weights.

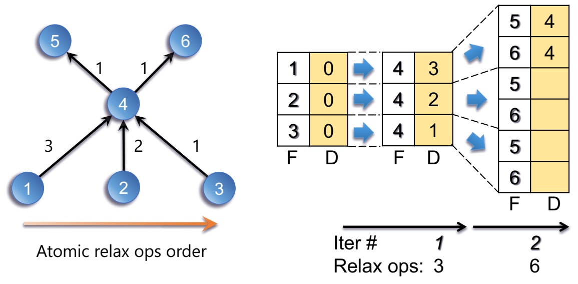


FIG. 8.3: *The duplicate problem in the frontier propagation.*

In literature, a technique to detect and remove duplicates has been proposed by Davidson et al [107]. Such a technique allows eliminating duplicates by interleaving the main computation kernel with two additional kernels. The first aims at marking every frontier vertex through a lookup table in global memory, while the second aims at accessing the look up table to check whether the vertex index exists before proceeding with the relax phase. The *non-duplicate* vertices are compacted before carrying on with a new algorithm iteration. This strategy involves four memory accesses (two of them not coalesced) for each vertex in the frontier, and it introduces overhead for the compacting routine and for synchronizing with the host.

H-BF implements a different technique to completely avoid duplicate vertices during the graph visit by adding information (extra to the distance value) to each vertex. The distance (D) of each vertex is coupled with the number of the current algorithm iteration. The coupled information (`vertexInfo`) is stored into a 64-bit `int2` CUDA datatype, where D resides in the 32 most significant bits while the iteration number in the 32 least significant bits (see Figure 8.4). Access to such a variable is implemented through single-transaction 64-bit atomic operations, which are available for the GPU architectures with compute capability 3.5 onwards (i.e., Kepler, Maxwell GPUs).

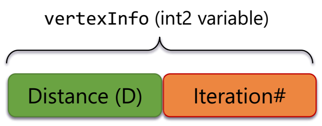


FIG. 8.4: *Use of 64-bit atomic instructions in the implementation of duplicate removal.*

Algorithm 10 represents the high-level implementation of the technique, which highlights how each vertex enqueue is controlled by a condition on the current algorithm iteration. If the iteration number stored in the variable is equal to the actual iteration, the vertex is already in the queue, otherwise it is going to be visited, and thus inserted for the first time in the frontier. Algorithm 11 represents the low-level implementation of such a control, by showing how the atomic primitives provided by Kepler have been applied.

Algorithm 10 ATOMIC64 RELAX PSEUDOCODE

```
Relax_Atom(u, v, w)

if $d(u) + w < d(v)$ then
    $d(v) = d(u) + w$
    if IterationNumber[$v$] $\neq$ currentIterationNumber then
        ENQUEUE($v$)
```

Algorithm 11 ATOMIC64 RELAX IMPLEMENTATION

```
Relax_Atom(u, v, w)

$u$_info = MERGE( $d(u)$, currentIteration )
old_info = ATOMICMIN( &vertexInfo[v], $u$_info )
if ( old_info.iteration $\neq$ currentIteration )
    ENQUEUE($v$)
```

8.4 Architecture-oriented Optimizations

Besides optimizations that target work efficiency (i.e., edge classification and duplicate removal), this Section presents different optimizations that aim at improving the memory accesses bandwidth and the workload balancing.

8.4.1 Memory coalescing, cache modifiers, and texture memory

Coalesced memory access or *memory coalescing* refers to combining multiple memory accesses into a single transaction. When threads of the same warp concurrently access to aligned addresses in global memory, they are coalesced by the device hardware into a single transaction. In the NVIDIA Fermi, Kepler, and Maxwell architectures, the maximum coalescence in memory accesses is achieved when threads of the same warp access 128 Bytes in an contiguous region. The coalescing control in GPUs is hardware-implemented and relies on the use of cache memory. The memory coalescing is the key to reduce the overhead involved by the DRAM memory latency, which, in GPU architectures, is amplified by the thousands of threads accessing such a "slow" memory.

Cache modifiers [203] are a feature provided by Kepler GPU architectures that allows the L1 cache to be enabled or disabled at run time. This allows reducing the miss rate of cache accesses by skipping the cache use for those data not frequently used or too sparse in memory.

In *H-BF*, memory coalescing has been implemented and combined with cache modifiers as follows. Considering each algorithm step (see Section 8.3):

1. The first step (DEQUEUE(F)) aims at reading the frontier vertices, which are stored in global memory in consecutive locations thanks to the use of adjacency list data structures. *H-BF* forces the *cache streaming policy* to take advantage of L1/L2 caches to perform coalesced memory accessed, thus avoiding cache

pollution[2]. The same cache policy is used to load edge offsets of each vertex, which are scattered and occur only one time. In this way, the caches are mainly reserved to the other steps.

2. The threads read information of edges ($v \in adj[u]$), which, similarly to the vertices, are stored in global memory in consecutive locations. Nevertheless, since the edges are much more with respect to the vertices, overloading the L1 cache may decrease the performance. *H-BF* disables the L1 cache for such data while takes advantage of the L2 cache and read-only texture memory (through the `__ldg()` Kepler operators) for caching these accesses.
3. *H-BF* implements the relax phase through atomic instructions, which do not allow exploiting memory coalescing or caching. However, in several cases, the relax operations are replaced by direct updating of the vertex distance $d(v)$ (see class 3 in Section 8.3.1). Memory accesses for such an operation are inevitably scattered and each distance reading occurs only once. *H-BF* directly accesses to the global memory by skipping (and avoiding cache pollution) through low-level PTX instructions [203].
4. The ENQUEUE(F, v) operation performed by each thread consists of updating the frontier data structure in global memory with each vertex v, which information is stored in the thread register. *H-BF* handles such a massively parallel memory writing by stepping into the SM shared memory to organize the data before moving into the global memory. The data organization aims at ordering the data values to enable memory coalescing. Figure 8.5 shows the main idea. The shared memory is partitioned into slots, one per warp. Each thread writes the vertices composing the own partial frontier into the shared memory. The threads write in parallel and start from the shared memory address (offset) computed through a *prefix-sum* procedure [179]. Then, all threads in a warp collaborate to read from the warp slot in shared memory and to perform a coalesced writing in the global memory. In Kepler architectures, the total memory dedicated to registers in each SM exceeds the size of the shared memory. This implies that a warp slot may be used more times for different transactions. Considering, for example, a 48KBytes shared memory and 2048 threads per SM, each slot is 768 Bytes sized (maximum 192 vertices per slot) and allows maximum 6 coalesced transactions to be performed. Then, the slot is released for a new set of data. In general, the thread registers are enough to store the whole frontier. In those particular cases the frontier size exceeds the available registers, the frontier updating in global memory is split in many iterations (register filling, writing in shared memory of the partial frontier, coalesced transaction in global memory, register filling, and so on).

8.4.2 Dynamic virtual warps

Virtual warp programming has been introduced in [125] to address the problem of workload imbalance and thread divergence in GPU graph algorithms. Such a thread scheduling strategy consists of organizing the GPU threads into groups

[2] The L1 cache could be disabled or partially enabled by the compiler to be used for other memory accesses like, for example, the edge reading.

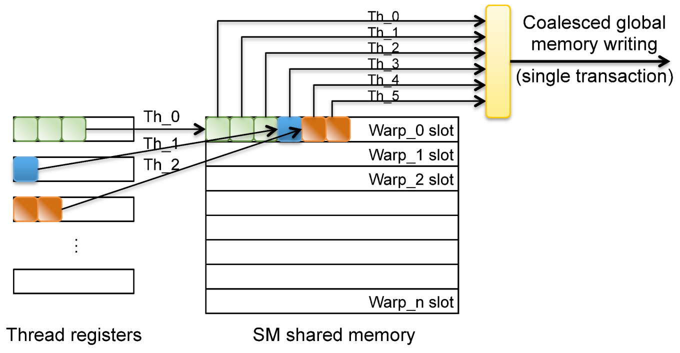


FIG. 8.5: *Example of memory coalescing for the enqueue phase*

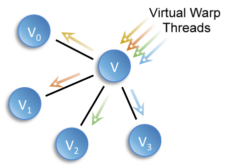


FIG. 8.6: *The Virtual Warp concept*

(i.e., virtual warps) smaller than a warp and whose size is statically tuned. The idea is to assign smaller tasks to few threads (i.e., less than a warp size) to reduce as much as possible the thread divergence. This helps increasing the speedup even when the parallelism degree is low. For example, the graph algorithms exploit the virtual warp strategy to allocate one virtual warp per vertex with the aim of partitioning and equally assigning the work over the edges outgoing the vertex to each thread (Figure 8.6 shows an example) and finds the best application in low-degree graphs.

H-BF exploits the virtual warp technique to increase the thread coalescence during the accesses to the adjacent lists and to reduce their divergence in the frontier propagation steps. In this context, the main limitation of such a technique occurs when the virtual warp size does not properly fit the vertex degree, thus leading to unused threads. In case of vertices with very different degrees over the propagation steps (e.g., power-law graphs), the size choice may be appropriated for some vertices only. Thus, differently from [125], *H-BF* implements a *dynamic* virtual warp, whereby the warp size is calibrated at each frontier propagation step i, as follows:

$$WarpSize_i = nearest\_pow2\left(\frac{\#Res\text{Threads}}{|F_i|}\right) \in [4, 32]$$

where $\#ResThreads$ is the maximum number of resident threads in the GPU device and $nearest\_pow2$ is the lower nearest power of two that rounds the division. $|F_i|$ is the size of the actual frontier.

The virtual warp size may range between 1 and the maximum size of a warp (i.e., 32 for NVIDIA GPUs). Nevertheless, we heuristically found that sizes smaller than 4 threads per warp lead to a decrease of performance due to the excessive non-coalescence (close to a mere serialization) of threads. In addition, the technique proposed in [125] suffers from two problems. First, it overloads the the warp scheduler when the virtual warp size is small and the number of virtual warps is large. Then, it provides workload balancing at warp-level while while, considering that the workload assigned to each virtual warp may be different, it does not provide workload balancing at block level. Indeed, a heavier virtual warp may lead to the situation in which lighter warps of the same block terminate (and thus some SM cores become ready for new warps) but any new block allocation is prevented until the end of all warps.

H-BF overcomes such a problem by assigning more than one vertex per *virtual warp*. The warp scheduler overhead is minimized since there are less thread blocks in the kernel grid and the thread local queues are filled with more items that, in average, are more uniformly distributed. This provides better load balancing and coalesced global memory accesses. We heuristically fixed such a workload to 32 vertices per warp. We found that such a value leads to an increase of performance for all the analysed graphs. Higher values lead to a slight performance improvement only in graph with very high average degree.

8.4.3 Dynamic parallelism

The dynamic virtual warp strategy provides a fair workload balancing when applied to irregular graphs. Nevertheless, to further improve the speedup in case of very irregular graphs (i.e., scale free networks or graphs with power-law distribution), *H-BF* exploits the dynamic parallelism feature of the Kepler architectures. Dynamic parallelism allows implementing recursion in the kernels and, thus, dynamically creating threads and thread blocks at run time without requiring kernel returns. In the *H-BF* context, the idea is to invoke a multi-block kernel properly configured to manage the workload imbalance due to the difference of the vertex degrees. Nevertheless, the (even low) overhead caused by the dynamic kernel stack may elude this feature advantages when replicated for all frontier vertices unconditionally.

H-BF applies dynamic parallelism to a limited number of frontier vertices at each frontier propagation step. Given the degree distribution of the visited graph, *H-BF* applies dynamic parallelism to the sub-set of vertices that have degree far from the average (AVG) and that exceeds a threshold, T_{DP} (Figure 8.7 shows an example).

H-BF combines dynamic parallelism with dynamic virtual warps. The threshold T_{DP} is a knob to be set in *H-BF*, which switches from the use of the former technique to the use of the latter. As explained in the experimental results, we heuristically fixed $T_{DP} = 4096$ vertices for all the analysed graphs.

8.5 Experimental Results

8.5.1 Experimental setup

H-BF has been run on two sets of graphs. The first set is from the 9th and 10th DIMACS implementation challenges [21, 22] and from the University of Florida

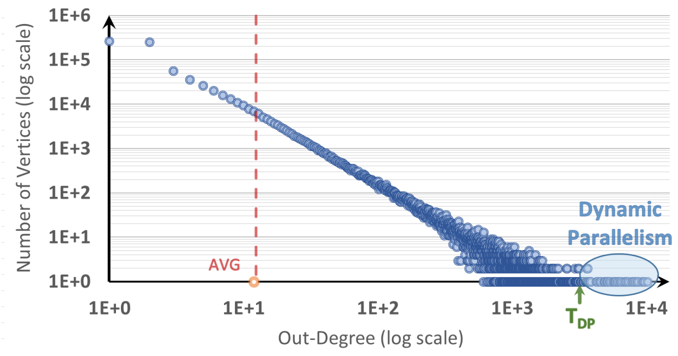


FIG. 8.7: *Example of dynamic parallelism applied to a sub-set of frontier vertices of a power-law graph (flickr)*

Sparse Matrix Collection [85]. It consists of graphs from different contexts such as, road networks, three-dimensional meshes, circuit simulations, social and synthetic graphs. The second set is from SNAP [252], 10th DIMACS, and GTGraph generator [21]. It consists of graphs from contexts like 2D dynamic simulations, communication networks, road networks, autonomous systems, and synthetic based on the Erdős-Rényi model. The graphs of the second set include some edges with negative weights though no negative cycles.

Table 1 shows the graph characteristics in terms of directed/undirected, vertices, edges, average degree, degree standard deviation, maximum degree, graph diameter, and degree distribution of the edges (abscissa) over the vertices (ordinate). The degree distribution, which is shown in log scale, expresses the potential unbalancing of a parallel algorithm to visit the graph. For example, graphs like *msdoor* or *random_0.1Mv.20Me* have the best balancing as they include many vertices with the same (high) degree. In contrast, *rmat.3Mv.20Me* or *wiki-talk* are strongly unbalanced as they include many vertices with low degree, few vertices with high degree and, in general, the degree is not uniform over the vertices.

H-BF has been run on a NVIDIA (Kepler) GEFORCE GTX 780 device, which has 12 SMs, 192 Cores per SM, 3 GB of DRAM, and 5 GHz PCI Express 2.0 x16, with CUDA Toolkit 6.0, AMD Phenom II X6 1055T (3GHz) host processor, and Debian 7 operating system.

8.5.2 Execution time analysis and comparison

Table 2 reports the results in terms of execution time and millions of traversed edges per second (MTEPS) and the comparison of *H-BF* with the most representative SSSP implementations (both sequential and parallel for GPUs) at the state of the art. They include the Boost library sequential Dijkstra [246], which is based on priority queues and relaxed heap, and a queue-based sequential Bellman-Ford. As parallel implementations for GPUs, we selected the Lonestar GPU graph suite [56], which is a parallel implementation of Bellman-Ford, and Workfront Sweep and Near-Far Pile, which are the most efficient parallel implementations of Davidson et al. [82] (see Section 4.2). The results are presented as the average time and the average MTEPS obtained by running the tool from 100 sources randomly chosen, where, for each source, the connected component has at least 10^5 vertices.

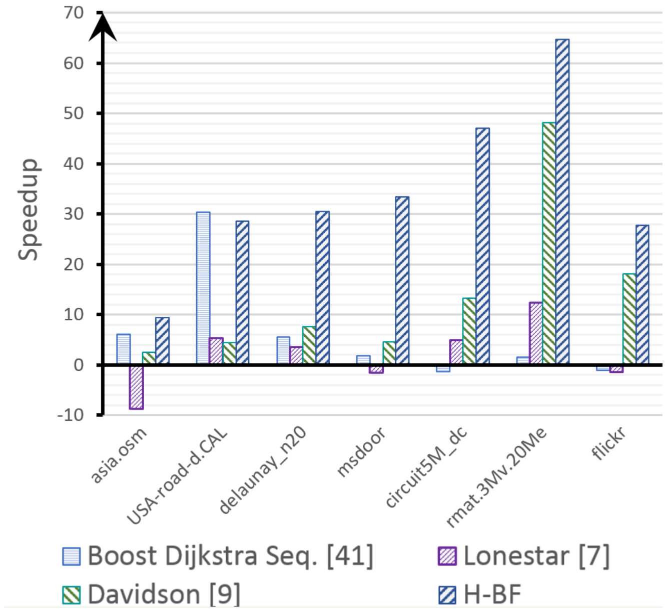


FIG. 8.8: *Comparison of speedups*

Figure 8.8 summarizes the speedup of the different implementations with respect to the sequential queue-based Bellman-Ford implementation. The results show how *H-BF* outperforms all the other implementations in every graph. The speedup on graphs with very high diameter (left-most side of the figure) is quite low for every parallel implementation. This is due to the very low degree of parallelism for propagating the frontier in such graph typology. In these graphs, *H-BF* is the only parallel implementation that outperforms the Boost Dijkstra solution in *asia.osm*, while it preserves comparable performance in *USA-road.d-CAL*. On the other hand, the sequential Boost Dijkstra implementation largely outperforms all the other parallel solutions in literature.

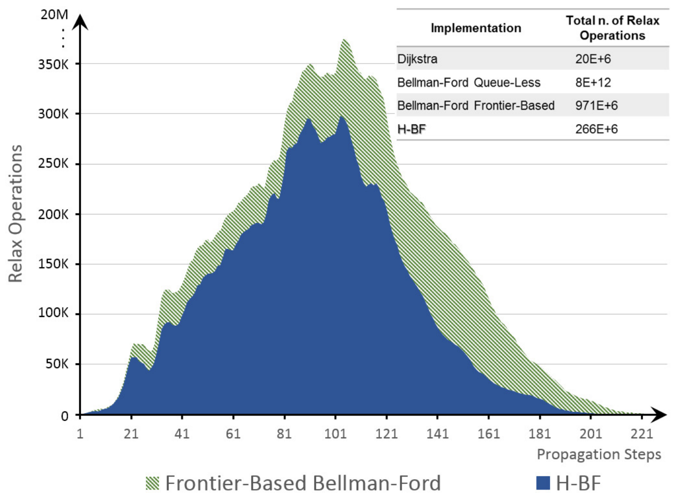
We observed the best *H-BF* performance (time and MTEPS) on the graphs in the right-most side of Figure 8.8 that allow high parallelism due to small diameter and high average degree. *H-BF* provides high speedup also in *rmat.3Mv.20Me* and *flickr*, which are graphs largely unbalanced (see standard deviation and power-law degree distribution in Table 8.1). This underlines the effectiveness of the proposed methods to deal with such an unbalancing problem in traversing graphs. We also verified that the optimization based on the *64-bit atomic* instruction strongly impacts on performance for graphs with small diameters. This is due to the fact that such graph visits are characterized by a rapid grow of the frontier, which implies a high number of duplicate vertices. The *edge classification* technique successfully applies to the majority of the graphs. In particular, *asia.osm* has a high number of vertices with in-degree equal to one, while in *msdoor* and *circuit5M_dc* each vertex has a self-loop. Scale-free graphs (e.g., *rmat.3Mv.20Me* and *flickr*) are generally characterized by a high number of vertices with low out-degree.

The second set contains graphs with negative weights (and no negative cycles) and, thus, the Dijkstra-based sequential implementation as well as the other parallel solutions at the state of the art could not be tested. For these graphs, we compared *H-BF* with respect to the Bellman-Ford sequential implementation and we evaluated the effects of each proposed optimization (see Sections 8.3 and 8.4) on the overall speedup. Table 8.3 reports the results. The basic frontier-based solution provides a speedup that ranges from 12.5x to 20x with respect to the sequential counterpart. The proposed optimizations improve such a speedup from a minimum of twice (from 58.1x to 110.5x by enabling the *duplicate removal*) to almost four times (from 15.4x to 52.6x by enabling the *edge classification*).

We evaluated the impact of the *warp workload* optimization (section 8.4.2) to deal with the lack of parallelism in the *hugetrace_00000* graph, since it represents a 2D dynamic simulation with a low and perfect uniform degree and it is representative to be hardly visited in parallel. The *warp workload* optimization improves the load balancing and the coalesced global memory accesses by filling the local queues with more vertices.

The second graph, *wiki-talk*, is a community network with very low average degree and power-law distribution. The *edge classification* optimization in this graph allows improving the performance by more than three times. The *edge classification* optimization is particularly effective in graphs with power-law distribution since they present a high number of low-degree vertices that, in many cases, have in-degree equal to 1 and out-degree equal to 0. This allows avoiding expensive atomic operations and vertex reinsertions in the frontier. For this graph, we reported both the time spent for the main computation and the additional time to perform the two complementary kernels (in round brackets). With a high maximum degree and the highest standard deviation, the *as-Skitter* graph has the most workload unbalancing. In this case, we underlined the effects of the *Dynamic Virtual Warp* and *Dynamic Parallelism* optimizations. The combination of these techniques allows reaching high throughput with irregular workload, by dealing with both low and high degree vertices. Finally, we considered a random-generated graph with a very low diameter and a high average degree. Traversing any graph with these characteristics leads to a high number of redundant vertices since many threads have high probability to concurrently access the same vertex for the relax operation. In this case, the *duplicate removal* optimization allows improving the performance of twice by avoiding multiple extractions of the same vertex from the frontier.

Finally, Figure 8.9 shows the global effect of the presented optimizations on the *H-BF* work efficiency. The figure reports such an analysis by comparing a Bellman-Ford queue-less (i.e., without frontier) sequential implementation, the basic Bellman-Ford queue-based sequential implementation, the Boost Dijkstra queue-based sequential implementation [246], and *H-BF* in terms of total number of relax operations performed during the SSSP elaboration on the *msdoor* graph (the analysis results are similar for the other graphs). For the sake of clarity, the Boost Dijkstra result is not reported in the figure since it consists of a very long horizontal line (one relax operation for each 20M of edges). As expected, the Di-



Implementation	Total n. of Relax Operations
Dijkstra	20E+6
Bellman-Ford Queue-Less	8E+12
Bellman-Ford Frontier-Based	971E+6
H-BF	266E+6

FIG. 8.9: *Impact of the proposed optimizations on the implementation work efficiency*

jkstra's and Bellman-Ford's queue-less are the most and the least work efficient implementations, respectively. The use of the frontier concept on the Bellman-Ford implementation sensibly reduces the relax operations. *H-BF* further reduces such a work to a final difference of one order of magnitude with respect of Dijkstra's rather than six orders of magnitude of the original Bellman-Ford's queue-less implementation.

8.6 Concluding Remarks

This Section presented *H-BF*, a parallel implementation of the Bellman-Ford algorithm for Kepler GPU architectures. The Section presented different optimizations oriented both to the algorithm and to the architecture, which have been implemented in *H-BF* to improve the performance and, at the same time, to optimize the work inefficiency typical of the Bellman-Ford algorithm. Experimental results have been conducted on graphs of different sizes and characteristics to compare the proposed approach with the most representative sequential and parallel implementations at the state of the art for solving the SSSP problem. Finally, the Section presented an analysis of the impact of the proposed optimization strategies over different graph characteristics to understand how they impact on the *H-BF* work efficiency. An OpenCL implementation of the proposed solution is currently under study. The challenge is to observe how much the performance of the OpenCL and CUDA implementations differ since they provide different low-level instructions as well as the opportunity of implementing different hardware-oriented techniques.

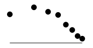
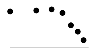
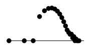
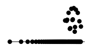
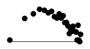
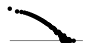
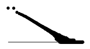
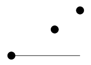
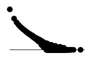
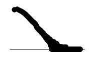
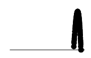
Graph Name	Directed / Undireced	Group	Vertices	Edges	Avg. Degree	Std. Deviation	Max. Degree	Diameter	Degree Distribution
asia.osm	U	Dimacs 10th [22]	12.0M	25.4M	2.1	0.5	9	38,576	
USA-road-d.CAL	D	Dimacs 9th [87]	1.9M	4.7M	2.5	0.9	7	2,575	
delaunay_n20	U	Dimacs 10th [22]	1.9M	6.3M	6.0	1.3	23	380	
msdoor	U	INPRO [85]	415K	20.6M	49.7	11.7	78	167	
circuit5M_dc	D	Freescale [85]	3.5M	19.2M	5.4	2.1	27	135	
rmat.3Mv.20Me	U	GTGraph [21]	3.0M	20.0M	6.7	10.2	521	15	
flickr	D	Gleich [85]	820K	9.8M	12.0	87.7	10,272	12	
Hugetrace_00000	U	Dimacs 10th [22]	4.6M	13.8M	3.0	0.0	3	4,119	
wiki-talk	D	SNAP [252]	2.4M	5.0M	2.1	99.9	100,022	9	
as-Skitter	U	SNAP [252]	1.7M	22.2M	13.1	136.9	35,455	31	
random_2Mv.128Me	U	GTGraph [21]	2.0M	128.0M	64.0	8	114	5	

Table 8.1: Characteristics of the graph datasets on which *H-BF* has been evaluated, including both real and synthetic datasets

Graph Name	Bellman-Ford Queue-Based Seq.		Boost Dijkstra Seq. [246]		LoneStar [56]		WorkFront Sweep / Near-Far Pile [107]		H-BF	
	Time	MTEPS	Time	MTEPS	Time	MTEPS	Time	MTEPS	Time	MTEPS
asia.osm	32.0 s	0.8	5.2 s	4.9	280 s	0.1	12.7 s	2	3.4 s	7.5
USA-road-d.CAL	20.6 s	0.2	588 ms	7.9	3.9 s	1.2	4.6 s	1	720 ms	6.4
delaunay_n20	3.2 s	2.0	581 ms	10.8	902 ms	7.0	420 ms	15	105 ms	60
msdoor	1.2 s	17.2	676 ms	30.6	1.9 s	10.8	206 ms	100	36 ms	570
circuit5M_dc	3.2 s	6.0	4.1 s	4.7	657 ms	29.2	240 ms	80	68 ms	282
rmat.3Mv.20Me	6.4 s	3.1	4.0 s	5.0	520 ms	38.5	133 ms	150	99 ms	201
flickr	887 ms	11.1	963 ms	10.2	1.2 s	8.2	49 ms	200	32 ms	307

Table 8.2: Performance comparison of *H-BF* with the most representative implementations at the state of art.

Graph Name	Optimization	Notes	Belman-Ford Queue-Based Seq.	H-BF w/out Opt.	Speedup w/out Opt. vs. Seq.	H-BF with Opt.	Speedup with Opt. vs. Seq.
hugetrace_00000	Warp Workload	Sparse graph	82.0 s	4.1 s	20.0x	1.4 s	58.6x
wiki-talk	Edge Classification	Sparse graph	1.0 s	65 ms	15.4x	17(+2) ms	52.6x
as-Skitter	Dynamic Parallelism + Dynamic Virtual Warp	High Std. Deviation and Max. Degree	2.5 s	199 ms	12.5x	77 ms	32.5x
random_2Mv.128Me	64-bit Atomic Instr.	Small Diameter	84 s	1,445 ms	58.1x	760 ms	110.5x

TABLE 8.3: Impact of *H-BF* optimizations. Comparison between the speedups versus the sequential implementation obtained by enabling or disabling a specific optimization.

9

Strongly Connected Components - Multi-Step Algorithm

The problem of decomposing a directed graph into strongly connected components (SCCs) is a fundamental graph problem that is inherently present in many scientific and commercial applications. Clearly, there is a strong need for good high-performance, e.g., GPU-accelerated, algorithms to solve it. Unfortunately, among existing GPU-enabled algorithms to solve the problem, there is none that can be considered the best on every graph, disregarding the graph characteristics. Indeed, the choice of the right and most appropriate algorithm to be used is often left to inexperienced users. In this Section, we introduce a novel parametric multi-step scheme to evaluate existing GPU-accelerated algorithms for SCC decomposition in order to alleviate the burden of the choice and to help the user to identify which combination of existing techniques for SCC decomposition would fit an expected use case the most. We support such scheme with an extensive experimental evaluation that dissects correlations between the internal structure of GPU-based algorithms and their performance on various classes of graphs. The measurements confirm that there is no algorithm that would beat all other algorithms in the decomposition on all of the classes of graphs. The contribution thus represents an important step towards an ultimate solution of automatically adjusted scheme for the GPU-accelerated SCC decomposition.

9.1 Introduction

Fundamental graph algorithms such as breadth first search, spanning tree construction, shortest paths, etc., are building blocks to many applications. Sequential implementations of these algorithms become impractical in those application domains where large graphs need to be processed. As a result, parallel algorithms for the processing of large graphs have been devised to efficiently use compute clusters and multi-core architectures. The transformation of a sequential algorithm into a scalable parallel algorithm, however, is not an easy task. Typically, the best sequential algorithm is not necessarily the best parallel algorithm from the practical point of view. This is especially the case of massively parallel graphics processing units (GPUs). These devices contain several hundreds of arithmetic units and can be harnessed to provide tremendous acceleration for many computation intensive scientific applications. The key to effective utilization of GPUs for scientific computing is the design and implementation of data-parallel algorithms that can scale to hundreds of tightly coupled processing units following a single instruction multiple thread (SIMT) model.

Section focuses on the problem of decomposing a directed graph into its strongly connected components (*SCC decomposition*). This problem has many applications leading to very large graphs, including for example web analysis [165], which require high performance processing.

Parallelization of the SCC decomposition is a particularly difficult problem. The reason is that the optimal (i.e., linear) sequential algorithm by Tarjan [261] strongly relies on the depth-first search which is difficult to be computed in parallel. The work in [24] shows how selected nonlinear parallel SCC decomposition algorithms, namely the FORWARD-BACKWARD (FB) algorithm [101,184], the COLORING algorithm [216] and the OBF algorithm [25], can be modified in order to be accelerated on a vector processing SIMT architecture. In particular, we have decomposed the algorithms into primitive data-parallel graph operations and reformulated the recursion present in the algorithms by means of iterative procedures. This approach has been recently improved by warp-wise and block-wise task allocation for primitive graph operations [89, 162]. The authors of [89] have further proposed a SIMT parallelisation of multi-step algorithms by [126, 250] extending the FB algorithm and combining it with the COLORING algorithm.

This Section presents a new parametric multi-step scheme that allows us to compactly define a set of algorithms for SCC graph decomposition as well as a type of the parallelization for individual graph operations. The scheme covers the existing algorithms and techniques mentioned above, but also introduces several new variants of the multi-step algorithm. We use the scheme to carry out an extensive experimental evaluation that helps us to dissect the performance of the individual parametrization on various classes of graphs. The results indicate that there is no single algorithm that would outperform other algorithms on all type of graphs. Moreover, the results show that there is a nontrivial correlation between the parameterization and the performance.

Based on the evaluation we identify, for each type of graphs, the key parameters of the scheme that significantly affect the performance and relate such behavior to the structural properties of the graph. Such analysis is essential for designing an adaptive scheme that would either automatically select an adequate parametrization based on a priori knowledge of the graph structure or automatically switch to a more viable parametrization during the decomposition process.

9.2 Multi-step Parametric Scheme for SCC Decomposition

This section introduces a new multi-step scheme for SCC decomposition, which consists of two levels of parametrization. The first allows setting the individual steps of the algorithm, while the second allows defining the parallelization strategy for the graph traversal.

The multi-step algorithm consists of 3 steps: 1) I_t iterations of the TRIMMING procedure that identifies *trivial components* of the graph (see Section 4.4.1), 2)

Algorithm 12 Parametric Multi-step

```
 1: Input: $G\,(V, E)$, parameters $I_t$ and $I_f$
 2: Ouput: SCC decomposition of $G$
 3: for $i = 1$; $i < I_t \wedge scc \neq V$; $i = i + 1$ do
 4:     OneStepTrimming(G, scc)
 5: for $i = 1$; $i < I_f \wedge V \neq scc$; $i = i + 1$ do
 6:     PivotSelection(G, pivots, ranges)
 7:     FWD-Reach(G, pivots, ranges, visited.f)
 8:     BWD-Reach(G, pivots, ranges, visited.b)
 9:     Update(scc, ranges, visited)
10: while terminate ≠ false do
11:     FWD-MaxColor(G, ranges, colors)
12:     BWD-Reach(G, ranges, colors, visited.b)
13:     Update(ranges, visited.b, colors, visited)
```

I_f iterations of the FB algorithm that aims at identifying *big components*, and 3) the Coloring algorithm that decomposes the rest of the graph. The algorithm parametrization determines the values of I_t and I_f. Algorithm 12 depicts the host code for the GPU-accelerated version of the algorithm.

In the first step (lines 1-2), the kernel oneStepTrimming implements a single iteration of the trimming procedure. It identifies and eliminates vertices of G that form trivial SCCs. It stores the eliminated vertices in the array *scc*. Note that the proposed scheme does not perform the trimming procedure in the later steps of the algorithm, i.e., within every FB iteration as in [24], since the Coloring algorithm handles the remaining trivial components more efficiently.

In the second step (lines 3-7), the algorithm selects a single pivot from the remaining (i.e., not eliminated) part of the graph, it computes the forward and backward closure for such a vertex, and it marks the four subgraphs (see Section 4.4.1) by using the Update kernel. Then, through further iterations of the second step, the algorithm selects multiple pivots and computes multiple closures restricted to the individual subgraphs. The array *ranges* is used to maintain the identification of the subgraphs, while the arrays *visited* indicate the vertices visited during the closure computations. The array *scc* is updated at every iteration to store all vertices that has been already identified in a SCC. For the pivot selection over multiple subgraphs, the algorithm implements the approach proposed in [89] extended to apply the heuristics defined in [250] to favour vertices with a high in-degree and out-degree. The FWD-Reach and BWD-Reach kernels implement parallel BFS visits of the graph, which have been adequately modified for providing reachability results.

The last step implements the coloring algorithm, which is iteratively applied to decompose the remaining subgraphs. The max color is propagated to the successor non-eliminated vertices, and stored in the array *colors* (line 9). The parametric BWD-Reach kernel implements the backward closure to identify a single component for each subgraph. Finally, the updating kernel partitions each subgraph into multiple subgraphs based on the max colors and updates the ranges accordingly for the next iteration.

Note that in the implementation the data associated to vertices in the form of the aforementioned arrays are merged and stored in two 32-bit arrays.

9.2.1 Parallelization strategy for graph traversal

Another dimension of parametrization relates to the way reachability procedures are implemented within the FB and COLORING parts of the Algorithm 12 (lines 5, 6, and 10 respectively).

Recall that when computing the reachability relation (closure), the longest path along which the algorithm has to traverse is given by the diameter of the graph. Assuming that the closure computation consists of multiple kernel calls, where each kernel call shortens this distance by at least one, we immediately have that the diameter of the graph also gives the bound on the number of kernel calls needed. However, there are multiple strategies how to implement such a single kernel call. If the kernel call is guaranteed to shorten the distance only by one, but its complexity itself is linear (e.g. it inspects all vertices/edges), we obtain an overall procedure that computes the closure in a quadratic amount of work in the worst case with respect to the size of the graph.

Alternatively, we may employ a strategy that mimics the serial graph traversal procedure and uses queue of vertices to be processed as the uderlying data structure (the so called frontier queue). In such the case, the complexity of the kernel call is proportional to the amount of vertices processed and the overall complexity of the procedure remains linear. And indeed, when dealing with large graphs, it has been shown that this works the best among various GPU-oriented implementations [57]. On the other hand, the overhead introduced by the maintanance of the frontier queue may render the linear solution inefficient when applied to compute the closure operation on subgraphs with small diameter.

In The parametric scheme we, therefore, allow to specify which strategy should be used to compute the closures in individual phases. In particular, we support the three following options.

1. *Quadratic parallelization (Q).* The closure computation is based on the quadratic parallel breadth-first search as proposed in [117]. It implements the simplest static workload partitioning and vertex-per-thread mapping, thus involving the smallest runtime computation overhead. This strategy works the best for large graphs with regular structure and small diameter.
2. *Quadratic parallelization with Virtual Warps (Q_{VW}).* In this strategy we also employ the quadratic parallel breadth-first search, however, the workload partitioning and mapping rely on the *virtual warps* as proposed in [125]. This modification allows for almost even workload assigned to individual threads, which after all results in reduced branching divergence – an aspect very crucial for the performance of GPU algorithm. Virtual warps also allow improved coalescing of memory accesses since more threads of a virtual warp access to adjacent addresses in the global memory. This strategy is supposed to work the best for graphs with uneven edge distribution.

3. *Linear parallelization (L).* This strategy is the implementation of the linear closure procedure as proposed in [57]. It provides a highly tunable solution that allows efficient handling of very irregular graphs with the overhead of queue maintenance and dynamic load balancing at the runtime. This strategy should work the best for graphs with large diameter and nonuniform edge distribution.

Since the TRIMMING step is typically performed only through a couple of iterations, the strategy used in the TRIMMING kernel rely on a very light thread-per-vertex allocation and the quadratic parallelisation. The overhead of the linear or even more complex approach in this step would never pay off. The very same strategy has also been used for the implementation of the maximal color propagation in the COLORING phase (line 9 of Algorithm 12).

9.3 Experimental results

The experimental results have been run on a dataset of 17 graphs, which have been collected to represent very different structure of the graphs. The dataset covers both synthetic and real-world graphs from different sources and contexts such as social networks, road networks, and recursive graph models. The real-world graphs have been selected from Stanford Network Analysis Platform (SNAP) [252], Koblenz Network Collection [150], and University of Florida Sparse Matrix Collection [85], while the random and R-MAT graphs have been generated by using the GTGraph tool [21].

Table 9.1 summarizes the graph features in terms of number of vertices (in million), edges (in million), average degree, the percentage of vertices with out-degree equal to zero ($d(v) = 0$), out-degree standard deviation, average diameter (over 100 BFS from random sources), number of SCCs, percentage of vertices in the largest SCC, and the percentage of vertices in SCCs with size equal to one.

The table underlines, for instance, that road networks, such as *CA-road* and *USA-road*, present in general a single SCC, a low average degree, and a low number of vertices with $d(v) = 0$. In contrast, social networks (*LiveJournal* and *Flickr*) and the R-MAT model show small-world network properties, which imply one large SCC and a high number of single-vertex SCCs.

We run the experiments on a Linux system (Ubuntu 14.04) with a NVIDIA Kepler Tesla K40 GPU device with 12 GB of memory, CUDA Toolkit 7.5, AMD Phenom II X6 1055T 3GHz host processor, and gcc host compiler v. 4.8.4.

We compared three implementations: a sequential version that implements the Tarjan algorithm [261], which is considered the most efficient sequential algorithm. The data-parallel GPU implementation by Devshatwar et al. [89], the fastest GPU solution at the state of the art, and the proposed approach. Table 9.2 reports the results in terms of runtime (milliseconds) and performance (million of edges per seconds - MTEPS). The results show that the application throughput (MTEPS) of the parallel implementations is directly related to the size and the average diameter of graphs. For instance, *cit-Patents* graph shows a high value of MTEPS due to a low average diameter and a regular degree distribution that allow a high GPU

Graph Name	Vertices	Edges	Avg. Degree	N. of d(v) = 0	Std. Deviation	Avg. Diameter	N. of SCCs	Largest SCC	Trivial SCCs
amazon-2008 [252]	0.7M	5.2M	7.0	12.0%	3.9	25.7	90,660	85%	12%
LiveJournal [252]	4.8M	69.0M	14.2	11.1%	36.1	12.6	971,232	79%	20%
Flickr [150]	2.3M	33.1M	14.4	32.3%	87.7	8.0	277,277	70%	19%
R-MAT [21]	10.0M	120.0M	12.0	20.2%	22.3	7.8	2,083,372	79%	21%
cit-Patents [252]	3.8M	16.5M	4.4	44.6%	7.8	4.2	3,774,768	0%	100%
Random [21]	10.0M	120.0M	12.0	0.0%	3.5	9.0	125	100%	0%
Pokec [252]	1.6M	30.6M	18.8	12.4%	32.1	9.9	325,892	80%	20%
Language [150]	0.4M	1.2M	3.0	0.0%	20.7	33.6	2,456	99%	1%
Baidu [150]	23.9M	58.3M	8.3	22.7%	23.2	12.8	1,503,003	28%	69%
Pre2 [150]	0.7M	6.0M	9.0	0.0%	22.1	60.7	391	100%	0%
CA-road [252]	23.9M	5.5M	2.8	0.3%	1.0	655.9	2,638	100%	0%
web-Berkstan [150]	0.9M	7.6M	2.8	0.7%	16.4	465.6	109,409	49%	15%
SSCA8 [21]	8.4M	99.0M	11.8	0.2%	4.4	1,535.9	55,900	97%	0%
trec-w10g [150]	1.6M	8.0M	5.0	4.4%	72.0	54.8	531,539	29%	31%
Fullchip [85]	3.0M	26.6M	8.9	0.0%	23.1	37.2	35	100%	0%
USA-road [88]	23.9M	58.3M	2.4	0.0%	0.9	6,277.0	1	100%	0%
Wiki-Talk [252]	18.3M	127.3M	9.4	93.8%	80.0	0.4	14,459,546	21%	79%

Table 9.1: Characteristics of the graph dataset.

Graph Name	Sequential SCC		Devshatwar et al. [89]		Proposed implementation	
	Time	MTEPS	Time	MTEPS	Time	MTEPS
amazon-2008	162	32	16	325	17	305
LiveJournal	2,575	26	86	802	87	793
Flickr	821	40	54	611	54	611
R-MAT	9,182	13	193	621	192	625
cit-Patents	536	31	16	1,031	16	1,031
Random	10,619	11	231	519	218	550
Pokec	1,344	23	42	729	33	927
Language	75	16	29	41	22	55
Baidu	582	100	70	832	50	1,166
Pre2	127	47	30	200	19	316
CA-road	223	25	166	33	79	70
web-Berkstan	94	81	1,754	4	717	11
SSCA8	4,237	23.4	1,174	84	465	213
trec-w10g	147	54	12,508	1	2,218	4
Fullchip	547	49	506	53	72	369
USA-road	2,191	27	7,041	8	669	87
Wiki-Talk	5,835	22	18,907	7	731	174

Table 9.2: Runtime (milliseconds) and performance of the three implementations.

utilization. On the other hand, the performance of the sequential version depends on the number of vertices and edges of the graphs.

Table 9.3 presents the configuration of the proposed parametric approach that leads to the best performance and compares such performance to those provided by the "static" solution of Devshatwar et al. The configurations are expressed in terms of which strategy is used in the FB and in the coloring step, i.e., linear (L),

Graph Name	FB Alg.	Coloring Alg.	Trimming steps	FB steps	Speedup vs. Sequential	Speedup vs. Devshatwar et al. [89]
amazon-2008	Q_{VW}/L	Q/L	1	1	9.1x	1.0x
LiveJournal	Q_{VW}	Q/L	1	1	29.5x	1.0x
Flickr	Q_{VW}/L	Q/L	1	1	15.9x	1.0x
R-MAT	Q_{VW}	Q/L	1	1	47.8x	1.0x
cit-Patents	Q/L	Q/L	1	1	34.5x	1.0x
Random	Q_{VW}	Q/L	0	1	48.7x	1.0x
Pokec	Q_{VW}	L	1	1	41.0x	**1.3x**
Language	L	Q	0	2	3.4x	**1.3x**
Baidu	L	L	3	1	11.6x	**1.4x**
Pre2	L	L	0	1	6.8x	**1.6x**
CA-road	L	L	0	1	2.8x	**2.1x**
web-Berkstan	L	L	FULL	17	0.13x	**2.5x**
SSCA8	L	L	0	1	9.1x	**2.5x**
trec-w10g	L	L	2	20	0.07x	**5.7x**
Fullchip	L	L	0	1	7.6x	**7.0x**
USA-road	L	Q	0	1	3.3x	**10.5x**
Wiki-Talk	L	L	5	1	8.0x	**25.9x**

TABLE 9.3: Parametrization results and performance comparison.

static quadratic (Q), and quadratic with virtual warps (Q_{VW}), and the number of iterations of the trimming and FB steps. Notations Q/L or Q_{VW}/L indicate that the two algorithms provide similar performance.

The proposed implementation provides similar performance compared to Devshatwar et al. for the first six graphs of the dataset, while it reports speedup up to 26 times for the other graphs. This is due to the parametric feature of the proposed approach, which allows properly combining the quadratic and linear algorithms and tuning the algorithm iterations for each step i.e. trimming (I_t parameter), forward-backward (I_f parameter), and coloring. In particular, graphs with low average diameter, such as *Flickr, R-MAT, cit-Patents, Random*, show good performance also with the quadratic traversal algorithms due to less overhead compared to the linear approach that maintains frontier data queues.

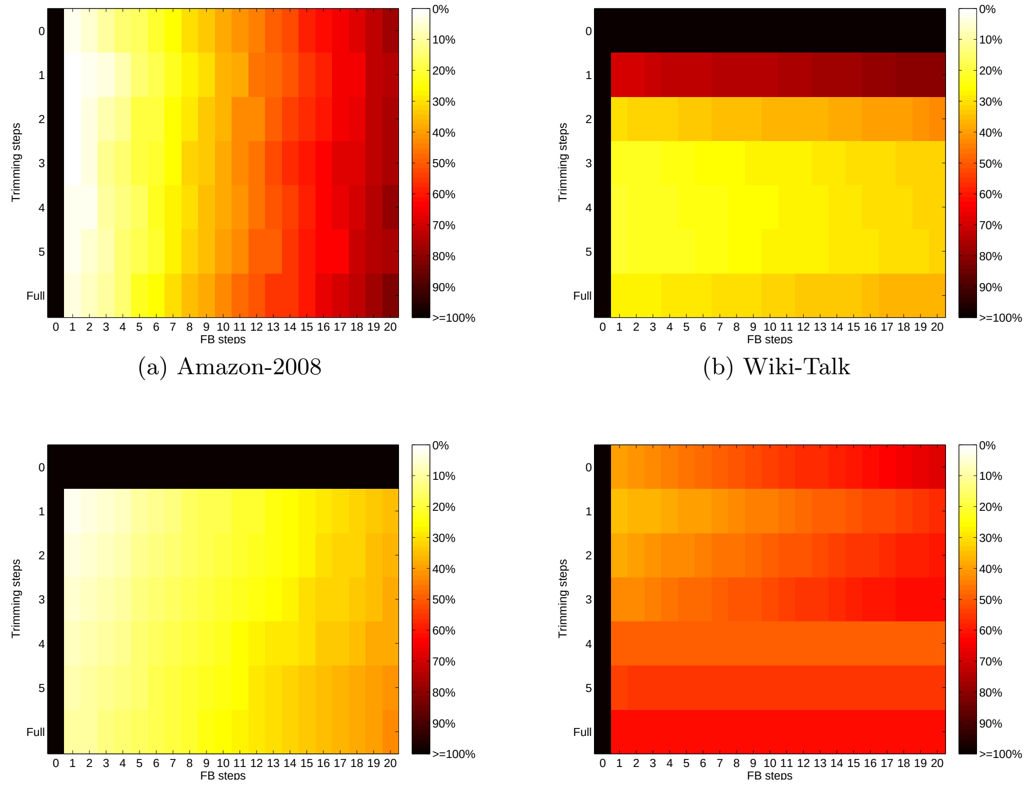
The *LiveJournal* graph presents the same average diameter of *Baidu* but shows different SCC characteristics. *LiveJournal* has a very large SCC and a small percentage of trivial components, while *Baidu* the opposite. In this case, a high number of vertices with out-degree equal to zero (22.7%) favours quadratic parallelization and one iteration of trimming.

The *amazon-2008* graph, even though it has a middle-sized average diameter, shows the best results with the quadratic approach. This is due to its very small size (*amazon-2008* is the second smallest graph in the dataset). The *Language* graph has similar size but it has a high unbalanced out-degree distribution (i.e., standard deviation 20.7 versus 3.9 of *amazon-2008*) and thus the load balancing techniques implemented in the linear BFS outperforms the quadratic parallelisation of the FB algorithm.

The proposed parametric implementation clearly outperforms the static Devshatwar et al. approach on graphs with high average diameter, such as *USA-Road*, and not uniform workload, such as *Wiki-Talk* (std. deviation equal to 80) thanks to the switch to the linear algorithm, which is more efficient in such a kind of graphs.

Finally, both parallel implementations provide poor performance in *Web-Berkstan* and *trec-w10g* graphs due to the lack of data parallelism, which results from the small size, high diameter and low average out-degree.

We can also observe that the trimming step in road networks (*CA-Road* and *USA-Road*), *Pre2*, *Random* and *Fullchip* graphs does not significantly improve the overall performance, since the graphs contain small number of trivial SCCs. The *web-Berkstan* and *trec-w10g* require a high number of FB algorithm steps due to a high number of middle-sized SCCs. For instance, *trec-w10g* graph has the sum of the percentages of the largest SCC (29%) and trivial SCCs (31%) equal to 61% which indicates a remaining of 39% of middle-sized SCCs.



(a) Amazon-2008 (b) Wiki-Talk

(c) Flickr (d) R-MAT

FIG. 9.1: *Performance analysis through parametrization of I_t and I_f*

Figure 9.1 illustrates the impact of the parameters I_t and I_f on the overall performance for the selected graphs. The performance is represented using a color scale – lighter colors denote lower runtime. The linear parallelization evaluated on *Amazon-2008* (Fig. 9.1a) shows that the performance strongly depends on the number of FB iterations (I_f set around 1 gives the best results), while the number of trimming iterations does not affect the execution time. The linear parallelization applied to *Wiki-Talk* (Fig. 9.1b) shows the opposite behavior: I_f has a very low impact on the performance, while setting a wrong I_t (e.g., I_t equal to 1 as in Deshatawar et al.) leads to 60% performance decrease. Such a different behavior of performance over I_t and I_f relies on the different characteristics of the two graphs. *Amazon-2008* has one large SCC and a very small number of trivial SCCs, while *Wiki-Talk* has a high number of trivial SCCs. The performance of the linear parallelization over I_t and I_f on graphs *Flickr* and *R-MAT* shows a more uniform behavior (Fig. 9.1c and 9.1d), since the graphs have one large SCC but also a high number of trivial SCCs.

9.4 Conclusions

We have presented a novel parametric multi-step scheme to evaluate existing GPU-accelerated algorithms for SCC decomposition. The extensive experimental results clearly indicate that there is no algorithm that would be the best for all classes of the graphs. We have dissected correlations between the internal structure of the algorithms and their performance on structurally different graphs. The contribution, thus, represents an important step towards an ultimate solution of automatically adjusted GPU-aware algorithm for SCC decomposition.

10

Approximate Subgraph Isomorphism - APPAGATO

Motivation: Biological network querying is a problem requiring a considerable computational effort to be solved. Given a target and a query network, it aims to find occurrences of the query in the target by considering topological and node similarities (i.e. mismatches between nodes, edges, or node labels). Querying tools that deal with similarities are crucial in biological network analysis since they provide meaningful results also in case of noisy data. In addition, since the size of available networks increases steadily, existing algorithms and tools are becoming unsuitable. This is rising new challenges for the design of more efficient and accurate solutions.
This Section presents *APPAGATO*, a stochastic and parallel algorithm to find approximate occurrences of a query network in biological networks. *APPAGATO*, handles node, edge, and node label mismatches. Thanks to its randomic and parallel nature, it applies to large networks and, compared to existing tools, it provides higher performance as well as statistically significant more accurate results. Tests have been performed on protein-protein interaction networks annotated with synthetic and real gene ontology terms. Case studies have been done by querying protein complexes among different species and tissues.

10.1 Materials and methods

10.1.1 Definitions and notations

A graph G is a pair (V, E), where V is the set of nodes and $E \subseteq (V \times V)$ is the set of edges. If $(u, v) \in E$, we say that v is a neighbour of u. G is *undirected* iff $\forall(u, v) \in E$, then $(v, u) \in E$, i.e. u is a neighbour of v and vice-versa. The degree of a node u, $Deg(u)$, is the number of its neighbours. Given a set of labels A, the function $Lab : V \rightarrow A$ assigns a label to each node of G. We assume that graphs are undirected and labelled only on nodes.

Exact Subgraph Isomorphism

Let $Q = (V, E)$ and $T = (V', E')$ two graphs, named *query* and *target*, respectively. The *exact SubGraph Isomorphism* problem (SubGI) aims to find an injective function, $M : V \to V'$, which maps each node in Q to a unique node in T, such that $\forall(u, v) \in E$: (i) $(M(u), M(v)) \in E'$; (ii) $Lab(u) = Lab(M(u))$; (iii) $Lab(v) = Lab(M(v))$. A solution of the SubGI problem can be represented as the set $m = \{(v_1, M(v_1)), (v_2, M(v_2)), \ldots, (v_{|V|}, M(v_{|V|}))\}$, called a *match* of Q in T. Q may have different maps m_i in T.

Inexact Subgraph Isomorphism and matching costs

In this Section, we deal with the *Inexact SubGraph Isomorphism* problem (ISubGI)[1], which is a variant of the SubGI problem, and in which we admit node and edge mismatches. A mismatch occurs when (i) two nodes with different labels are mapped through a similarity function, or (ii) a query edge or (iii) a query node is missing in the target graph. The absence of a node implies mismatches for all its edges. A cost c is associated to each mismatch. For the sake of simplicity, the same cost $c = 1$ is associated to each of the three types of mismatch.

We denote with $C = \sum c$ the total cost of mismatches between Q and T. The goal of the ISubGI problem is to find an injective function $M : V \to V'$, such that C is minimized. In this case, a solution for the ISubGI, $m = \{(v_1, M(v_1)), (v_2, M(v_2)), \ldots, (v_k, M(v_k))\}$ with $k \leqslant |V|$, is called an *approximate match* with a cost $C \geqslant 0$.

Let $Q_m = (V_m, E_m)$ be the subgraph of query Q that has been mapped in the match m, that is, $V_m = \{v \in V : (v, M(v)) \in m\}$ and $E_m = \{(u, v) \in E : (u, M(u)) \in m\,(v, M(v)) \in m\,(M(u), M(v)) \in E'\}$. We define $V_{\bar{m}} = V \backslash V_m$ and $E_{\bar{m}} = E \backslash E_m$, the nodes and the edges in Q, respectively, that have not been matched in m. Let $S_{|V| \times |V'|}$ be the label similarities between each node $q \in Q$ and $t \in T$. The *label similarity* values belong to the interval $[0, 1]$. The computation of S is application dependent. In the case of PPI networks, the similarity can be based on sequences, functional, or structural protein similarity.

For example, establishing the conservation of a protein-complex CO of the species A within the species B, consists of searching the subgraph Q_{CO}, extracted from the PPI of A (named G_A), into the PPI of B (named G_B). The two PPIs may have different proteins (i.e., nodes with different names), but with similar function, detectable by looking at sequence similarities. An ISubGI algorithm must search for occurrences of Q_{CO} in G_B that minimize sequences and topology differences. We conclude that CO is conserved in B if we find highly similar occurrences.

The total matching cost C is obtained by summing all node and edge costs and by normalizing them over the number of query elements, as follows:

$$C = \frac{\sum_{q \in V_m} (1 - S(q, M(q)) + |V_{\bar{m}}| + |E_{\bar{m}}|)}{|V| + |E|} \tag{10.1}$$

10.1.2 The *APPAGATO* algorithm

The method consists of the following three main phases.

[1] Here called also approximate subgraph querying

Phase 1: Computation of matching probability matrix

Before starting the search, *APPAGATO* computes a matrix P of matching probabilities between all possible node pairs $<q,t>$ ($q \in Q$ and $t \in T$), by combining (i) the label similarity $S(q,t)$, (ii) the degree similarity $D(q,t)$, and (iii) the breadth-first similarity $BFS_{Sim}(q,t)$. The label similarity has been defined in Section 10.1.1. In *APPAGATO*, the label similarity matrix, S, may be provided as input by the user. Alternatively, *APPAGATO* computes a boolean similarity function to compare node labels. It assigns 1 if labels are identical, 0 otherwise. The degree similarity is a binary function D(q,t) = 1 if $Deg(q) \leqslant Deg(t)$, otherwise it is 0. $BFS_{Sim}(q,t)$ is computed by performing breadth-first visits (BFSs) of the query and target graphs by starting from q and t and evaluating label and degree similarities of the visited nodes, level by level. The maximum depth of the BFS visits is a user-defined parameter l_{max}, with $l_{max} \geqslant 1$. Given a node x, and a level $l \leqslant l_{max}$ we denote with $BFS_l(x)$ the set of nodes at level l in the BFS tree rooted at x. An edge $e = (u,v)$ in the BFS tree of q is defined *matchable* iff there exists an edge $e' = (u',v')$ in the BFS tree of t such that $S(u,u')$ and $S(v,v')$ are not 0 and $D(u,u') = D(v,v') = 1$. We denote with $MaxMatch(BFS_l(q), BFS_l(t))$ a maximal set of matchable edges in the BFS tree of q at level l, with respect to the BFS tree of level l rooted in t. The BFS similarity between q and t assumes values in $[0,1]$ and is defined as follows:

$$BFS_{Sim}(q,t) = \frac{\sum_{l=1}^{l_{max}} l \times |MaxMatch(BFS_l(q), BFS_l(t))|}{\sum_{l=1}^{l_{max}} l \times |BFS_l(q)|} \qquad (10.2)$$

Matching probability matrix.

The three similarity values are linearly combined in $MScore(q,t) = S(q,t) + D(q,t) + BFS_{Sim}(q,t)$ and normalized to get the matching probability:

$$P(q,t) = \frac{MScore(q,t)}{\sum_{z \in T} MScore(q,z)} \qquad (10.3)$$

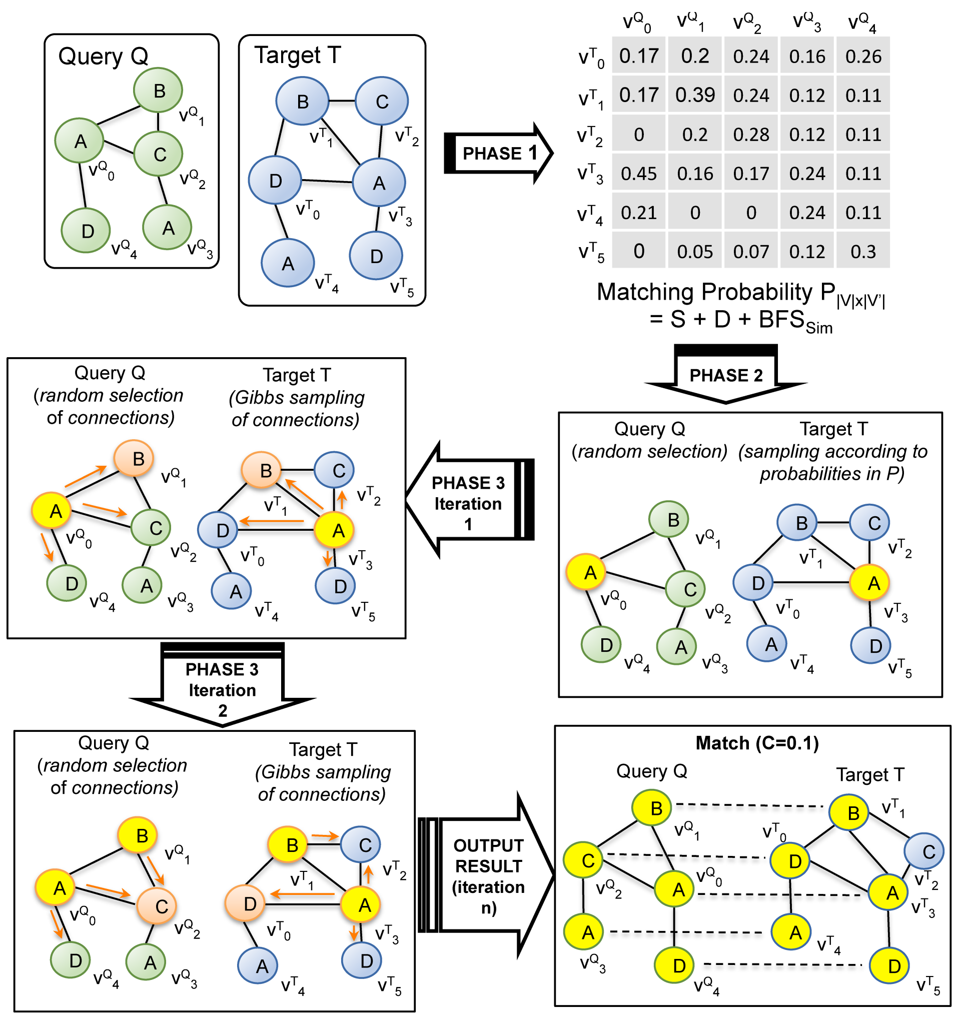
Equation 10.3 ensures that $\sum_{t \in T} P(q,t) = 1$. In phase 2, the probability matrix is used as a transition matrix within an iterative sampling to extract the best possible matches. The upper side of Figure 10.1 shows an example of such a matrix computation.

Phase 2: Seed selection

APPAGATO searches the first pair of nodes to be matched by randomly selecting q and t according to the probabilities defined in Equation 10.3 (see the example of Figure 10.1).

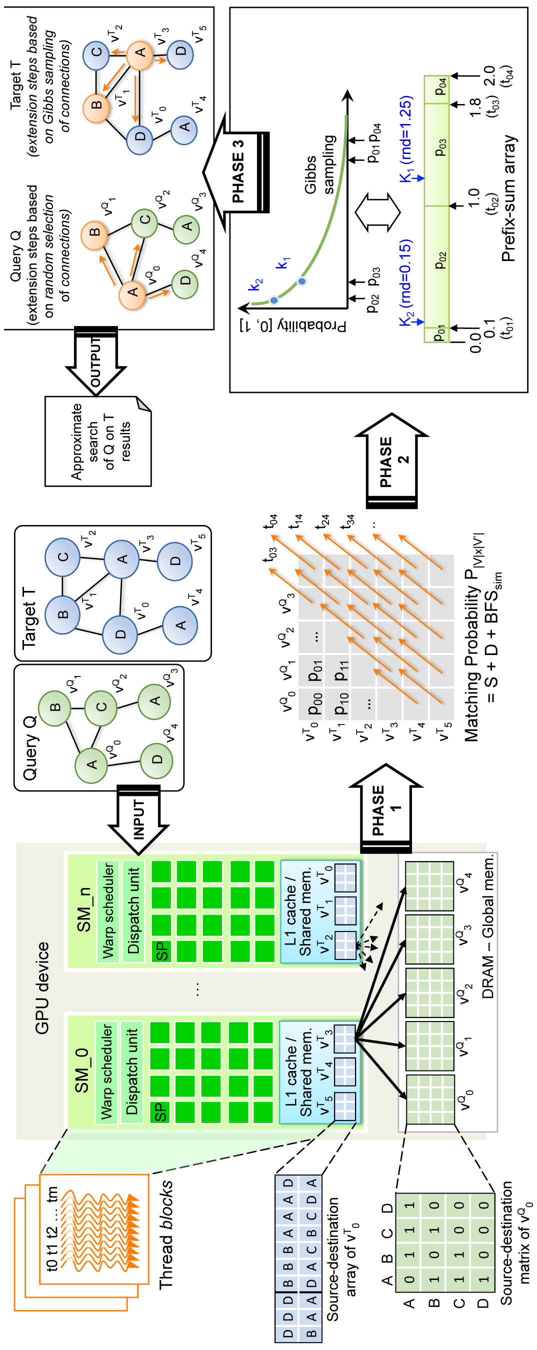
Phase 3: Extension

Gibbs sampling is used to navigate within a Markov chain, where each state represents a possible query-target node match. The initial state corresponds to the


Query Q
Target T
PHASE 1
Matching Probability $P_{|V|x|V'|}$
$= S + D + BFS_{Sim}$
PHASE 2
Query Q (random selection)
Target T (sampling according to probabilities in P)
PHASE 3 Iteration 1
Query Q (random selection of connections)
Target T (Gibbs sampling of connections)
PHASE 3 Iteration 2
OUTPUT RESULT (iteration n)
Match (C=0.1)

	v^Q_0	v^Q_1	v^Q_2	v^Q_3	v^Q_4
v^T_0	0.17	0.2	0.24	0.16	0.26
v^T_1	0.17	0.39	0.24	0.12	0.11
v^T_2	0	0.2	0.28	0.12	0.11
v^T_3	0.45	0.16	0.17	0.24	0.11
v^T_4	0.21	0	0	0.24	0.11
v^T_5	0	0.05	0.07	0.12	0.3

FIG. 10.1: *The APPAGATO approximate matching algorithm.*



GPU device
SM_0
SM_n
Warp scheduler
Dispatch unit
SP
L1 cache / Shared mem.
DRAM – Global mem.
Thread blocks
t0 t1 t2 ... tm
Source-destination array of v^T_0
Source-destination matrix of v^Q_0
INPUT
Query Q
Target T
PHASE 1
Matching Probability $P_{|V|x|V'|}$ = S + D + BFS$_{sim}$
PHASE 2
Gibbs sampling
Probability [0, 1]
K_1 (rnd=1.25)
K_2 (rnd=0.15)
Prefix-sum array
PHASE 3
Query Q (extension steps based on random selection of connections)
Target T (extension steps based on Gibbs sampling of connections)
OUTPUT
Approximate search of Q on T results

FIG. 10.2: *The parallel search of APPAGATO on the GPU device.*

seed selected in phase 2. The sampling method iteratively performs a transition from a state to another, by replacing the query-target nodes pair with a new one, according to a properly defined transition probability. As an example, Figure 10.1 shows the first two iterations of the extension phase. Transition probabilities are defined by starting from similarity scores, and by taking into account the connections of candidate nodes with already matched nodes. Let Q_m and T_m be the set of query-target matched nodes at a certain step of the extension process. We denote with $Q_m[i]$ ($T_m[i]$) the i-th query (target) node added to the partial match. Let q be a query node neighbour to at least one node in Q_m and t be a target node neighbour to at least one node in T_m. We represent the set of connections between q and the nodes in Q_m through a bit vector $CP(q)$ of $|Q_m|$ elements, called *connection profile* of q, where the i-th element is defined as follows:

$$CP(q)[i] = \begin{cases} 1 & \text{if } (q, Q_m[i]) \in E \\ 0 & \text{otherwise} \end{cases} \tag{10.4}$$

We define $CP(t)$ in the same way. The *connection profile similarity* between q and t is the corresponding number of equal bits in the connection profiles of q and t:

$$CP_{Sim}(q,t) = \frac{|\{1 \leqslant i \leqslant |CP(q)| : CP(q)[i] = CP(t)[i]\}|}{|CP(q)|} \tag{10.5}$$

The overall similarity scores is $MScoreExt(q,t) = S(q,t) \times CP_{Sim}(q,t)$. The result value is normalized to obtain the final transition probability[2]:

$$P_T(q,t) = \frac{MScoreExt(q,t)}{\sum_{z \in T} MScoreExt(q,z)} \tag{10.6}$$

After a number of iterations, n, which is a user-defined parameter, the algorithm returns the reached match between the query and the target node. The quality of such a match is evaluated by summing the costs of node and edge mismatches between Q and T. *APPAGATO* does not require any user-defined threshold for the maximum allowed cost of a match. In Figure 10.1, the approximate match has only a label mismatch, v_2^Q whose label C is mapped with v_0^T having label D, and the cost of the match is $C = 0.1$, computed by applying equation 10.1. *APPAGATO* iterates K times phases 2 and 3 and, in each iteration, it starts the sampling procedure from a different seed. Each run of *APPAGATO* always returns K solutions (approximate matches), each one with the corresponding cost.

10.1.3 The *APPAGATO* parallel implementation for GPUs

APPAGATO has been implemented to take advantage of massively parallel GPU architectures. All the processing phases presented in Section 10.1.2 have been implemented through different CUDA *kernels* [3], which are invoked by the *host* CPU.

[2] Notice that *MScore* is not used in the extension phase. *MScoreExt* strongly influences the convergence of the approach [154, 194].

[3] http://www.nvidia.co.uk/object/cuda-parallel-computing-uk.html

This allows performing the most compute-intensive tasks of the search algorithm on the GPU *device*. As for the parallel implementation paradigm for GPUs, each kernel is executed in parallel by several *blocks* of *threads*. Thread blocks spread and run concurrently and independently over *streaming multiprocessors* (SMs). Threads of the same block efficiently cooperate through fast shared memory and by synchronizing their execution through extremely fast (i.e., HW implemented) barriers. Groups of 32 threads of the same block are called *warps*. Each warp executes one kernel instruction at time in parallel on different data (i.e., single instruction multiple data-SIMD architecture) over the many stream processors (cores) of the GPU device. A warp scheduler efficiently switches between warps with the aim of hiding the latency of thread accesses to the memory.

Given the query and the target graphs, Q and T, the three phases have been implemented as follows (see Figure 10.2).

Phase 1: Parallel computation of matching probability matrix.

Computing the matching probability matrix is one of the most computation-intensive part of the whole algorithm. It requires $|V| \times |V'|$ computations of Equation 10.3 and, in particular, $O(|V|+|V'|)$ BFSs over Q and T and the corresponding comparisons between the visited edges (Equation (10.2)).

APPAGATO implements such a phase through a customized version of *BFS-4K* [59], a parallel implementation of BFS for GPU architectures. *BFS-4K* relies on the concept of *frontier* [74] (i.e., a FIFO queue that contains the nodes to be visited at each BFS iteration) to implement the graph visit. Through the frontier-based visiting, *BFS-4K* allows equation (3) to be performed over two levels of parallelism: Each parallel warp of a block is mapped to each node of the frontier, and, each parallel thread of a warp is mapped to each outgoing edge from a frontier node.

APPAGATO extends the BFS visit over a third level of parallelism, by running a total number of $|V| + |V'|$ independent BFSs in parallel, one for each node of Q and T. This is done by allocating one block of threads per BFS. The block allocation is automatically done at runtime. A total number of $|V|$ thread blocks perform, in parallel, $|V|$ BFSs (of depth l_{max}) for the query graph. The result consists of *source-destination matrices*, one per node, which are stored in the global memory (the left-most side of Figure 10.2 shows an example, assuming $l_{max} = 2$). Each matrix contains information on the labels of such edges visited during the BFS from the node along l_{max} levels. In the example of Figure 10.2, the V_0^Q matrix contains information on the edges of the first level BFS ($A - B$, $A - C$, $A - D$) as well as the edges of the second level BFS ($B - A$, $B - C$, $C - A$, $C - B$, $D - A$).

Similarly, and concurrently, a total number of $|V'|$ thread blocks perform the BFSs for the target graph. The result consists of a set of *source-destination arrays*, one per node, which are stored in the device *shared memory*. This allows an extremely fast memory access for the following comparisons between the generated node structures. The array data structure has been chosen as it allows to represent in a more compact way the source-destination information of T in the limited shared memory. In contrast, the matrix data structure has been chosen as it guarantees a faster access to the source-destination information of Q, to be stored in the larger global memory.

Finally, $|V'|$ thread blocks compare, in parallel, their own source-destination array stored in the local shared memory with all the source-destination matrices in global memory. Such a data structure organization over the GPU memory hierarchy allows the complexity of equation (3) to be reduced from $O(|V| \times |V'|)$ as for the sequential algorithm, to a parallel complexity of $O(1)$. The result of Phase 1 is the matrix $P_{|V|x|V'|}$, which is stored in the device global memory (see center part of Figure 10.2).

Phase 2: Parallel seed selection.

APPAGATO emulates the Gibbs sampling to select the K seeds for the successive extension phase. The emulation relies on two parallel primitives, *prefix-sum* [40, 179] and *weighed random number generation* [4], which are efficiently implemented in the literature for GPUs. Given the similarity value of each query-target node pair p_{xy} of $P_{|V|\times|V'|}$, *APPAGATO* performs the parallel prefix-sum of such values through $|V| \times |V'|$ threads (i.e., one thread per similarity value). The result is a prefix-sum array, in which each element is associated to a thread and the corresponding similarity value. As an example, Figure 10.2 shows the prefix-sum array of four threads, $t_{01}, t_{02}, t_{03}, t_{04}$ having similarity value $0.1, 0.9, 0.8$, and 0.2, respectively. The array elements have been depicted through different sizes to better represent the corresponding similarity values. Then, all the threads generate a random sequence of K values in the interval $[0, \sum p_{xy}]$ (i.e., $[0, 2]$ in the example). The parallel primitive for the random number generation allows the threads to share the generation seed and, as a consequence, to generate the same sequence of random values. This allows the threads to concurrently recognize whether the own boundaries in the prefix-sum array include any randomly generated value. In the example, the sequence of random values $K_1 = 1.25$ and $K_2 = 0.15$ leads to the pair of nodes (v_0^Q, v_3^T) and (v_0^Q, v_2^T) associated to threads t_{03} and t_{02}, respectively, to be selected for the extension phase.

Phase 3: Parallel extension.

The extension phase has been implemented through primitives of BFS, prefix-sum, weighed random number generation over different levels of parallelism. As a first level, the K query-target nodes selected in phase 2 are mapped to thread blocks (i.e., one pair of query-target nodes per block). They are concurrently processed as follows. Given a node pair (e.g., (v_0^Q, v_3^T) in Figure 10.2) the two nodes are processed in parallel by two thread warps (second level of parallelism). The two warps perform a one-step parallel BFS (third level of parallelism) on Q and T, respectively, to visit the neighbour nodes (i.e., candidate *connections*) of v_0^Q and v_3^T. The result is two frontiers of neighbours ($\{v_1^Q, v_2^Q, v_4^Q\}$ and $\{v_0^T, v_1^T, v_2^T, v_5^T\}$ in the example). One step of extension over Q performs through a random selection of a node (connection) from the first frontier (v_1^Q in the example). For such a node, *APPAGATO* generates the connection profile through a one-step parallel BFS. Such a connection profile strongly affects the extension over T, which is performed

[4] `https://developer.nvidia.com/curand`

as follows. Starting from all the nodes of the second frontier, *APPAGATO* (i) runs one step of parallel BFS (one per node), (ii) generates the connection profiles of the visited nodes, and (iii) generates the connection profile similarity of each of such nodes with the connection of Q. Through an emulation of the Gibbs sampling similar to that implemented in phase 2, *APPAGATO* selects the new connection for T. The algorithm iterates over the new pair of nodes (i.e., connection of Q and connection T) for a total number $n = |V|$ iterations.

10.1.4 Datasets

Physical Interaction Networks

We used the PPI networks taken from the STRING v10.0 databases [256] of three species: *Mus musculus*, *Homo sapiens*, and *Danio rerio*. These networks differ significantly in size (number of nodes and edges) and density (i.e, the average number of neighbours per node). For each network, we used up to 250 synthetic labels and gene ontologies annotation downloaded from *BioDbNet* [5]. This yielded 12 different PPIs (i.e., 3 species, each one labelled in 4 different ways). We constructed the queries by randomly extracting sets of 100 connected subgraphs, from each network, by varying the size of the queries up to 128 nodes. In this dataset, the similarities matrix $S_{|V|x|V'|}(q,t) = 1$ if $Lab(q) = Lab(t)$ otherwise is set to 0.

Functional Interaction Networks

The STRING database reports, among two proteins and beside the direct physical interactions used above, indirect functional relations such as structural similarity, similarity between the transcript sequences encoding them, and functional correlations. It gives a score, ranging from 0 (namely no relation is known) to 999, which combines physical and functional (i.e., co-expression data analysis) interactions. We constructed a second dataset by taking into account such a combined score. We extracted 4 PPI networks related to the species *Mus musculus*, *Homo sapiens*, *Danio rerio* and *Saccharomyces cerevisiae*. We fixed the interaction score threshold at 998 to get few but highly functional related interactions within each network. As queries, we used 10 human protein complexes taken from the CORUM database [226]. Since CORUM only reports the set of proteins belonging to a given complex, and not their interactions, we reconstructed the topology of the complex by taking into account the interactions reported in the full STRING database with respect to the *Homo sapiens* species. Finally, we labeled target and query nodes with the protein sequences. We computed the query-target node similarities matrix $S_{|V|x|V'|}$, by making use of *CUDASW* [6], which implements a parallel version for GPUs of the Smith-Waterman algorithm for local alignment of sequences. We normalized the matrix by row in order to set to 1 the maximum similarity of the target and query node. We used this dataset to investigate the biological significance of the results. The approximate subgraph matching algorithms were capable to identify functional conservation of protein complexes among different species. We refer the reader to Section 1 and Tables S1-S2 of the Supplementary for more details.

[5] `http://biodbnet.abcc.ncifcrf.gov`
[6] `http://cudasw.sourceforge.net`

10.2 Results and discussion

We compared *APPAGATO* with *NeMA* [142] and *RESQUE* [231] on both the physical and functional datasets described in Section 2.4. All the tools solve ISubGI by taking into account the query topology. Unless differently specified, with the term *APPAGATO* we refer to its implementation on top of CUDA. In the Supplementary, Section 2, we report details on the *APPAGATO* implementation and tuning of parameters (Fig. S1-S3), we assess the robustness of *APPAGATO* over query construction (Fig. S4-S5) and the efficiency of both sequential and parallel versions of *APPAGATO* (Fig. S6-S7).

10.2.1 Performance

For the physical interaction networks, we report the comparison results only between *APPAGATO* and *NeMA*, since *RESQUE* does not support such a large dataset. Fig. 10.3 shows the average running times of the two tools on the *Danio rerio* network. In the total running time of *NeMa*, we distinguish the target preprocessing and the querying time. Note that *APPAGATO* does not perform any prepocessing step. The results show that *APPAGATO* is at least three times faster than *NeMA* in case of very small queries (i.e., 4, 8, 16 nodes). The performance difference sensibly increases with larger queries. The plots clearly show that the *APPAGATO* running time is almost constant when increasing the query size and the number of labels. We do not report the comparison results on *Mus musculus* and *Homo sapiens* since, in those networks, the running time difference is even more evident (i.e., *NeMa* requires more than 10,000 seconds for the preprocessing phase and more than 6,000 seconds for the execution phase, while *APPAGATO* always requires around two seconds). Fig. S8 in Section 3 of Supplementary reports the details on the *APPAGATO* running time in all the physical interaction networks, by showing its efficiency varying the number of labels, query size, and network size.

Fig. 10.4 reports the comparison of *APPAGATO* with *RESQUE* on the functional interaction networks. For the sake of clarity, we do not include the *NeMa* results in the comparison since in this kind of networks, *RESQUE* outperforms *NeMa*. The performance of *RESQUE* mainly depends on the size of query and target and on the number of possible candidates for each query node. *RESQUE* requires, as an input, a similarity matrix between query and target nodes. Such a matrix can be partially defined and this affects the quality of the results. If the similarity matrix is fully defined, then the algorithm execution becomes infeasible (i.e., *RESQUE* takes hours for a single query run). Therefore, we run several tests by changing the percentage of target nodes that can match to a specific query node. Given a threshold t, we set all entries in the similarity matrix with values less than t to 0 (i.e., making them not possible candidates). We then normalized each row by the row maximum value. We chose the percentages 10%, 5% and 1% to obtain reasonable *RESQUE* running times (i.e., 14, 5, 1 seconds, respectively. *APPAGATO* always requires around 0.69 seconds). The *RESQUE* running time rapidly rises as the t threshold increases. In contrast, the *APPAGATO* running times are always below 1 second.

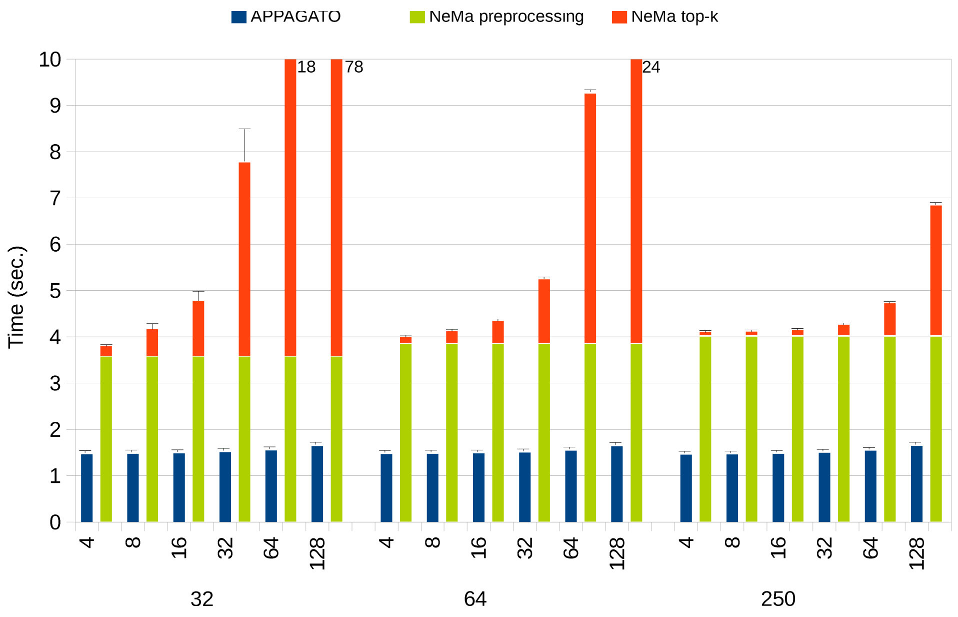


FIG. 10.3: *The running time comparison between APPAGATO and NeMa on the Danio rerio PPI network, randomly labeled with 32, 64 and 250 labels. Chart values report the average time on 100 queries. Queries are grouped with respect to the number of nodes, namely 4, 8, 16, 32, 64, 128. For each query, the tools have been run to find 10, 50 and 100 matches.*

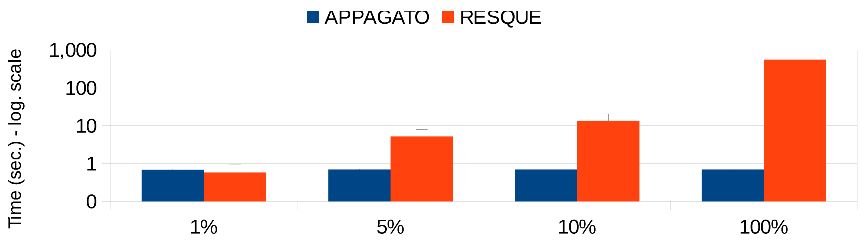


FIG. 10.4: *Running times of APPAGATO and RESQUE on the functional interaction networks. Results are grouped by the similarity thresholds. The running time of RESQUE highly depends on the number of target nodes that can be matched with a query node (i.e., on the similarity threshold t).*

10.2.2 Quality measurements of matches

Fig. 10.5 shows a comparison of the average response costs of *APPAGATO* and *NeMA* on the *Danio rerio* physical PPI network. We removed the duplicated matches from the results of *APPAGATO* to avoid the bias coming from low cost matches. Both algorithms are executed to return the best 10, 50, 100 matches. As expected, both algorithms are highly dependent on the query size. However there is a clear difference in their output quality. The cost of *NeMa* results are often close to 1, which means they involve a high number of mismatches. In contrast, the averages of the *APPAGATO* costs range from 0.1 to 0.55. Fig. S9-S10 in Section 3 of Supplementary confirm the accuracy of *APPAGATO*, also on *Homo sapiens* and *Mus musculus*.

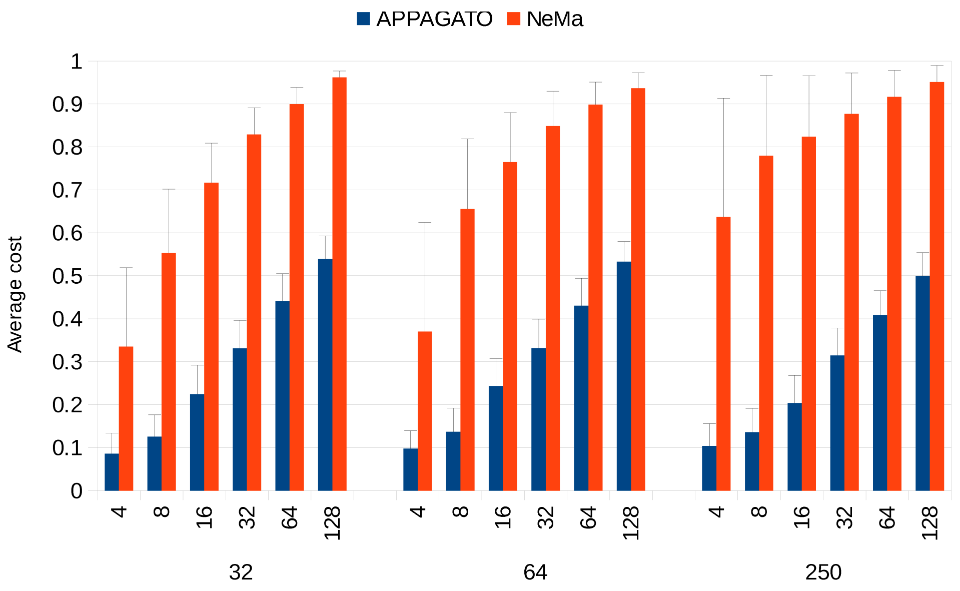


FIG. 10.5: *Average costs (and their standard deviations) by taking into account the set of distinct output matches. Analysis have been performed on the physical interaction PPI of Danio rerio. Results are grouped with respect to the number of target labels and query size.*

We measured the statistical significance of the differences between the *APPAGATO* and *NeMa* performance. We computed the p-values with a Wilcoxon rank-sum test together with a FDR-correction (false discovery rate) for multiple testing. Fig. S11 in Section 3 of Supplementary shows that *APPAGATO* significantly outperforms *NeMa*. The number of tested queries having lower p-values increases as the output size becomes larger, particularly when the number of required output matches increases.

10.2.3 Querying protein complexes among different species.

We compared *APPAGATO* and *RESQUE* using 10 human protein complexes taken from CORUM and queried on the functional interaction dataset composed by *Mus musculus*, *Homo sapiens*, *Drosophila melanogaster* and *Saccharomyces cerevisiae* networks (see Fig. 10.6 and Fig. S12 Supplementary). We test *RESQUE* using two similarity threshold values, 1% and 100%. *RESQUE* shows the main performance limitation with a similarity threshold equal to 1% on every target network, while it provides better performance by increasing the cut-off. In all cases, *APPAGATO* outperforms *RESQUE* even on the quality of the results. To confirm this, we run the Wilcoxons rank-sum tests (see Fig. S13 in Supplementary). For low similarity thresholds (from 1% to 10%), *APPAGATO* provides p-values close to 1×10^{-12}. Better p-values (between 1×10^{-5} and 1×10^{-6}) are shown when we defined the whole similarity matrix. Nevertheless, this turned out to be unfeasible from the running time point of view.

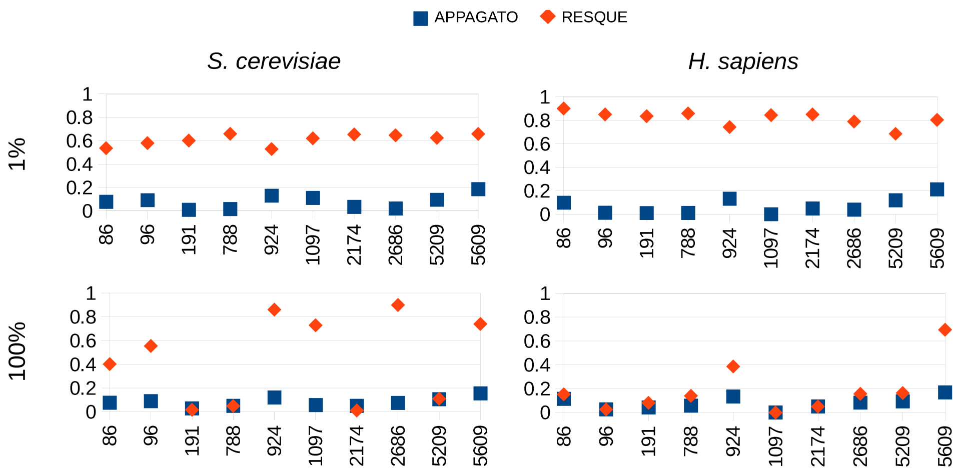


FIG. 10.6: *A chart showing the costs of the 10 protein complexes over the S. cerevisiae and H. Sapiens networks. The CORUM ID of the protein complexes is reported on the x-axis. In the top charts, the similarity threshold is equal to 1%. For those reported in the bottom side the similarity matrix has not been filtered.*

Fig. S14 in Section 4 of Supplementary shows the functional coherence of results with respect to gene ontology. We computed the average p-value for both algorithms obtained by querying the 10 protein complexes for each of the four species. *APPAGATO* outperforms *RESQUE* on every type of target networks and similarity threshold. We refer the reader to Sections 4-5 (Fig. S15-S16-S17) of the Supplementary for details and further application of *APPAGATO* to compare disease modules over tissue specific protein interaction networks.

10.2.4 Datasets and query details

We built two datasets of protein-protein interaction (PPI) networks taken from STRING [256]. Table 1 reports the topology properties of each network.

The physical dataset contains large networks. The network size, which is expressed in terms of number of nodes, ranges from 5,700 to 173,780 edges. In each network, nodes are proteins and edges represent the experimentally validated physical interactions between two proteins. We labeled each network first by randomly assigning 32, 64, and 250 labels through a uniform distribution. Then we used real labeling taken from gene ontologies (GOs) annotations in the following way. We downloaded from BioDbNet [198] the GO biological processes relative to the proteins of the PPI networks. Second, for each species, we built a representative set of the most recurrent GO terms among proteins and we mapped each GO process to the closest representative GO term. We used the shortest path in the GO tree as a distance measure between two GOs. Finally, we assigned to each protein a unique label as the most frequently mapped GO term in the list of GO processes linked to that protein. The process returned up to 43 different labels.

For this dataset, we constructed the queries by randomly extracting sets of 100 connected subgraphs from each network. We varied the query size over 4, 8, 16, 32, 64, and 128 nodes. By construction, the queries are exact subgraphs of the networks. *APPAGATO* returns matches with costs $\geqslant 0$. However, the analysis focuses on matches with costs greater than 0 (i.e., it does not take into account exact matches, whose number is negligible, and which are of less interest w.r.t. the approximate ones). This dataset was inspired by the benchmark used for the contest on graph matching algorithms for pattern search in biological databases (http://biograph2014.unisa.it/). We used this dataset to assess the algorithm performance in terms of running time and to show that the existing tools, except *APPAGATO*,, do no scale on such large biological networks. That is, considering the compared systems, only NeMa [142] runs in reasonable time on the smallest specie, i.e., *Danio rerio.*

To compare *APPAGATO* with RESQUE [231], we created functional PPI networks of smaller size (compared to the physical interaction datasets, see Table 2. We constructed the functional interaction networks in the following way. Beside the physical interactions between proteins, PPI networks have been constructed by using: (i) the structural similarity; (ii) the similarity between the transcript (or gene) sequences encoding them; and (iii) the functional correlations (activation, catalysis, inhibition and so on). The STRING dataset reports a score, ranging from 0 (namely no relation is known) to 999, which combines all above interactions. We set the threshold to 998 to get few but highly functional related interactions. The network size, which is expressed in terms on number of nodes, varies from 1,920 to 3,131, while there are up to 17,560 edges. The query size, which is expressed in number of nodes, ranges from 9 to 23. They are protein complexes taken from CORUM [226].

10.3 *APPAGATO* Implementation

APPAGATO, has been developed on top of the CUDA-C++ Toolkit 7.0 framework. The experimental results have been run on a personal computer with AMD Phenom II X6 1055T (3Ghz) CPU, 8 GB of RAM, NVIDIA GTX 780 GPU device, and Debian 7 operating system. *APPAGATO* takes as input the maximum depth of the BFS visits, l_{max}, which has been set to 2 in all experiments. *APPAGATO* takes as input queries with maximum 254 nodes and $65,536$ edges, and

Physical interactions	nodes	edges	avg. degree
Homo sapiens	12,575	173,780	27.63 (52.06)
Mus musculus	9,781	104,322	21.33 (39.48)
Danio rerio	5,720	51,464	17.99 (31.60)
Functional interactions			
Homo sapiens	3,131	17,560	5.60 (10.98)
Saccharomyces cerevisiae	3,214	41,230	12.82 (26.91)
Mus musculus	1,572	8,708	5.53 (11.99)
Drosophila melanogaster	1,920	14,242	7.41 (16.16)

TABLE 10.1: Topology properties (number of nodes, number of edges, and average degree) of the target networks, for both physical and functional interaction types.

CORUM ID	nodes	edges	name
86	10	72	Nucleosomal methylation activator complex
96	9	72	Anaphase-promoting complex
191	14	182	20S proteasome
788	10	90	Exosome
924	7	40	Toposome
1097	13	156	eIF3 complex
2174	9	72	TRAF6 oligomer complex
2686	13	140	BRCA1-core RNA polymerase II complex
5209	6	26	Ubiquilin-proteasome complex
5609	9	32	Emerin regulatory complex

TABLE 10.2: Queries of the CORUM complexes.

target graphs having maximum 65,536 nodes and 2^{32} edges. *APPAGATO* , run with in input the similarity matrix, does not have constrains on the number of labels, otherwise, the number of labels is fixed to 256. The extension phase of *APPAGATO* in Section 10.1.2 describes the standard application of Gibbs sampling, which iterates the extension a sufficiently large number (n) times. That is, the sampling replaces the target node with another one, according to a properly defined transition probability, and this is iterated n times. The sampling procedure in *APPAGATO* is inspired by the procedure that finds the local alignment of biological networks presented in [194]. n should be a user parameter, but since *APPAGATO* is implemented in parallel and K seeds are processed, *APPAGATO*, sets $n = |V|$, where $|V|$ is the number of nodes in the query. *APPAGATO* is also released as a sequential version, to be run on CPU-based architectures with no GPU accelerators. This allows us to understand how much the *APPAGATO* efficiency is due to the algorithm and how much it is due to the parallel implementation. The sequential implementation has been developed in C++11.

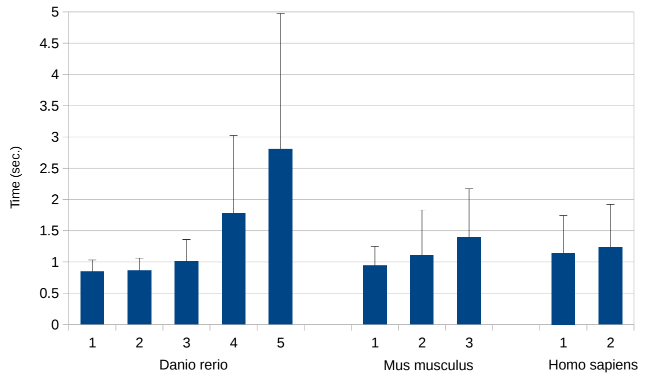


FIG. 10.7: *Running times (and their standard deviation) of APPAGATO results over the physical PPI networks obtained by varying the seed BFS depth l_{max} from* 1 *to* 5*.*

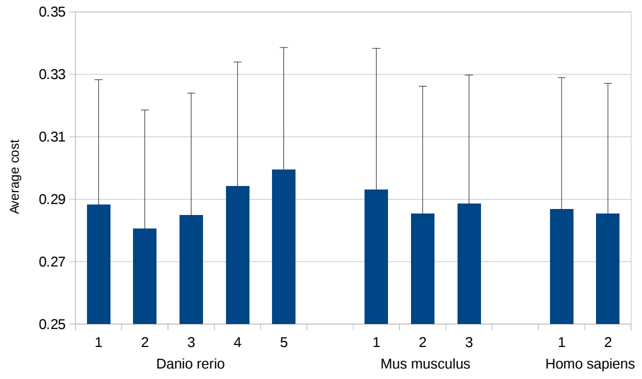


FIG. 10.8: *Average costs (and their standard deviation) of APPAGATO results over the physical PPI networks obtained by varying the BFS depth l_{max} from* 1 *to* 5*.*

Computing the matching probability matrix. *APPAGATO* computes a matrix P of matching probabilities between all possible query-target node pairs based also on the similarity of the neighbors of the two nodes (see Section 10.1.2). This section explains the rationality behind several choices made by *APPAGATO* to compute P.

Default value of l_{max}.

In order to compute the matrix P, *APPAGATO* performs breadth-first visits (BFSs) of target e query nodes. The maximum depth of BFS visit, l_{max}, is, per default, set to 2. This value has been empirically chosen by analysing the performance of *APPAGATO* by varying the value of l_{max} from 1 to 5. A BFS depth equal to 1 means taking into account only the adjacent nodes of the starting one.

Figure 10.7 reports the running time obtained by varying l_{max} over the three physical PPI networks. The time grows exponentially as the BFS depth increases.

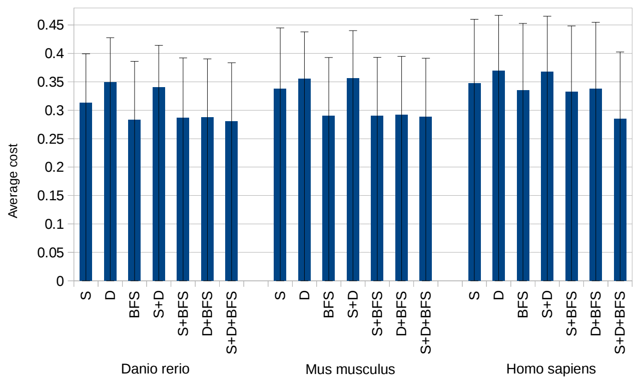


FIG. 10.9: *Average costs (and their standard deviation) obtained by aggregation of the $MScore$ components: label similarity (S), degree similarity (D), and BFS similarity (BFS), over the physical PPI networks.*

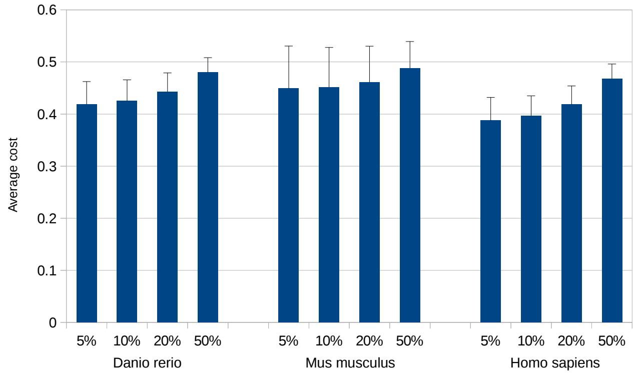


FIG. 10.10: *Average costs of matches obtained by querying a set of perturbed queries over the physical PPI networks. The results are grouped according to the percentage of alterations, which ranges from* 5% *to* 50%.

Due to memory limitations, BFS depth greater than 3 is not applicable on the *Mus musculus* PPI, as well as a depth greater than 2 on the *Homo sapiens* network.

However, we noticed that, the running times for $l_{max} = 2$ do not substantially increase with respect to $l_{max} = 1$ in all datasets. Figure 10.8 shows the solution costs obtained by varying l_{max} over the three physical PPI networks. The lowest costs are achieved with $l_{max} = 2$.

Score contribution.

The P values are based on a linear combination of three distinct scores: label similarity S, degree similarity D, and BFS_{Sim} similarity (see Section 10.1.2). We evaluated the contribution that these three components give to the accuracy of *APPAGATO*,. We analyzed each of them alone and by combining them two by two. Figure 10.9 shows the average costs over the physical PPI networks. If

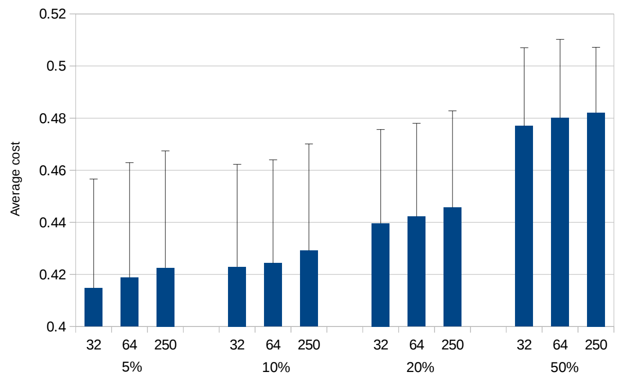


FIG. 10.11: *Average cost of matches by querying a set of perturbed queries over the physical PPI networks of Danio rerio. Firstly, the results are grouped according to the percentage of alterations, which ranges from 5% to 50%. Then they are grouped by the number of distinct labels of the target graph.*

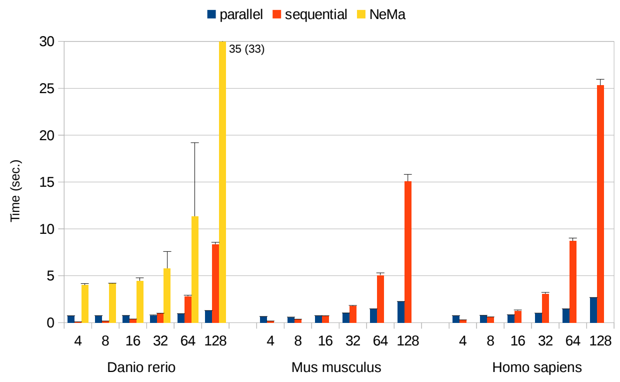


FIG. 10.12: *Running times (and their standard deviation) of NeMa and the sequential and parallel version of APPAGATO over the physical PPI networks. The results are grouped by query dimension.*

the three similarity functions are used alone, the BFS similarity is the one that reaches the lowest costs, and every combination that includes the BFS similarity maintains the same behavior. However, the lowest costs are obtained by combining together the three similarity functions. This becomes more evident by increasing the network size.

***APPAGATO* robustness over query perturbations.** We evaluated how query perturbations can influence the matching costs. We randomly modified the queries constructed in the datasets (as explained in Section 10.1.4) by applying label alterations and edge swapping to a number of nodes, where such a number varies from 5% to 50% of the total nodes (perturbation degree). A node label alteration simply consists of randomly changing the node label. Given an edge connecting two nodes, an edge swapping consists of randomly changing one of the

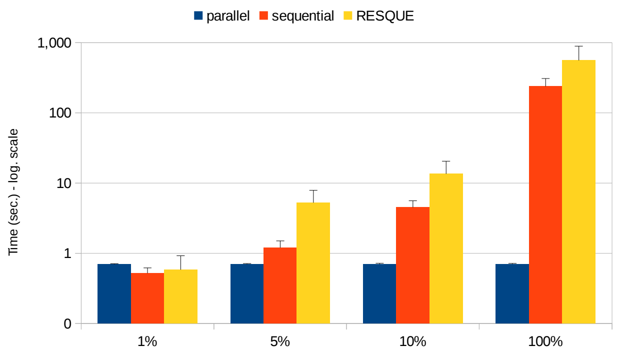


FIG. 10.13: *Running times (and their standard deviation) of RESQUE and the sequential and parallel version of APPAGATO over the functional PPI networks. The results are grouped by similarity threshold.*

two nodes. Figure 10.10 shows the average costs of 100 perturbed queries, over the physical PPI dataset. Figure 10.11 shows the results obtained on the *Danio rerio* network by varying the number of labels of the target nodes. For each perturbed query, we extracted the top 10 matches. Figures 10.10 and 10.11, which have to be compared with Figure 10.5, show that the perturbation degree affects the solution costs non linearly. Any minimal perturbation (e.g., 5%) increments the solution costs around 30% (see for example the average costs of 32-node queries with 32 labels of Figure 10.3 and the *Danio rerio* results of Figures 10.10 and 10.11). Then, any further perturbation (from 5% to 50%) still increases the costs, even though to a lesser extent.

Parallel versus sequential implementation of *APPAGATO*. We created a sequential version of *APPAGATO* in order to understand how much its efficiency is due to the algorithm and how much it is due to the parallel implementation. It is important to note that, beside efficiency, an important feature of the algorithm core is the high degree of potential parallelism. This allows us to parallelize the implementation massively for GPUs. Figures 10.12 and 10.13 show that the sequential version outperforms the parallel one on very small queries. The parallel implementation provides a significant speed-up as soon as the query size increases. The gained speed-up also depends on the size of the target network (see Figure 10.12). The sequential version is faster then the parallel one in case of queries with size up to 16 over the *Danio rerio* network, which is the smallest one in the dataset. However, the performance difference between sequential and parallel in these cases is negligible. In contrast, the parallel version outperforms the sequential one also on queries of size equal or less than 16 in the *Homo sapiens* PPI networks. Similar results have been obtained by taking into account the functional PPI networks. The sequential implementation is slightly more efficient than the parallel one for a similarity threshold equal to 1%, and it loses its (negligible) gain for higher thresholds (see Figure 10.13). The results also show that the sequential version of *APPAGATO* is faster than the two compered methods,

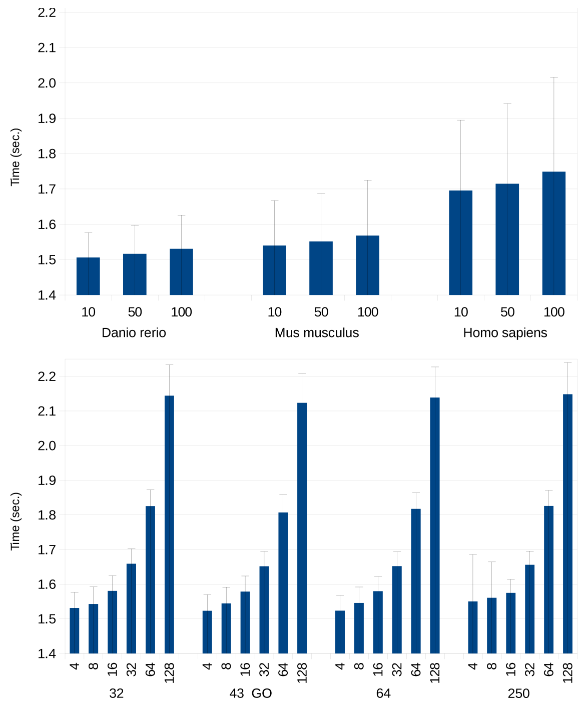


FIG. 10.14: *APPAGATO running times over the physical interaction networks. In the upper side of the figure, the whole set of queries is grouped by species. For each query, 10, 50 and 100 results are given. In the lower side of the figure, the same set of results is grouped according to the number of target labels (from 32 to 250) and number of query nodes (from 4 to 128). The times include the whole software execution (host CPU plus GPU device).*

NeMa and *RESQUE*. The sequential version is released together with parallel one at *http://profs.sci.univr.it/%7Ebombieri/APPAGATO*.

10.3.1 *APPAGATO* performance

Figure 10.14 reports the running time of *APPAGATO* over the physical interaction networks. For each query, *APPAGATO* returns $K = 10, 50, 100$ matches. The upper side of the figure reports the whole set of query results for species. The parallel implementation allows the algorithm to handle larger match sizes (i.e., to set K up to 100) without decreasing the performance in a significant way. On the other hand, by varying K, the overhead due to the loading and downloading data towards the GPU device increases. The performance mainly depends on the size and topology of the target network and on the number of query nodes. The lower side of Figure 10.14 reports and groups the results for each species with respect to the number of label and query size. The figure shows how the performance highly depends on the query size, since it increases the number of extension steps performed by the algorithm. On the other hand, increasing the number of target labels does not cause significant differences.

Due to the randomness of sampling, *APPAGATO*, may return duplicated matches. The upper side of Figure 10.15 shows the ratio between the number of distinct matches and the total number of results. By taking into account both average and standard deviation, the ratio is always above 90%. The chart shows that the percentage decreases as long as the number of target labels increases. The lack of distinct labels reduces the number of matches having a low node mismatch cost. Figure 10.15 (lower side) reports the average query costs of *APPAGATO* over the physical interaction networks. *APPAGATO* maintains relatively low costs even for targets with 250 labels. The cost chart shows a very slight difference between synthetically labeled targets and those annotated with GOs. Such a difference may depend on the distribution of labels (we recall that synthetic labels are uniformly distributed). This may reduce the label variability of target node neighbourhood, thus leading APPAGATO to node mismatches.

Figure 10.16 shows the solution costs provided by *APPAGATO* ordered over query size, grouped per species and per number of labels. Notice that the costs do not increase by increasing the network density or the number of labels. The costs vary by changing the size of the query graph. This result is highly related to the number of exact matches that exist within a target network and by varying the query size [51], since the probability of finding an exact match decreases as the size of the query increases. Figure 10.18 shows that *APPAGATO* outperforms *RESQUE* on the term of low query costs obtained in *Mus musculus* and *Danio rerio* with similarity threshold t per RESQUE set to 1% and 100%.

Figures 10.17 and 10.19 report the p-values with a Wilcoxon rank-sum test together with a FDR-correction (false discovery rate) for multiple testing, thus showing that *APPAGATO*significantly outperforms *NeMa* and *RESQUE*. Finally, Figure 10.20 shows the functional coherence of the results with respect to gene ontology. We computed the average p-value for both algorithms obtained by querying the 10 protein complexes for each of the four species. *APPAGATO* outperforms *RESQUE* on every type of target networks and similarity threshold.

10.4 Functional coherence measurement in querying protein complexes among different species.

We computed the functional coherence of *APPAGATO* and *RESQUE* results with respect to GO. The average p-value for both algorithms have been obtained by querying the 10 protein complexes for each of the four species. We adapted the algorithm used in *GO::TermFinder* [52] to work with proteins (see Fig. 10.20). Protein annotations have been retrieved through the BioStar framework (www.biostars.org). We used the three GO domains: Cellular components, molecular function, and biological process. Then, we extracted the complete DAG (directed acyclic graph) of GOs. The functional coherence determines whether any GO term annotates a specific list of proteins at a frequency greater than that we

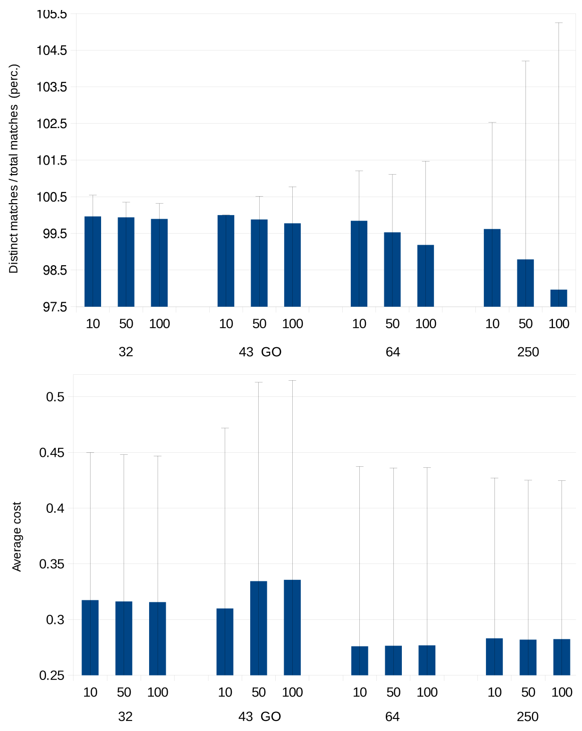


FIG. 10.15: *APPAGATO average and standard deviation costs over Homo sapiens and Mus musculus physical interaction networks grouped by number of labels. For each interaction network, the complete set of 600 queries (100 for each query size) has been tested by requiring 10, 50 and 100 APPAGATO matches.*

expected by chance. Given a match, we extracted the set of GOs that are associated with any of the protein reported in the match. For each extracted GO, we calculated a p-value by making use of the hypergeometric distribution:

$$p = 1 - \sum_{i=0}^{k-1} \frac{\binom{M}{i}\binom{N-M}{n-i}}{\binom{N}{i}} \tag{10.7}$$

where N is the total number of proteins in the background distribution, namely the total number of nodes in the target graph; M is the number of proteins within the whole target graph, which are annotated with the specific GO term, n is the number of matched proteins of the query, and k is the number of query nodes annotated with the specific GO term. The complete set of p-values, one for each GO term, has been corrected with the false discovery rate (FDR). Finally, we calculated the average of such p-values. Fig. 10.20 shows the functional coherence of *APPAGATO* and *RESQUE* results with respect to GO and that *APPAGATO* outperforms *RESQUE* on every type of target networks and similarity threshold.

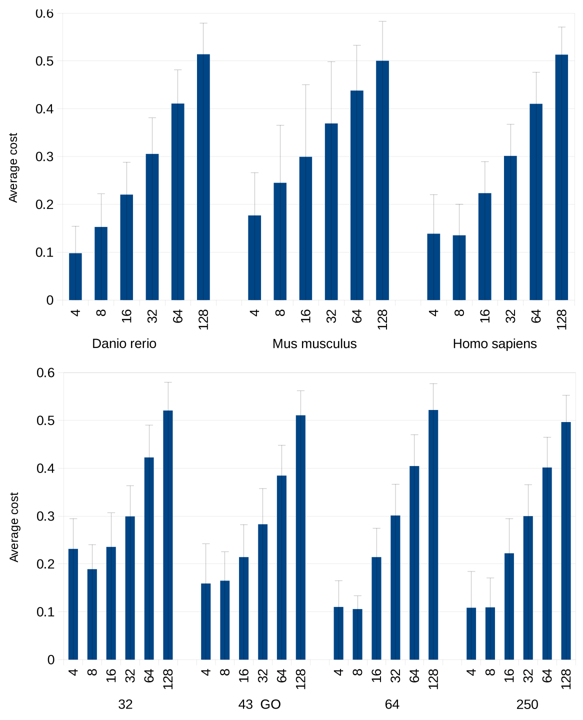


FIG. 10.16: *Average costs and their standard deviation of APPAGATO approximate matches over the Homo sapiens and Mus musculus physical interaction networks. The whole set of queries is firstly grouped (upper side of the figure) by species and query size (number of nodes), and then by the number of labels and query size (lower side of the figure).*

Figure 10.21 shows the results of *APPAGATO* and *RESQUE* on the protein CORUM[7] complex *Ubiquilin-proteasome complex* (CORUM id: 5209). The topology of the protein complex 5209 has been extracted from the *Homo sapiens* PPI and its phylogenetic conservation has been searched in *Saccharomyces cerevisiae*, *Mus musculus* and *Drosophila melanogaster* PPIs. According to the information reported in CORUM, the 5209 protein complex is conserved with a decreasing score from very high, medium to medium-low in *Mus musculus*, *Drosophila melanogaster*, and *Saccharomyces cerevisiae*, respectively. The compared algorithms consider both similarities on nodes (sequence similarity) and topology. We show that an accurate algorithm that maximizes both the above similarities better captures the phylogenetic conservation. Moreover, Figure 10.21 reports that *APPAGATO* is more accurate than *RESQUE*, since according to *APPAGATO*, the protein complex 5209 is much more conserved in *Mus musculus* than in the other two species,

[7] http://mips.helmholtz-muenchen.de/genre/proj/corum/

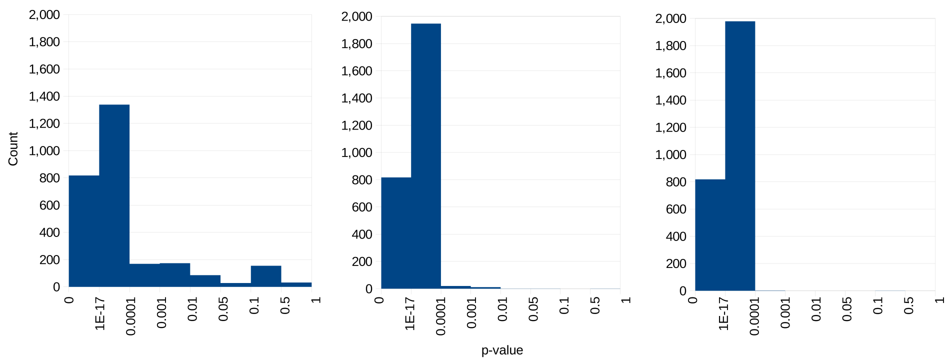


FIG. 10.17: *Statistical significance of match costs. Wilcoxon rank-sum test of APPAGATO, versus* NeMa.

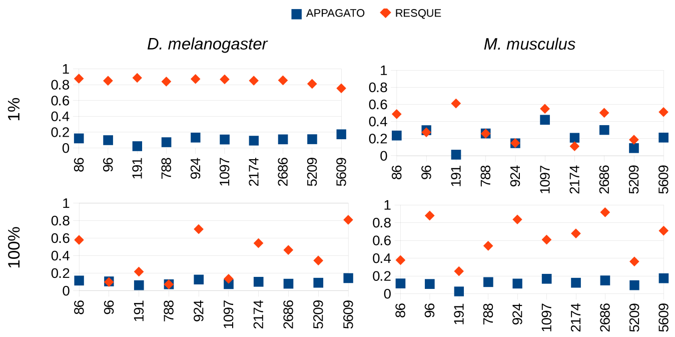


FIG. 10.18: *A chart showing the costs of the 10 protein complexes over the Mus musculus and Drosophila melanogaster networks. The CORUM ID of the protein complexes is reported on the x-axis. In the top charts, the similarity threshold is equal to 1%. For those reported in the bottom side the similarity matrix has not been filtered.*

whereas *RESQUE* gives higher conservation in *Saccharomyces cerevisiae*. By combining topology and node similarities in the query matching helped us to rank the results in *Drosophila melanogaster* closer than the one in *Saccharomyces cerevisiae*. *APPAGATO* maintains similar quality results with the similarity matrix filtered to 1%, 5%, and 10%, while the performance of *RESQUE* drastically decreases. Figure 10.21 shows the results obtained with the filter set to 10%. In this case, *RESQUE* tries to maximize the topology, but it finds in *Mus musculus* another protein complex named *Minichromosome Maintenance Complex*. We also recall that *RESQUE* gains quality and increases running time as the filter on the similarity matrix decreases.

10.4.1 Querying disease modules

Recently, great attention has been given to tissue-specific PPI networks, in which nodes represent proteins whose genes are preferentially expressed in specific tis-

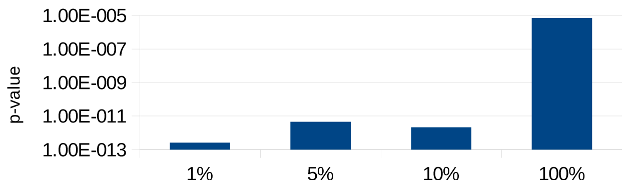


FIG. 10.19: *Wilcoxon rank-sum tests and FDR correction of APPAGATO results with respect to RESQUE, grouped by similarity thresholds (from 1% to 100%) on the functional interaction network. 10 human protein complexes have been queried over all target networks.*

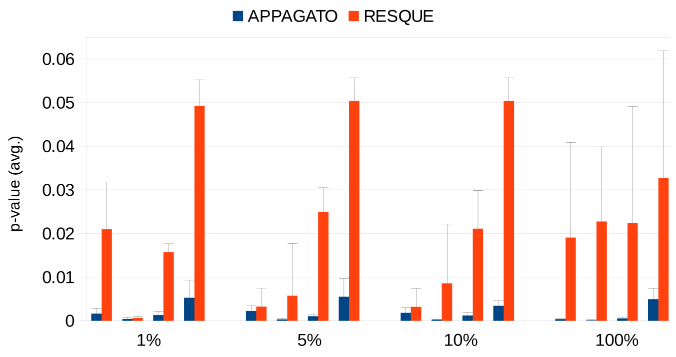


FIG. 10.20: *Averages (and standard deviations) of functional coherence. P-values are grouped by similarity threshold (from 1% to 100%). Within each group, the order of the bars correspond to the S. cerevisiae, D. rerio, H. sapiens and M. musculus networks, respectively. For each query, the functional coherence of the single RESQUE response has been taken into account, while the 100 APPAGATO responses (requested as output) have been averaged to report a single value. The APPAGATO p-values range from* 0.010 *to* 9×10^{-5}*, while RESQUE values range from* 0.053 *to* 6×10^{-4}.

sues. Proteins form tissue-selective complexes and remain inactive in other tissues [28, 282]. Moreover, disease genes have higher transcript levels and more interacting partners in the disease-tissue specific PPI networks with respect to the unaffected tissues [27, 174]. Differently from most of the available PPI network repositories, which represent interactions ignoring where these take place, SPECTRA [193], MyProteinNet [28] and GIANT [114] collect tissue- and tumor-specific networks. Given a tissue or a condition, by integrating expression data from public repositories [26, 228, 268], they score the edges in PPI networks based on the node co-expression.

We used tissue-specific PPI networks extracted from GIANT [114] to analyze protein interaction modules related to the Obsessive-compulsive Disorder (OCD). Nodes have been compared by using the query-target node sequence similarities (as described in Section 10.1.2). OCD is a pathologic neuropsychiatric condition

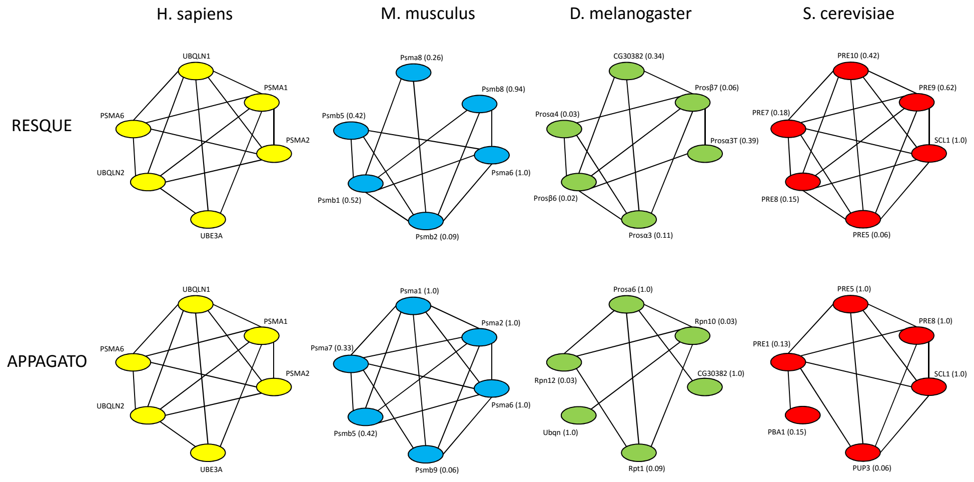


FIG. 10.21: *Top matches of human* Ubiquilin-proteasome complex *(CORUM id: 5209) in Mus musculus, Drosophila melanogaster, and Saccharomyces cerevisiae retrieved by APPAGATO and RESQUE with no filter (100%) on the similarity matrix. Matched nodes in different species are horizontally aligned.*

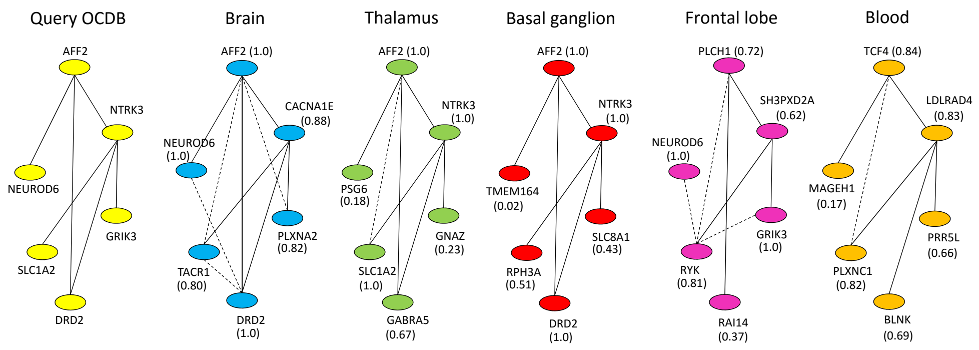


FIG. 10.22: *APPAGATO top matches of OCDB query in Brain, Thalamus, Basal ganglion, Frontal lobe, and Blood tissues. Matched nodes in different tissues are horizontally aligned. Dashed lines represent edges of target graphs that are not present in the query. Solid lines are edges that are present in both the query and the target graphs.*

characterized by obsession (recurring thoughts) and compulsions (recurring behaviors), which produces considerable problems for the patients [222]. Given the nature of such a disorder, we have investigated five tissue-specific networks: Brain, Thalamus, Basal ganglion, Frontal lobe and Blood. The networks have sizes varying from about 3,000 to 12,000 nodes and degrees ranging from 3 to 6[8]. The choice

[8] Due to the size of the target networks and the use of matrix similarities, we could not run the compared systems on these instances. Sections 10.2 reports details on the system performance.

of these tissues is motivated by the fact that cortico-striato-thalamo-cortical circuitry (CSTC) shows dysfunctions in OCD patients [50, 218] and, in this circuit, the functional areas involved are Prefrontal cortex, Basal Ganglion and Thalamus. Moreover, we chose Brain to have a general disease-associate tissue and Blood since several soluble factors circulate in the serum and can influence the mood. According to this, one can try to understand if there are other probable genes, and therefore genetics products, that could have a role in OCD pathophysiology.

We queried a dedicated database, OCDB [222], to achieve lists of disease-candidate genes to be investigated. The connections among genes have been extracted from the Brain network by choosing only edges with medium-high confidential score (greater than 30). Fig. 10.22 reports the *APPAGATO* results on a query composed by 6 nodes (genes), differently associated to OCD according to the OCDB classification: DRD2 (Class 1), SLC1A2 (Class 1), NTRK3 (Class 1), GRIK3 (Class 2), NEUROD6 (Class 4) and AFF2 (Class 4).

The chart confirms that the module is generally well conserved in the Brain (where different occurrences of the query beside the exact one are returned), in Thalamus, and Basal ganglion and let us to formulate some hypothesis. Notably, the glutamate amino acid transporter family SLC1 is involved in function of CSTC circuits [138]. The SLC1 (SLC1A1, SLC1A2, SLC6A2, SLC6A4) family is a heterogeneous group of genes whose members have been deeply studied in OCD. So far, only the neuronal glutamate transporter gene (SLC1A1) has been reported as expressed gene in CSTC circuits [138,222]. The *APPAGATO* results highlight the solute carrier family 8 (sodium/calcium exchanger), member 1 (SLC8A1) as a possible candidate gene in OCD (see the "Basal ganglion" network). By exploring the literature, we have found that SLC8A1 contributes to the regulation of Ca(2+)-dependent events in several cell types [143]. In the literature, a precise association with OCD is not reported, nevertheless the Na(+)/Ca(2+) exchangers (NCX) protein encoded by SLC8A1 contributes to long-term potentiation of the brain and learning, neurotransmitter as well as in immune responses. Moreover, a functional annotation of SLC8A1 using DAVID [129] shows that that SLC8A1 is expressed in the Brain as well as in other tissues (Airway smooth muscle, Heart, Liver, PCR rescued clones, Placenta). By querying OCDB genes in blood tissue, we found an approximate match containing the transcription factor 4 (TCF4) that has a role in Schizophrenia. Therefore, a possible association, in OCD, of the matched genes in the Blood and in the other tissues could be further experimentally investigated.

10.5 Conclusions

We have developed *APPAGATO*, a stochastic and parallel algorithm to find approximate occurrences of a query in biological networks. *APPAGATO* deals with node, edge, and node label mismatches. It is implemented for GPUs. The choice of such devices is motivated by their accessible costs, high-performance, and widespread availability on any personal computer. All above features allow *APPAGATO* to compute efficiently functional and topological node similarity together with fast searching of a large number of query matching within the target graph. The results show that *APPAGATO* outperforms the existing tools in terms of running time and result accuracy and, unlike competitors, it scales also on very large PPI networks.

Part III

Profiling and Analysis Framework

Introduction

This part of the thesis presents a study of advanced profiling models to analyze performance, energy and power consumption of algorithm implementations. It first focuses on a specific class of algorithms called parallel primitives which are widely used as building blocks for more complex applications. The proposed model (Section 12) aims at evaluating such primitives by considering both architectural details and low-level profiling information. Then, it presents a performance model based on complementary features and a set of microbenchmarks to provide a comprehensive characterization of a given GPU device from the performance point of view (Section 13). Thanks to the collected information, it allows accurately estimating the potential performance of the application under tuning, at the same time, it provides programmers with interpretable profiling hints. Later, this part describes a extended set of microbenchmarks to characterize a given GPU device in terms of power and energy consumption beside performance (Section 14). In Section 15, we present an overall model which considers and extends the previous results to allow efficiently driving the application tuning by considering the three design constraints (power, performance, energy consumption) and the characteristics of the target GPU device.

This part of the thesis is organized as follows. Section 11 discusses related work concerning GPU performance models and microbenchmarking. Sections 12, 13, 14, and 15 introduce the methodology and the experimental results in this context. Finally, Section 16 presents the evaluation of the workload decomposition problem as case of study. The section analyzes and compares the most important strategies and approaches for load balancing in the literature in terms of performance, power, and energy efficiency.

11

Related Work

The related work section presents the main approaches in the literature for two fundamental aspects which compose the profiling framework, performance models and microbenchmarking. The section first introduces state-of-the-art models for performance evaluation and prediction which rely on analytical analysis, instruction simulators, and machine learning algorithms. Secondly, it presents the related work for power, performance, and energy consumption analysis and characterization through microbenchmarking.

11.1 Performance models

Different performance models for GPU architectures have been proposed in literature. They can be classified into *specific models*, which apply on a particular application or pattern only, and *general-purpose models*, which are applicable to any program/kernel for a comprehensive profiling [166].

In the class of specific models, [90] proposes an approach for performance analysis of *code regions* in CUDA kernels, while, in [75], the authors focus on profiling divergences in GPU applications. A different analytical approach is proposed in [115], which aims at predicting the kernel execution times of sparse matrix-vector multiplications.

General-purpose models allow profiling applications from more optimization criteria point of view and, thus, they can give different hints on how to optimize the code. As an example, [127] proposes a model for NVIDIA GPUs based on two different metrics: Memory warp parallelism (MWP) and computation warp parallelism (CWP). Although the model predicts the execution costs fairly well, the understanding of performance bottlenecks from the model is not so straightforward. This model has been extended in two different ways [148, 247]. [148] introduces two kernel behaviors, MAX and SUM, and shows how they allow generating predictions close to the real measurements. Nevertheless, they do not provide clues as how to choose the right one for a given kernel. In contrast, [247] extends the model with additional metrics, such as cache effect and SFU instructions.

All these analytical performance models, although accurate in several cases, rely on simulators (e.g., Ocelot, GPGPU, Barra) to collect necessary information for profiling, which implies a high overhead in the profiling phase. An attempt has been made in [292] for collecting more efficiently information on the GPU characteristics and using simple static analysis methods to reduce the overhead of runtime profiling.

Besides the often prohibitive overhead introduced in the profiling phase, especially for complex applications, a big problem of the simulator-based models is portability. They can be applied to profile applications on GPU models that are supported by the simulator, which, often, is not updated to the last releases of GPU models.

Differently from the analytical models, [141] and [233] are based on machine-learning techniques, which allow identifying hardware features and using feature selection, clustering and regression techniques to estimate the execution times. Nevertheless, both these models are inaccurate, thus providing approximate estimations with high variability.

11.2 GPU Microbenchmarking

In [201], the authors evaluated throughput and the power efficiency of three 128-bit block ciphers (AES, Camellia, and SC2000) on Nvidia Geforce GTX 680 with Kepler architecture and on AMD Radeon HD 7970 with GCN architecture. For the comparison, the authors used Nvidia Geforce GTX 580 and AMD Radeon HD 6770 with architecture of one generation earlier. The authors developed a microbenchmark suite that allows understanding that arithmetic logical instructions are required by encryption processing but are eliminated from some of the processing cores in NVIDIA Kepler architecture, unlike AMD graphics core next (GCN) architectures.

In [286], the authors propose an OpenCL microbenchmark suite for GPUs and CPUs. They present the performance results of hardware and software features such as bus bandwidth, memory architectures, branch architectures and thread hierarchy, etc., evaluated through the proposed microbenchmarks on multi-core X86 CPU and NVIDIA GPUs.

In [98], the authors propose a microbenchmarking methodology based on short elapsed-time events (SETEs) to obtain comprehensive memory microarchitectural details in multi- and many-core processors. This approach requires detailed analysis of potential interfering factors that could affect the intended behavior of such memory systems. They lay out guidelines to control and mitigate those interfering factors.

In [185], the authors propose a fine-grained benchmarking approach and apply it on two popular GPUs (i.e., Fermi and Kepler), to expose the previously unknown characteristics of their memory hierarchies. The authors investigate the structures of different cache systems, such as data cache, texture cache, and the translation lookaside buffer (TLB). They also investigate the impact of bank conflict on shared memory access latency. the GPU memory hierarchy, which can help in the software optimization and the modelling of GPU architectures.

Thoman et al. [264] proposed a suite of OpenCL microbenchmarks, which allows measuring functional characteristics of both GPUs and CPUs. Lemeire et al. [158] presented the most complete and comprehensive set of microbenhmarks among those proposed in literature, for both computational units and memory analysis.

Each of these contributions either presents microbenchmarks for characterizing a GPU device from a specific design constraint point of view (performance or power) or presents and analysis of GPU performance and power of a specific application. Nevertheless, all these approaches have three main limitations. First, they are limited to *static* characteristics of GPUs. Indeed, as explained in Section 15.2, also the dynamic characteristics of a GPU are essential to understand how application bottlenecks involving selected functional components or underutilization of them can affect the code quality. Second, they do not cover all the functional components of the GPU devices. Third, they are sensitive to the compilation phase, which often makes the generated low-level code very inaccurate in measuring the GPU characteristics.

12 Parallel Primitives Profiling

Parallelizing software applications through the use of existing optimized primitives is a common trend that mediates the complexity of manual parallelization and the use of less efficient directive-based programming models. Parallel primitive libraries allow software engineers to map any sequential code to a target many-core architecture by identifying the most computational intensive code sections and mapping them into one ore more existing primitives. On the other hand, the spreading of such a primitive-based programming model and the different GPU architectures have led to a large and increasing number of third-party libraries, which often provide different implementations of the same primitive, each one optimized for a specific architecture. From the developer point of view, this moves the actual problem of parallelizing the software application to selecting, among the several implementations, the most efficient primitives for the target platform. This Section presents *Pro++*, a profiling framework for GPU primitives that allows measuring the implementation quality of a given primitive by considering the target architecture characteristics. The framework collects the information provided by a standard GPU profiler and combines them into optimization criteria. The criteria evaluations are weighed to distinguish the impact of each optimization on the overall quality of the primitive implementation. The Section shows how the tuning of the different weights has been conducted through the analysis of five of the most widespread existing primitive libraries and how the framework has been eventually applied to improve the implementation performance of two standard and widespread primitives.

12.1 Introduction

Computing platforms have evolved dramatically over the last years. Because of the physical limitations imposed by thermal and power requirements, frequency scaling has proven to be no longer the solution to increase the performance of processors. As a consequence, many hardware manufacturers have turned to scale the number of cores in a processor in order to boost application performance. Apart from the significantly improved simultaneous multi-threading capabilities, such heterogeneous multi-core platforms also contain general-purpose graphic processing units

(GPUs) to exploit fine-grained parallelism [4]. As a result of such hardware trends, the heterogeneity of these platforms and the need to program them efficiently has led to a spread of parallel programming models, such as CUDA and OpenCL. In this context, many parallel applications frmo different context have been developed for GPUs, ranging from artificial intelligence [43], to electronics design automation [39, 49, 270].

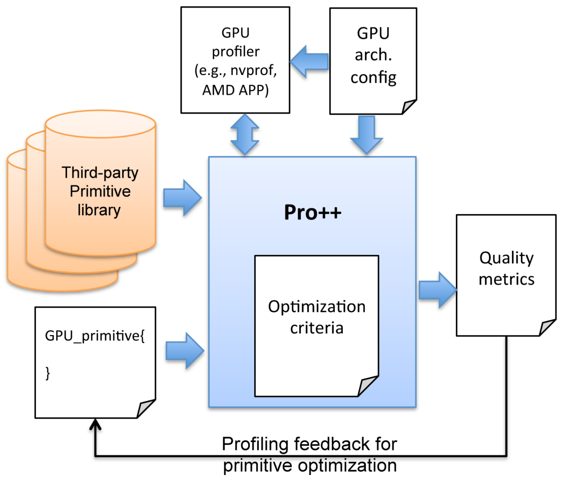

GPU profiler (e.g., nvprof, AMD APP)
GPU arch. config
Third-party Primitive library
Pro++
Optimization criteria
Quality metrics
GPU_primitive{
}
Profiling feedback for primitive optimization

FIG. 12.1: *Overview of the Pro++ framework*

On the other hand, while many software developers possess a working knowledge of basic programming concepts, they typically lack of expertise in developing efficient parallel programs in a short time. As a matter of fact, the programming process with a CUDA or OpenCL-based environment is much more complicated and time-consuming than that with a parallel programming environment for conventional multiprocessor systems. Programmability of such parallel platforms is consequently a strategic factor impacting on the approach feasibility as well as costs and quality of the final product.

In this context, directive-based extensions to existing high-level languages (OpenACC [10], OpenHMPP [223]) have been proposed to help software engineers through sets of directives (annotations) for marking up the code regions intended for execution on a GPU. Based on this information, the compiler generates hybrid executable binary. Despite their user-friendliness and expressiveness, such directive-based solutions require notable effort from the developers in organizing correct and efficient computations and, above all, compilers are often over conservative, thus leading to poor performance gain by the parallelization process [254].

Domain-specific languages (DSLs) (e.g., Delite [255], Spiral [12]) have been also proposed to express the application parallelism for GPUs in specific problem domains. DSL-based approaches allow the language features and the specific problem domain features to be brought closer and, at the same time, the parallel applications to be developed not strictly customized for any particular hardware platform. Nevertheless, these solutions require the user to implement the algorithms by using a proprietary language, with consequent limitations to SW IP reuse and portability.

A more user-friendly and common trend is to implement the application algorithm through existing primitives for GPUs. This generally provides sound tradeoff between parallelization costs and code performance. Such primitive-based programming model relies on identifying parts of code computationally intensive and re-implementing their functionality through one or more basic primitives provided by an existing library. Due to its efficiency, the primitive-based programming model has been recently also combined to both directive-based solutions and DSLs [259] to exploit the portability of annotations/DSLs as well the performance provided by the GPU primitives.

An immediate consequence of such a trend has been the spreading of an extensive list of accelerated, high-performance libraries of primitives for GPUs ([6] and [2] are some exampling collections of them). On the one hand, all these libraries cover a wide spectrum of use cases, such as basic linear algebra, machine learning, and graph applications. On the other hand, many libraries provide different implementations of the same primitives. From the developer point of view, this moves the actual problem of parallelizing a software application to selecting the most efficient primitives for the target platform among several implementations.

The motivation for this work is precisely the observation that it would be nice to measure the implementation quality of a given primitive, with the aim of helping the software developer (i) to choose the best implementation of a given primitive among different libraries, and (ii) to understand whether such a primitive implementation fully exploits the architecture characteristics and how the implementation efficiency could be improved. To do that, this Section presents *Pro++* (see Figure 12.1), an enhanced profiling framework for the analysis and the optimization of parallel primitives for GPUs. *Pro++* collects the information about a given primitive implementation (i.e., profiling metrics) through a standard GPU profiler. The framework combines the standard metrics into *optimization criteria*, such as, multiprocessor occupancy, load balancing, minimization of synchronization overheads, and memory hierarchy use. The criteria are evaluated, weighed, and finally merged into an overall measure of quality metrics. The quality metrics allows the user to classify and compare the different implementations of a primitive in terms of performance over the selected GPU architecture configuration.

The main contributions of this work are the following:

- A classification of optimization criteria that mainly impact on the primitive performance.
- An analysis of such optimization criteria over five different primitive libraries for GPUs to weigh the impact of each single criterion on the overall primitive performance.
- A framework that combines profiling metrics, optimization criteria, and weights to provide (i) an overall quality metrics of a given primitive and (ii) profiling feedbacks to improve the primitive implementation.

The work is organized as follows. Section 12.2 summarizes the key concepts of GPU profiling. Section 12.3 presents the optimization criteria by which the primitives are evaluated. Section 12.4 reports the analysis conducted to measure the impact of the optimization criteria on the overall quality metrics of primitives. Section 12.5 presents the case studies of Pro++ application while Section 12.6 is devote to the conclusions.

12.2 Profiler Metrics

Extracted Information	Information source	Description
#SM	Hardware Info	Total number of stream multiprocessors.
#SM_threads	Hardware Info	Total number of threads per stream multiprocessor.
reg_granularity	Hardware Info	Register allocation granularity.
block_size	Kernel Configuration	Number of threads per block associated to a kernel call.
grid_size	Kernel Configuration	Number of thread blocks associated to a kernel call.
#registers	Compiler Info	Number of used registers per thread associated to a kernel call.
SM_registers	Hardware Info	Total number of registers per Streaming Multiprocessor.
#Blocks_per_SM	Profiler Event	Number of resident blocks per Streaming Multiprocessor.
max_SM_blocks	Hardware Info	Maximum number of resident blocks per Streaming Multiprocessor.
#resident_threads	Hardware Info	Maximum number of threads that can run concurrently on the device.
Static_SMem	Compiler Info	Bytes of static shared memory per block.
Dynamic_SMem	Kernel Configuration	Bytes of dynamic shared memory per block.
SM_SMem	Hardware Info	Total available shared memory per Streaming Multiprocessor.
active_warps	Profiler Event	Number of active warps per cycle per SM.
threads_launched	Profiler Event	Number of threads run on a multiprocessor.
stall_sync	Profiler Event	Percentage of stalls occurring because the warp is blocked at a __syncthreads() call.
Int_instr, SP_instr, DP_instr	Profiler Event	Number of arithmetic instructions (integer, single-precision floating point, double-precision floatig point) executed by all threads.
cudacopy_size	Profiler Event	Number of bytes associated to a host-device memory transfer function.
DRAM_transactions	Profiler Event	Total number of DRAM memory accesses.
#L1_transactions	Profiler Event	Total number of L1 memory accesses.
#L2_transactions	Profiler Event	Total number of L2 memory accesses.
$\#Mem\_instr_T$	Profiler Event	Total number of global memory instructions of size T. Where T can be 1/2/4/8/16 bytes.
#SharedLoadTrans, #SharedStoreTrans	Profiler Event	Total number of shared memory load/store transactions.
#SharedLoadAcc, #SharedStoreAcc	Profiler Event	Total number of shared memory load/store accesses.
alu_utilization	Profiler Event	Utilization level of the GPU arithmetic units (ALU/FPU).
ld_st_utilization	Profiler Event	Utilization level of the GPU load/store instruction units.
KernelStart, cudacopy_start_time, KernelExeTime, cudacopy_time	Profiler Info	Start time and duration of a kernel call or CUDA memory transfer function.

Table 12.1: Profiler events, compiler information, hardware (device) information, and kernel configuration considered in the proposed optimization criteria.

Developing high performance applications requires adopting tools for understanding the application behaviour and for analyzing the corresponding performance. At the state of the art, there exist several profiling tools for GPU applications that provide advanced profiling information through the analysis of *events*, kernel configuration, hardware and compiler information. Table 12.1 summarizes a selected list of such profiling information, which are strongly related to the application performance.

In this work we refer to the NVIDIA *nvprof* profiler terminology and information. However, the proposed methodology is independent from the adopted profiler. *Nvprof* has two operating modes that generate two distinct outputs. The first mode is the *trace mode*, which provides a timeline of all activities taking place on the GPU in chronological order. From this mode, we extract the kernel configuration and any timing associated to a kernel (e.g., start time, latency, etc.). The second mode, called *summary mode*, reports a user-specified set of events for each kernel, both aggregating values across the GPU units and showing the individual counter for each SM.

12.3 Optimization Criteria

We define different *optimization criteria*, which express the quality of a given primitive to exploit a GPU characteristic. Examples are the occupancy of all the computing (SP) resources, the load balancing, and the memory coalescing. The selection of the most representative and influential optimization criteria has been guided by the best practices guide [206], by the main CUDA books [145] [68] and by the programming experience [59]. The optimization criteria are defined to cover all the crucial properties a GPU application should satisfy to exploit the full potential of the GPU device. We consider the properties adopted in [75, 90, 127, 148, 247] concerning divergence, memory coalescing, and load balancing. In addition, we define optimization criteria to cover synchronization issues. Differently from the literature, the proposed criteria are more accurate (i.e., fine-grained) to evaluate such properties. All the criteria are defined in terms of events and static information, which are all provided by any standard GPU profiler. Each criterion value is expressed in the range [0, 1], where 0 represents the worst and 1 represents the best evaluation of such an optimization.

12.3.1 Occupancy (OCC)

In order to take advantage of the computational power of the GPU, it is important to maximize the SP utilization of each SM. This criterion gives information on the maximum theoretical occupancy of the GPU multiprocessors in terms of active threads over the maximum number of threads that may concurrently run on the device.

The criterion value, which is calculated statically, depends on the kernel configuration as well as on the kernel implementation. In particular, it depends on the block size (i.e., number of threads per block), grid size (i.e., number of blocks per kernel) as well as amount of used shared memory for the kernel variables, and number of used registers. In general, the kernel configuration of the primitives is set at compile time by exploiting information on the device compute capability and no tuning is allowed to the user (to comply with the principle of user-friendliness). The criterion takes into account how well the limited resources like registers and shared memory have been exploited in the kernel implementation and, thus, how and how many variables have been declared (e.g., automatic and shared). A low value means underutilization of the GPU multiprocessors. More in details, the overall occupancy is calculated as the minimum value between the occupancy related to block_size (taking into account also the maximum number of blocks per SM), to shared memory utilization (StaticalSMem + DynamicSMem), to the register utilization (#registers), and to the grid_size with respect to the minimum number of blocks required to keep busy all SMs [1]:

$$\text{SMEM\_OCC} = \left\lfloor \frac{\text{SM\_SMem}}{\text{StaticSMem} + \text{DynamicSMem}} \right\rfloor$$

$$\text{Reg\_OCC} = \left\lfloor \frac{\frac{\text{block\_size}}{\text{warp\_size}} \cdot \lceil \text{warp\_size}\cdot\text{\#registers}\rceil^{[\text{reg\_granularity}]}}{SM\_\text{Re}gister} \right\rfloor$$

$$\text{Block\_OCC} = \max\left(\text{max\_SM\_blocks}, \left\lfloor \frac{SM\_threads}{\lceil \text{block\_size} \rceil^{[\text{warp\_size}]}} \right\rfloor \right)$$

$$\text{Thread\_OCC} = \frac{\text{grid\_size} \cdot \lceil \text{block\_size} \rceil^{[\text{warp\_size}]}}{\text{\#resident\_threads}}$$

$$OCC = \min\left(\text{SMEM\_OCC}, \text{Reg\_OCC}, \text{Block\_OCC}, \text{Thread\_OCC}, 1\right)$$

12.3.2 Host Synchronization (HSync)

Many complex parallel applications organize the compute-intensive work into several functions offloaded to GPUs through host-side kernel calls. Depending on the code complexity and on the workflow scheduling, this mechanism may involve significant overhead that can compromise the overall application performance. The host synchronization criterion aims at evaluating the amount of time spent to coordinate the kernel calls. It is defined as follows:

$$\text{HS{\small YNC}} = \frac{\sum_{\text{i}=1}^{\text{N}} \text{KernelExeTime}_{\text{i}}}{\text{KernelStart}_N + \text{KernelExeTime}_N - \text{KernelStart}_1}$$

where N is the number of kernels in which the application has been organized, $KernelExeTime_i$ is the real execution time of kernel i on the device, and $KernelStart_i$ is the clock time in which kernel i starts executing. A fragmented computation that involves many kernel invocations and many small data transfers is represented by a low value of this criterion.

This criterion helps programmers to understand if the overall application speedup is bounded by an excessive host synchronization activity. Merging different kernels, using inter-block synchronization [281] or reducing small memory transfers improve the quality value of this criterion.

[1] The notation $\lceil A \rceil^{[B]}$ denotes the nearest multiple of B equal or greater than A.

12.3.3 Device Synchronization (DSync)

In GPU computing, the synchronizations of threads in blocks are one of the main causes of idle state and, thus, they strongly impact on the application performance. Beside introducing overhead in the kernel execution, they also limit the efficiency of the multiprocessors in the warp scheduling activity. This criterion gives a quality value of a kernel by measuring the total time spent by the kernel for synchronizing thread blocks:

$$\mathrm{D}\textsc{Sync} = 1 - \left(1 - \frac{\mathrm{TotActiveWarps/warps\_size}}{\mathrm{CLK\_cycles}}\right) \cdot\ \mathrm{StallSync}$$

where $\mathrm{TotActiveWarps} = \sum_{\mathrm{i=1}}^{\mathrm{CLK\_cycles}} \mathrm{ActiveWarps_i}$.

$|Warps|$ represents the maximum number of thread warps of the device, while CLK_cycles represents the total number of GPU clock cycles elapsed to execute the kernel. The formula takes into account the number of active warps at each clock cycle, and it adds them to the total counter $TotActiveWarps$. The value in the round bracket represents the overall percentage of inactivity of the GPU warps (i.e., warps in waiting state). *StallSync* represents the percentage of the GPU time spent in synchronization stalls over the total number of stalls. *StallSync* depends on the load balancing among threads as well as the number of synchronization points (i.e. thread barriers) in the kernel.

12.3.4 Thread Divergence (TDiv)

Branch conditions that lead threads of the same warp to execute different paths (i.e., thread divergence) are one of the main causes of inefficiency of a GPU kernel. This criterion evaluates the thread divergence of a kernel as follows:

$$\mathrm{T}\textsc{Div} = \frac{\#\mathrm{ExeInstructions}}{\#\mathrm{PotExeInstructions}}$$

where *#ExeInstructions* represents the total number of instructions executed by the threads of a warp and *#PotExeInstructions* represents the total number of instructions potentially executable by the threads of a warp. The final value is calculated as the average over all warps run by the kernel. The criterion gives a clear evaluation of the branching factor of a kernel code.

12.3.5 Warp Load Balancing ($\mathbf{LB_W}$)

This criterion expresses how well the workload is uniformly distributed over the cores of each single SM:

$$LB_W = \frac{\left(\frac{\mathrm{TotActiveWarps}}{\mathrm{TotActiveCycles}}\right)}{\left(\frac{\mathrm{block\_size}}{\mathrm{warp\_size}}\right) \cdot \#Blocks\_per\_SM}$$

where *TotActiveCycles* represents the total number of clock cycles in which the single SMs are not in idle state. The formula takes into account the number of active warps at each clock cycle, and it adds them to the total counter $TotActiveWarps$. The denominator represents the theoretical maximum occupancy of the SMs in terms of number of warps. It is calculated by considering the block size and the number of blocks mapped to each single SM. A low value of this criterion underlines that some warps doing most of the work while others are in the idle state. This is a common behavior in irregular problems and suggests to programmers to choose a different load balancing strategy.

12.3.6 Streaming Multiprocessor Load Balancing ($\mathbf{LB_{SM}}$)

Besides the load balancing on each single SM, the model evaluates the load balancing at SM level. The SM load balancing criterion is defined as follows:

$$LB_{SM} = 1 - \frac{\max\limits_{SM}(\text{TotActiveCycles}) - \text{AvgCycles}}{\max\limits_{SM}(\text{TotActiveCycles})}$$

where

$$\text{AvgCycles} = \frac{\sum\limits_{SM} \text{TotActiveCycles}}{\#\text{SM}}$$

The formula expresses the SM Load Balancing as the difference between the maximum execution cycles required among all SMs and the best case where all SMs take the same execution cycles.

12.3.7 L1/L2 Granularity ($\mathbf{Gran_{L1}/Gran_{L2}}$)

GPU applications require optimized data access patterns and properly aligned data structures to achieve high memory bandwidths. In particular, efficient applications hide the latency of memory accesses by combining multiple memory accesses into single *transactions* that match the granularity (i.e., the cache line size) of the memory space[2]. Hides latency of memory accesses in CUDA is feasible by combining multiple memory accesses into a single transaction that match the granularity (cache line size) of the memory space. The proposed performance model includes two complementary criteria to describe the quality of memory access patterns:

$$Gran_{L1} = \frac{\#\text{L1\_transactions} \cdot 128}{\sum\limits_{T \in \{Mem\_instr\}} \#\text{Mem\_instr}_\text{T} \cdot \text{size}_\text{T}}$$

$$Gran_{L2} = \frac{\#\text{L2\_transactions} \cdot 32}{\sum\limits_{T \in \{Mem\_instr\}} \#\text{Mem\_instr}_\text{T} \cdot \text{size}_\text{T}}$$

The criteria take into account the number of actual transactions towards the

[2] This concept applied to the L1 cache is also known as *memory coalescing.*

L1(L2) memory, the cache line size (128 Bytes for L1, 32 Bytes for L2), the total number of memory instructions (`load` and `store`) to access the global memory, and the size of their accesses $size_T$ (1/2/4/8/16 Bytes). The ratio of useful data accesses to total data accesses is calculated by comparing the total size of the data required by threads with the number of transactions multiplied by the respective memory granularity.

12.3.8 Shared Memory Efficiency (SMem$_{\mathbf{eff}}$)

This criterion measures the kernel efficacy to exploit the *data locality* concept through the on-chip shared memory. The shared memory allows high memory bandwidth for concurrent accesses, but it requires appropriate access patterns to achieve the full efficiency. On the other hand, an excessive and disorganized use of the shared memory leads to bank conflicts, which involve the memory instructions to be re-executed thus serializing the thread execution flow. This optimization criterion is defined as follows:

$$\text{SMem}_{\text{eff}} = \frac{\#\text{SharedLoadTrans} + \#\text{SharedStoreTrans}}{\#\text{SharedLoadAcc} + \#\text{SharedStoreAcc}}$$

The formula is defined in terms of total number of transactions towards shared memory for both load and store operations over the total number of accesses in shared memory for load and store instructions (which includes the re-executed memory instructions due to bank conflicts).

12.3.9 Computation Intensity (CI)

This criterion takes into account the amount of instructions that make use of arithmetic units, both integer and floating point, and load-store (instruction) units (LDST) for all memory spaces.

$$CI = \frac{\text{alu\_utilization} + \text{ld\_st\_utilization}}{2}$$

The formula expresses the computation intensity as the average of the utilization level of the ALU/FPU units and the LDST units. This criterion is strictly related to the code optimization. A high value of computation intensity criterion means that the ALU, FPU and LDST units have not been wasted. Considering also the same functionality of all tested code for the same primitive, this information indicates how much the code is optimized. A high value of the utilization level of an instruction unit indicates that the unit performs a high number of independent operations of the same type. As a consequence, all the operations can be run in parallel and saturate the computational throughput of the specific unit.

12.3.10 Data Transfer (DT)

It takes gives a quality measure of the primitive to address the data transfer overhead. As an example, pipelining (overlapping) between data transfer and data computation allows the primitive to rich higher value of this criterion:

$$DT = 0.5 + \frac{\text{Overlapping\_mem\_transf}}{\text{GPU\_allocated\_byte} + \sum \text{cudacopy\_size}} - \frac{\sum \text{cudacopy\_size}}{\text{GPU\_allocated\_byte} + \sum \text{cudacopy\_size}}$$

Overlapping data transfers with kernel computation may reduce the execution time, but it requires a fine-tuning of the data size to be transferred. Too large data sizes may involve no advantage, while too small sizes may involve heavy synchronization overhead.

It also takes into account the amount of bytes transferred in the host-device communication during a kernel computation over the actual I/O bytes required for the computation. Any extra data transfer between host and device is considered as overhead.

12.3.11 Overall Quality Metrics (QM)

All the proposed values of the optimization criteria are finally combined into an overall quality metric to provide, through a single value, an evaluation of the profiled code. We express this value as the weighted average of the values of the optimization criteria as follows:

$$\mathbf{QM} = \frac{\begin{array}{l}\text{OCC} \cdot \text{W}_{\text{OCC}} + \text{HSync} \cdot \text{W}_{\text{HSync}} + \\ \text{DSync} \cdot \text{W}_{\text{DSync}} + \text{TDiv} \cdot \text{W}_{\text{TDiv}} + \\ \text{Gran}_{\text{L1}} \cdot \text{W}_{Gran_{L1}} + \text{Gran}_{\text{L2}} \cdot \text{W}_{Gran_{L2}} + \\ \text{LB}_{W} \cdot \text{W}_{\text{LB}_{W}} + \text{LB}_{SM} \cdot \text{W}_{\text{LB}_{SM}} + \\ \text{SMem}_{\text{eff}} \cdot \text{W}_{\text{SMem}_{\text{eff}}} + CI \cdot W_{CI} + DT \cdot W_{DT}\end{array}}{\begin{array}{l}\text{W}_{\text{OCC}} + \text{W}_{\text{HSync}} + \text{W}_{\text{DSync}} + \text{W}_{\text{TDiv}} + \\ \text{W}_{Gran_{L1}} + \text{W}_{Gran_{L2}} + \text{W}_{\text{LB}_{W}} + \text{W}_{\text{LB}_{SM}} + \\ \text{W}_{\text{SMem}_{\text{eff}}} + W_{CI} + \text{W}_{\text{DT}}\end{array}}$$

W_{xy} express the weight of each single criterion in the overall quality measure. In this work, we tuned the different weights through the analysis of different libraries of primitive, as detailed in the following section.

12.4 Weighing of Optimization Criteria on the Overall Quality Metrics

12.4.1 Parallel Primitives

The impact of the optimization criteria classified in the previous section on the overall quality metrics has been measured through the analysis of five primitive

Parallel Primitives		**Library**				
		ArrayFire	**CUB**	**CUPDD**	**MGPU**	**Thrust**
Independent Linear Transformation	Fill/Generate/Sequence/ Tabulate	X				X
	Modify/Transform/ Replace/Adjacent Difference	X				X
	Modify_If					X
	Comparison					X
	Simple Copy					X
Advanced Coping	Gathering					X
	Gathering_If					X
	Scattering					X
	Scattering_If					X
Reduction	Couting	X				X
	Extrema	X	X			X
	Reduction	X	X	X	X	X
	Reduce_by_keys/ Segmented_Reduction	X	X		X	X
	Histogram	X	X			
Prefix-Scan	Inclusive	X	X	X	X	X
	Exclusive	X	X	X	X	X
	Prefixscan_By_Key/ Segmented_Prefixscan			X		X
Search	Unsorted Search/Find					X
	Vectorized Binary Search				X	X
	Load-Balancing Search				X	
Reordering	Partitioning/Partitioning_If		X			X
	Compaction/Copy_If/Select		X	X		X
	Merge				X	X
	Merge Sort	X			X	
	Radix Sort		X	X		X
Set (ordered)	Union	X			X	X
	Intersection	X			X	X
	Set Difference				X	X
	Unique	X	X			X

Table 12.2: Parallel primitives evaluated for the weight tuning. The table reports also the alternative names of primitives.

libraries for NVIDIA GPU architectures. The first library, *Thrust* v1.8.1 [124], is provided by NVIDIA in the CUDA Toolkit and it is based on the C++ Standard Template Library high-level interface. This library provides a wide range of parallel primitives to simplify the parallelization of fundamental parallel algorithms such as *scan*, *sort*, and *reduction*. The second library, *CUB* v1.4.1 [188], provides a set of high performance parallel primitives for generic programming for both host and device programming layer. The third library, *CUDPP* v2.2 [122], focuses on common data-parallel algorithms such as *reduction* and *prefix-scan*, and includes also a set of specific-domain primitives such as compression and suffix array functions. The fourth library, *ModernGPU* v1.1 (MGPU) [32], implements basic primitives such as *reduction* and *prefix-scan* but the main goal of ModernGPU is providing very efficient implementations of *parallel binary search* algorithm applications such as *segmented reduction/prefix-scan*, *load balancing* algorithm, *merge*, *set operations* and matrix-vector multiplication. Finally, *ArrayFire* v3 [176] includes hundreds of high performance parallel computing functions. In particular, it is focused on complex algorithms across various domains such image processing, computer vision, signal processing and linear algebra. In *ArrayFire*, the common parallel primitives are proposed as vector algorithms.

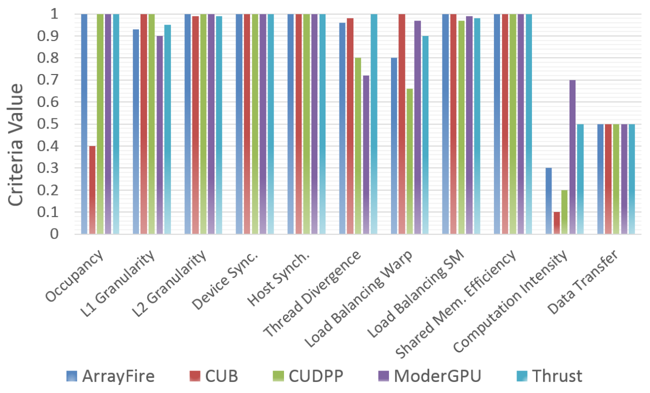
These libraries have been selected as they provide different implementations of widely used and common primitives for the parallelization of fundamental algorithms. This allowed us to compare such implementations by running them over several datasets and by measuring their actual speedups w.r.t. a reference sequential implementation. The comparison results has been finally used to tune the weight of each optimization criteria in the overall quality metrics.

Table 12.2 summarizes the parallel primitives that have been evaluated for such a tuning, by specifying which libraries provide an implementation of a specific primitive. The primitives are grouped by similar functionality in seven main classes. The most basic primitives implementing data elaboration are grouped in the *Independent Linear Transformation* class, which applies concurrent operations on every single element of the input data. This class includes primitives implementing predicate functions for linear transformation on subsets of the input data as well as on multiple sets of data concurrently. The second class, *Advanced Copying*, includes two classic collective operations, i.e., *gathering* and *scattering*, as well as their version with *predicate*. The *Reduction* class refers to all the primitives that apply an operation to the input data and that return a single value as result (e.g., counting, maximum, reduction). The segmented version of the reduction applies the operation to a subset of input data. The fourth class includes all variants (i.e., inclusive, exclusive, etc.) of the *prefix-scan* procedure, which represents the building blocks of many parallel algorithms. The *search* class contains primitives for searching elements in sorted or unsorted sets of data. The *load-balancing* primitives are a specialization of the *vectorized sorted search*. They are largely used to extrapolate, from a given input data, the indices to map threads to the corresponding input elements. The primitives in the *Reordering* class include different procedures to manipulate the input data or to select a subset of such a data by using predicates. Finally, the *Set* class covers the most common operations on sets represented as continuous sorted data values.

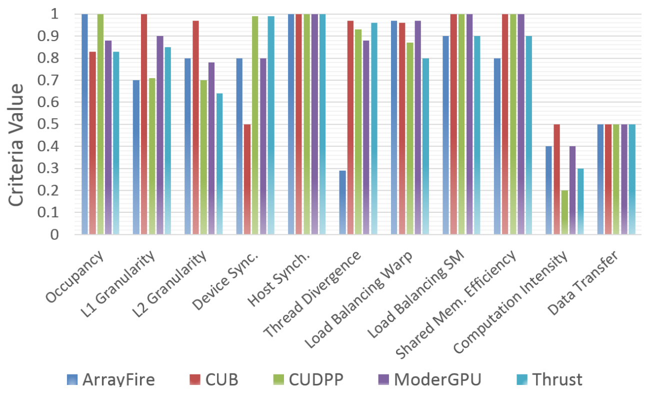
12.4.2 Evaluation

For all parallel primitives, we firstly measured the value of each optimization criterion as proposed in Section 12.3. The evaluation of all primitives has been run on two different systems: a NVIDIA Kepler GeForce GTX 780 device with CUDA Toolkit 7.5, AMD Phenom II X6 1055T 3GHz host processor, and Debian 7 OS and a NVIDIA Fermi GeForce GTX 570 device with CUDA Toolkit 7.0, AMD FX-4100 1.4 GHz host processor, and Debian 7 OS. The dataset applied for the evaluation consists of a large set of random generated input data.

Figure 12.2 reports, as an example, the values of the optimization criteria of the *reduction* and *prefix-scan* primitives. The figure shows that the *Host Synchronization* criterion reaches the maximum value for all the implementations of the five libraries of both the primitives. This is due to the fact that both the



(a) *Reduction*



(b) *Prefix-Scan*

FIG. 12.2: *Optimization criteria evaluation of the reduction and prefix-scan primitives*

Parallel Primitives	GPU vs. CPU Simulation speedup					Quality metrics value ([0, 1])				
	ArrayFire	**CUB**	**CUDPP**	**MGPU**	**Thrust**	**ArrayFire**	**CUB**	**CUDPP**	**MGPU**	**Thrust**
Compaction		67	24		14		0.87	0.75		0.59
Merge				32	21				0.93	0.78
Partition		44			6		0.83			0.58
PrefixScan	63	223	114	135	68	0.82	0.81	0.78	0.89	0.87
Reduction	1009	961	865	1074	1069	0.66	0.89	0.74	0.76	0.69
Segmented PrefixScan			26		4		0.93		0.85	0.62
Segmented Reduction	err	28		28	8	err	0.86			0.46
SetUnion	3			13	3	0.68			0.90	0.73
Sort	80	85	39	48	80	0.68	0.69	0.51	0.58	0.69
Unique	err	73			17	err	0.70			0.56
Vect. Binary Search				527	167				0.48	0.32

TABLE 12.3: Quality metrics values obtained with $W_{OCC} = 30$; $W_{Gran_{L1}} = 100$; $W_{Gran_{L2}} = 100$; $W_{\text{HSync}} = 40$; $W_{\text{DSync}} = 15$; $W_{TDiv} = 40$; $W_{LB_W} = 50$; $W_{LB_{SM}} = 30$; $W_{\text{SMem}_{\text{eff}}} = 30$; $W_{CI} = 100$; $W_{DT} = 50$ and the corresponding actual GPU vs. CPU simulation speedup. Blank cells indicate that the corresponding libraries do not support the parallel primitive, while the `err` notation means that the primitive execution returns a run-time error.

reduction and the *prefix-scan* execute few kernel calls to compute the respective algorithms involving negligible host-device synchronization overhead. The different implementations of the reduction also show a high value in almost all criteria except *Computation Intensity* and *Data Transfer*. This is due to two main reasons. First, the reduction primitive implements a highly regular computation on the input data, which does not cause work unbalance and, second, involves a simple memory access pattern that allows to achieve memory coalescing. In contrast, the *prefix-scan* primitive has been implemented, in all the evaluated libraries, through a two-phase algorithm that presents a more complex memory access pattern. This involves lower values of *L1 granularity* and *L2 granularity*. The algorithm requires also a sophisticated control flow that affects the *device synchronization* and the *thread divergence* criteria. Finally, all implementations of both algorithms shows a low value of *computation intensity* criterion because such primitives are clearly memory-bounded. This characteristic limits the opportunity to take advantage of the huge processing power of the GPU.

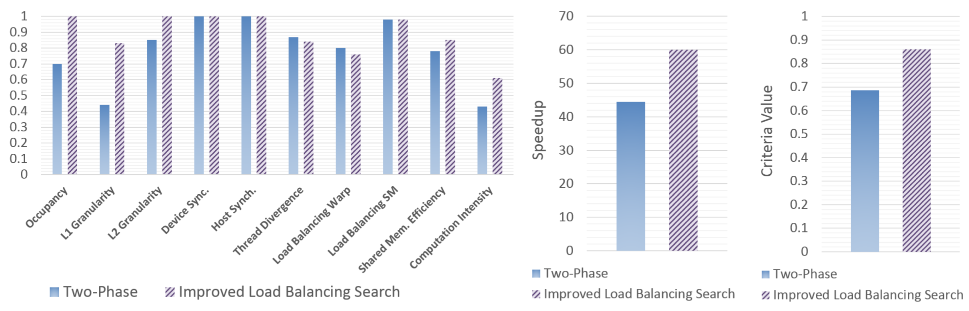
The impact of each criteria on the overall quality metric value has been weighed by considering the criteria values and the actual CPU vs. GPU speedup of each single primitive obtained during simulation. The tuning has been performed with the aim of obtaining the quality metric value of each primitive implementation linearly proportional to the actual CPU vs. GPU speedup of such an implementation. The weight values of the optimization criteria are calculated through a multi-variable regression analysis between all information returned by the different criteria and the execution time. Since the weights depend on the actual architecture, future work aims at automating such a weight computation. The idea is to define a software framework based on a collection of primitives to be run on the target architecture and that automatically extrapolates the weight values.

Table 20.3 reports some of the most meaningful obtained results. The table reports the weights of the optimization criterion extrapolated during simulation, the corresponding quality metrics values and the actual CPU vs. GPU simulation speedup of each parallel primitive. The results show how, given the weights reported in the table caption, the values of the overall quality metrics reflect the actual simulation speedup. The performance accuracy of the proposed model is with 10%-15%, as shown in the experimental results. All the other results, which have not been reported in the table for the sake of brevity, show the same correlation.

From the results reported in Figure 12.2, it is possible to compare different implementations of a given primitive in terms of performance and to understand which characteristics of such implementations lead to the corresponding speedup. As an example, the *Thrust* library provides the best implementation of the *reduction* primitive even though such an implementation presents a value lower than one for *load balancing warp* criterion. On the other hand, the code has been implemented by fully exploiting *L1/L2 Granularity* and by showing a good value of *computation intensity*, whose criteria values have more impact in the overall quality metrics. The *reduction* primitive implemented in *CUDPP* shows low values of *load balancing warp*, *thread divergence* and *computation intensity* that on average are worse than the other library implementations. This underlines that the threads organization and coordination presents many issues in the primitive. Another example is the very low value of *thread divergence* criterion obtained with the *prefix-scan* primitive of *ArrayFire*. The divergence issue indicates a high number of different execution paths among warp threads that, combined with a low value of *L1 Granularity*, represent the main performance bottlenecks.

This analysis allows us to understand whether, given a primitive implementation, there is room to improve such an implementation and how. We applied the proposed profiling framework to analyze and improve the implementations of a *load balancing search* and a *matrix transpose*, as explained in the following section.

12.5 Case Studies



(a) *Optimization criteria values* ([0, 1]) (b) *CPU vs. GPU sim. speedup* (c) *Quality metrics values ([0 ,1])*

FIG. 12.3: *Load balancing search primitive evaluation*

12.5.1 The *Load Balancing Search* Primitive

The *load balancing search* is a special case of *vectorized sorted search* (i.e., binary search). It is commonly applied as an auxiliary function to uniformly partition irregular problems. Given a set of input values that represent the problem workload, the primitive generates a set of indices for mapping threads to the corresponding input elements.

Among the libraries evaluated in this work, only *ModernGPU* provides an implementation of the *load balancing search* primitive. We applied *Pro++* to such a primitive to calculate the optimization criteria values, the CPU/GPU simulation speedup, and the overall quality metric value by considering the weights proposed in Section 12.4 (Table 20.3). Figure 12.3 reports the results (*MGPU* columns). Then, starting from the *ModernGPU* implementation, we optimized the code by exploiting the profiling information with the aim of improving the CPU vs. GPU simulation speedup.

Considering the different optimization criteria weights, we started from the analysis of the criteria related to the memory coalescing. To improve these values, we modified the code to better organize the data in shared memory, registers and texture memory. Such a modification led to a better organization of the data in local memory, which also simplified the management of the memory accesses and allowed us to improve the memory coalescing among threads. These first modifications of the code increased both the *L1 Granularity* and *L2 Granularity* criteria values from 0.44 to 0.83 and from 0.85 to 1, respectively. Further improving memory coalescing has been evaluated as a hard task, due to the many sparse global memory accesses that are closely related to the algorithm. Thus, it has not been further investigated.

On the other hand, improving the two memory criteria required the introduction of many extra control flow statements, which decreased the value of the *thread divergence* criterion with respect to the original *ModernGPU* implementation. Nevertheless, considering such a decrease and the weight of the instruction optimization criterion, we didn't invest effort to limit such a side-effect.

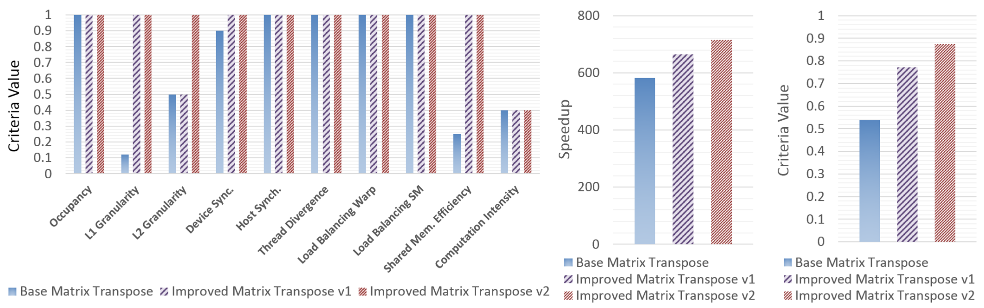
Then, the analysis results underline the low value of the *Occupancy* criterion. To improve this criterion, we modified the code by improving the kernel configuration, the use of automatic variables (and thus the use of SM registers), and the allocation of shared memory. Beside an improvement on occupancy, these modifications had an impact on the value of the *load balancing* criteria. This is due to the fact that the execution flows of all threads during the primitive execution take similar paths and, as a consequence, improving the occupancy criterion leads also to an improvement of the load balancing criteria. The modifications also slightly reduced the *thread divergence* and *Load Balancing Warp* values, which, on the other hand, still remains high. As a consequence, any further investigation or modification of the code, targeting thread divergence would not be worth to improve the overall quality of the primitive implementation. The *device* and *host synchronization* criteria had the highest values, both in the original and the modified version of the code. Thus, no modifications on barriers or synchronization have been considered.

In conclusion, the use of Pro++ allowed us to improve the *loading balancing search* primitives by better concentrating the effort in those code optimizations with more room for improvement and, as a consequence, to save time. The case of study has shown how Pro++ framework has been applied to significantly improve step-by-step, in the optimization cycle, the performance of the *load balancing search* exploring the suggested guideline on the optimization criteria.

12.5.2 The *Matrix Transpose*

Transpose of a matrix is a basic linear algebra operation that has a deep impact in many computational science applications. The performance of matrix transpose is often compared with matrix copy due to the memory bottleneck. We analyzed the matrix transpose implementation presented in [227], which is characterized by data tiling in shared memory and thread organization in 2D hierarchical grids and blocks.

Figure 12.4 shows the results. The original code already provides values close to the maximum for the *host* and *device synchronizations*, *thread divergence*,



(a) *Optimization criteria values ([0 ,1])* (b) *CPU vs. GPU sim. speedup* (c) *Quality metrics values ([0 ,1])*

FIG. 12.4: *Matrix Transpose evaluation*

Warp/SM load balancing and *occupancy* criteria. This is due to the fact that the application algorithm relies on very regular and independent tasks.

All the other criteria have very low values (between 0.1 and 0.5), thus we investigated to improve the code by considering memory related criteria both for global and shared memory spaces.

In the first optimization (Version1), we focused on improving the shared memory bank conflicts (*shared memory efficiency* criterion) by applying the *memory padding technique*. The optimization has been designed mainly for the NVIDIA Fermi architecture that as a low number of independent memory banks. The memory padding has less impact on NVIDIA Kepler architecture and the gained speedup is marginal. We also improved the *device synchronization* criterion by removing barriers and by re-organizing the execution flow in order to assign an independent task to each warp .

In the second optimization (Version3) we have taken into account the memory access patterns to improve the *L1* and *L2 granularity* criteria. Their low values suggest that the memory accesses do not match the granularity of the respective caches, thus involving a waste of the memory bandwidth. We fully optimized both the criteria by simply re-organizing the thread block configuration and by resizing the memory *tiles*.

12.6 Conclusion

This Section presented *Pro++*, a profiling framework for GPU primitives that allows measuring the implementation quality of a given primitive. The Section showed how the framework collects the information provided by a standard GPU profiler and combines them into optimization criteria. The criteria evaluations are weighed to distinguish the impact of each optimization on the overall quality of the primitive implementation. The Section reported the analysis conducted on five among the most widespread existing primitive libraries to tune the different weights. Finally, the Section presented how the framework has been applied to improve the implementation performance of two standard GPU primitives, i.e., the load balancing search and the matrix transpose.

13

A Fine-grained Performance Model

The increasing programmability, performance, and cost/effectiveness of GPUs have led to a widespread use of such many-core architectures to accelerate general purpose applications. Nevertheless, tuning applications to efficiently exploit the GPU potentiality is a very challenging task, especially for inexperienced programmers. This is due to the difficulty of developing a SW application for the specific GPU architectural configuration, which includes managing the memory hierarchy and optimizing the execution of thousands of concurrent threads while maintaining the semantic correctness of the application. Even though several profiling tools exist, which provide programmers with a large number of metrics and measurements, it is often difficult to interpret such information for effectively tuning the application. This Section presents a performance model that allows accurately estimating the potential performance of the application under tuning on a given GPU device and, at the same time, it provides programmers with interpretable profiling hints. The Section shows the results obtained by applying the proposed model for profiling commonly used primitives and real codes.

13.1 Introduction

Even though graphics processing units (GPUs) are increasingly adopted to run general purpose applications in several domains, programming such many-core architectures is a challenging task [106, 199]. Even more challenging is efficiently tuning applications to fully take advantage of the GPU architectural configuration. Bottlenecks of a GPU application such as high thread divergence or poor memory coalescing have a different impact on the overall performance depending on which GPU device the application is run [247].

Different profiling tools (e.g., CUDA nvprof, AMD APP) have been proposed to help programmers in the application development and analysis. Nevertheless, interpreting the large number of metrics and measurements they provide to improve the application performance is often difficult or even prohibitive for inexperienced programmers.

This Section proposes a comprehensive and accurate performance model for GPU architectures, which aims at supporting programmers in the development and

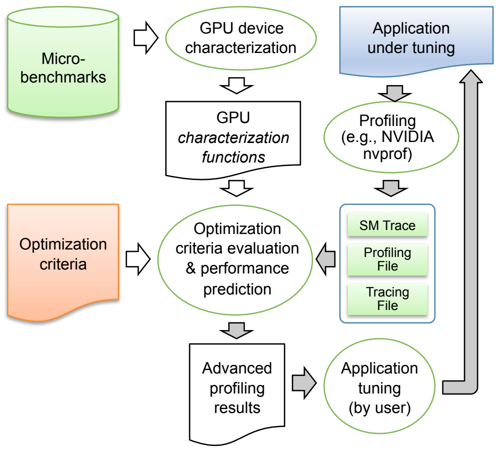

Micro-benchmarks
GPU device characterization
Application under tuning
GPU *characterization functions*
Profiling (e.g., NVIDIA nvprof)
Optimization criteria
Optimization criteria evaluation & performance prediction
SM Trace
Profiling File
Tracing File
Advanced profiling results
Application tuning (by user)

FIG. 13.1: *Overview of proposed model components and application.*

tuning of GPU applications. The model relies on two concepts, *microbenchmarks* and *optimization criteria*. The microbenchmarks consist of specialized chunks of GPU code that have been developed to (i) exercise specific functional components of the device (e.g., arithmetic instruction units, shared memory, cache, DRAM, etc.) and (ii) measure the actual characteristics of such components (i.e., throughput, latency, delay). The result of the measure allows the model to weigh the performance prediction by considering the GPU architecture configuration.

The optimization criteria aim at quantitatively expressing the quality of a given application to exploit the potential of a specific GPU device (e.g., strong coalescence in memory accesses) as well as identifying causes of performance bottlenecks (e.g., high thread divergence, workload unbalancing). At the same time, they aim at guiding the application developer during the tuning activity through understandable hints, by selectively pointing out the causes of such bottlenecks.

Figure 13.1 shows an overview of the proposed model application for tuning a GPU application. First, the microbenchmarks are run on the GPU device to extrapolate the *characterization functions*, i.e., dynamic characteristics of the functional components of the device. Then, the application under tuning is profiled through a standard profiling tool and the resulting information is combined with the characterization functions to measure how much the given application satisfies the optimization criteria. The resulting information provides the actual quality level and the potential improvement of each optimization criterion, the impact of each improvement on the overall application performance, and the overall potential performance of the application under tuning. The model allows the flow (underlined by grey arrows in Figure 13.1) to be iterated for incremental tuning of the application.

The Section presents the results obtained by applying the proposed model for tuning different GPU applications, by underlining how the advanced profiling results have been effectively used to focus the tuning effort in specific code optimizations.

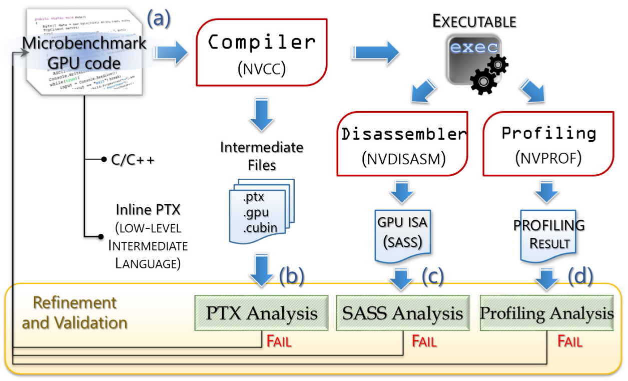


FIG. 13.2: *Microbenchmark development. (a) Code writing, (b) Compilation and PTX analysis, (c) Disassembling and GPU ISA Analisys, (d) Profiling analysis.*

The work is organized as follows. Section 13.2 presents the microbenchmark concept and how they have been developed to support the proposed model. Section 13.3 presents the optimization criteria definition and evaluation. Section 13.4 explains how all the information are combined to predict the application performance, while Section 13.5 reports the experimental results. Section 13.6 is devoted to the concluding remarks.

13.2 Microbenchmarks

Each microbenchmark is developed with the aim of satisfying the following properties:

P1: Stressing capability. The microbenchmark applies heavy and extensive workloads to the selected functional component. This allows reaching the fully workloaded steady state of the component and measuring the real (vs. theoretical) peak performance, while minimizing any side effect that may incur during the measurement, as described in Section 13.2.1. This includes measuring the real latency of memory accesses for each memory level (i.e., registers, shared, L1 cache, L2 cache, and global memory).

P2: Intensity variability. The microbenchmark must exercise the functional component with different intensity. This allows predicting the effect of improving an optimization criterion on the application performance (which is generally not linear), as explained in Section 13.2.2.

P3: Selectivity. The microbenchmark exercises, as much as possible, only a specific functional component of the GPU device. This allows us to selectively associate the microbenchmark to a specific optimization criterion, as described in Section 13.3.

P4: Portability. The microbenchmark is developed independently from any model or configuration of GPU architecture.

13.2.1 Microbenchmark Development

Figure 13.2 shows the design process of a microbenchmark. The microbenchmark code is written with the aim of satisfying the four properties presented above. Since the compiler may optimize such a code (e.g., dead code elimination, code-block reordering etc.) and, through the consequent side effects it may elude the target properties, the code is checked and refined at different steps along the compilation process.

First, the code is written by combining CUDA C/C++ and inline PTX [205] languages (Figure 13.2(a)). The PTX (intermediate) assembly statements allow the code to be compilable for any device model (property $P4$) and, at the same time, to prevent *high-level* compiler optimizations. As an example, Figure 13.3 shows the microbenchmark code developed to measure the peak throughput of an integer arithmetic operation (i.e., `add`) through a long sequence of the arithmetic instructions with no interrupt or intermediate operation. The code implements dynamic value assignments to registers (see rows 1 and 2 in the upper side of the figure) to avoid the *constant propagation* optimization by the compiler[1]. The code also adopts *recursive* and *template-based* metaprogramming. This allows generating an arbitrarily long sequence of arithmetic instructions ($8,191 \times N$ `add` instructions in the example[2]).

As a second level checking, the code is compiled to generate both the intermediate file (Figure 13.2(b)) and the executable binary file. The intermediate PTX file is analysed to verify whether the target properties still hold after the higher-level compilation step. If not, the microbenchmark code undergoes a refinement iteration. Once verified, the code undergoes a more accurate check, whereby the executable binary file is disassembled in the native ISA code, called SASS (Shader ASSembly), as shown in Figure 13.2(c). This allows checking the properties after the lower-level compilation step.

Finally, the microbenchmark is validated through the profiler (Figure 13.2(d)) to ensure that it exercises only the target functional component.

13.2.2 GPU Device Characterization through Microbenchmarks

Similarly to the example of Figure 13.3, the developed microbenchmarks are applied to measure the peak performance of all arithmetic operations and of the memory at different hierarchy levels. In particular, they measure the maximum arithmetic instruction throughput of integer operations (`add`, `mul`, comparison, bitwise, etc.), simple single precision floating-point operations (`add`, `mul`, etc.), complex single precision floating-point operations (`sin`, `rcp`, etc.) and double precision floating-point operations.

Microbenchmarks are also applied to exercise the functional components with different intensity. As an example, the shared memory throughput is analysed

[1] Static value assignments to registers are generally solved and substituted by the compiler optimizations through inlining operations.

[2] In the example, 8,191 is the maximum number of unrolling iterations the pragma `unroll` supports. After that, the compiler would insert control statements for the loop. Such a limitation is overcome through recursive calls (row 5 of the template in Figure 13.2).

```
__global__ Add_throughput()
int R1 = clock();   // assign dynamic values to R1,R2 to␣
int R2 = clock();   // avoid constant propagation
int startTimer = clock();
Computation<N>(R1, R2);   // call the function N times
int endTimer = clock();
```

```
template<int N>()   // template metaprogramming
__device__ __forceinline__ Computation(int R1, int R2)
#pragma unroll 8191   // maximum allowed unrolling
for (int i = 0; i < 8191; i++) do
    asm volatile("add.s32 : "=r"(R1) : "r"(R1), "r"(R2));
end   // volatile: prevent ptx compiler optimization
Computation<N-1>(R1, R2);   // recursive call
```

FIG. 13.3: *Example of microbenchmark code. The code aims at measuring the maximum instruction throughput of the* `add` *operation.*

by running a microbenchmark that generates a different amount of bank conflicts, from zero to the maximum value, and measures the corresponding access times. The effect of the bank conflicts over the access time is then represented by a *characterization function* through a sampling, quantization, and interpolation process [48]. The characterization function strongly depends on the device architecture.

Overall, in the proposed model, microbenchmarks are applied to extrapolate:

- $\mathcal{F}_{Divergence}$ as the function that characterizes the effect of the thread divergence on performance. It is obtained through a microbenchmark that incrementally increases the percentage of control statements in the code.
- $\mathcal{F}_{DRAMThr}$ as the function that characterizes the effect of the (under)utilization of the global memory bandwidth on performance. It is obtained through a microbenchmark that incrementally exercises the memory bus through different amount of exchanged data.
- $\mathcal{F}_{ARITHThr}$ as the function that characterizes the effect of the (under)utilization of the arithmetic units on performance.
- $\mathcal{F}_{SHMEM}$ as explained in the example above.

The characterization functions are used as parameters in the evaluation of the optimization criteria targeting the divergence, the throughput/occupancy, and the shared memory efficiency, respectively, as explained in Section 13.3.7.

Finally, microbenchmarks have been defined to calculate the percentage of use of each functional components (i.e., shared memory, L1 and L2 caches, DRAM, arithmetic units) during an application run. They are used to calculate the application potential speedups, as explained in Section 13.4.

13.3 Optimization Criteria

The optimization criteria are defined to cover all the crucial properties a GPU application should satisfy to exploit the full potential of the GPU device. We consider the properties adopted in [75, 90, 127, 148, 247] concerning divergence, memory coalescing, and load balancing. In addition, we define optimization criteria to cover synchronization issues. Differently from the literature, the proposed criteria are more accurate (i.e., fine-grained) to evaluate such properties. All the criteria are defined in terms of events and static information, which are all provided by any standard GPU profiler. Each criterion value is expressed in the range [0, 1], where 0 represents the worst and 1 represents the best evaluation of such an optimization.

13.3.1 Host Synchronization

Many complex parallel applications organize the compute-intensive work into several functions offloaded to GPUs through host-side kernel calls. Depending on the code complexity and on the workflow scheduling, this mechanism may involve significant overhead that can compromise the overall application performance. The host synchronization criterion aims at evaluating the amount of time spent to coordinate the kernel calls. It is defined as follows:

$$\textsc{HostSync} = \frac{\sum_{\mathrm{i}=1}^{\mathrm{N}} \mathrm{KernelExeTime_i}}{\mathrm{KernelStart}_N + \mathrm{KernelExeTime}_N - \mathrm{KernelStart}_1}$$

where N is the number of kernels in which the application has been organized, $KernelExeTime_i$ is the real execution time of kernel i on the device, and $KernelStart_i$ is the clock time in which kernel i starts executing.

This criterion helps programmers to understand if the overall application speedup is bounded by an excessive host synchronization activity. Merging different kernels, using inter-block synchronization [281] or reducing small memory transfers improve the quality value of this criterion.

13.3.2 Device Synchronization

In GPU computing, the synchronizations of threads in blocks are one of the main causes of idle state and, thus, they strongly impact on the application performance. Beside introducing overhead in the kernel execution, they also limit the efficiency of the multiprocessors in the warp scheduling activity. This criterion gives a quality value of a kernel by measuring the total time spent by the kernel for synchronizing thread blocks:

$$\textsc{DeviceSync} = 1 - \mathrm{StallSync}$$

where $StallSync$ represents the percentage of the GPU time spent in synchronization stalls over the total number of stalls. $StallSync$ depends on the load balancing among threads as well as the number of synchronization points (i.e. thread barriers) in the kernel.

13.3.3 Thread Divergence

Branch conditions that lead threads of the same warp to execute different paths (i.e., thread divergence) are one of the main causes of inefficiency of a GPU kernel. This criterion evaluates the thread divergence of a kernel as follows:

$$\textsc{Divergence} = \frac{\#\text{ExeInstructions}}{\#\text{PotExeInstructions}} \times \mathcal{F}_{\text{Divergence}}$$

where $\#ExeInstructions$ represents the total number of instructions executed by the threads of a warp and $\#PotExeInstructions$ represents the total number of instructions potentially executable by the threads of a warp. The final value is calculated as the average over all warps run by the kernel. The value is weighted by the characterization function $\mathcal{F}_{Divergence}$ presented in Section 13.2.2.

13.3.4 Warp Load Balancing

This criterion expresses how well the workload is uniformly distributed over the cores of each single SM:

$$\textsc{LoadBalanc}_{\text{WARP}} = \frac{\left(\frac{\text{TotActiveWarps}}{\text{TotActiveCycles}}\right)}{\left(\frac{\text{BLOCKSIZE}}{32}\right) \cdot \#\text{Blocks\_per\_SM}}$$

where $TotActiveCycles$ represents the total number of clock cycles in which the single SMs are not in idle state. The formula takes into account the number of active warps at each clock cycle, and it adds them to the total counter $TotActiveWarps$. The denominator represents the theoretical maximum occupancy of the SMs in terms of number of warps. It is calculated by considering the block size and the number of blocks mapped to each single SM.

13.3.5 Streaming Multiprocessor (SM) Load Balancing

Besides the load balancing on each single SM, the model evaluates the load balancing at SM level. The SM load balancing criterion is defined as follows:

$$\textsc{LoadBalanc}_{SM} = 1 - \frac{\max\limits_{\text{SM}}(\text{TotActiveCycles}) - \text{AvgCycles}}{\max\limits_{\text{SM}}(\text{TotActiveCycles})}$$

where
$$\text{AvgCycles} = \frac{\sum\limits_{SM} \text{TotActiveCycles}}{|\text{SM}|}$$

13.3.6 L1/L2 Granularity

GPU applications require optimized data access patterns and properly aligned data structures to achieve high memory bandwidths. In particular, efficient applications hide the latency of memory accesses by combining multiple memory accesses into single *transactions* that match the granularity (i.e., the cache line size) of the memory space[3]. The proposed performance model includes two complementary criteria to describe the quality of memory access patterns:

[3] This concept applied to the L1 cache is also known as *memory coalescing.*

$$\text{L1\_G}\textsc{ranularity} = \frac{|\#\text{L1\_transactions}| \cdot 128}{\sum\limits_{\text{T}\in\{\text{Mem\_instr}\}} |\text{Mem\_instr}_\text{T}| \cdot \text{size}_\text{T}}$$

$$\text{L2\_G}\textsc{ranularity} = \frac{|\#\text{L2\_transactions}| \cdot 32}{\sum\limits_{\text{T}\in\{\text{Mem\_instr}\}} |\text{Mem\_instr}_\text{T}| \cdot \text{size}_\text{T}}$$

The criteria take into account the number of actual transactions towards the L1(L2) memory, the cache line size (128 Bytes for L1, 32 Bytes for L2), the total number of memory instructions (`load` and `store`) to access the global memory, and the size of their accesses $size_T$ (1/2/4/8/16 Bytes).

13.3.7 Shared Memory Efficiency

This criterion measures the kernel efficacy to exploit the *data locality* concept through the on-chip shared memory. The shared memory allows high memory bandwidth for concurrent accesses, but it requires appropriate access patterns to achieve the full efficiency. On the other hand, an excessive and disorganized use of the shared memory leads to bank conflicts, which involve the memory instructions to be re-executed thus serializing the thread execution flow. This optimization criterion is defined as follows:

$$\text{S}\textsc{h}\text{M}\textsc{em}\text{E}\textsc{fficiency} = \frac{\#\text{SharedLoadTrans} + \#\text{SharedStoreTrans}}{\#\text{SharedLoadAcc} + \#\text{SharedStoreAcc}} \times \times \digamma_{SHMEM}$$

The formula is defined in terms of total number of transactions towards shared memory for both load and store operations over the total number of accesses in shared memory for load and store instructions (which includes the re-executed memory instructions due to bank conflicts). It is weighted by the shared memory characterization function ($\digamma_{SHMEM}$) presented in Section 13.2.2

13.3.8 Throughput/Occupancy

The *Throughput/Occupancy* criterion is defined as follows:

$$\begin{array}{l}\text{T}\textsc{hroughput}/\\ \text{O}\textsc{ccupancy}\end{array} = \begin{cases} 1 & \text{if } MemThr \approx 1 \\ 1-(1-\text{occ}) \cdot \text{M}\textsc{em}\text{T}\textsc{hr} & otherwise \end{cases}$$

where the occupancy value (*occ*) depends on the kernel configuration (i.e., block size, grid size, and amount of shared memory allocated for the kernel variables), and

$$\text{M}\textsc{em}\text{T}\textsc{hr} = \frac{\text{AchivedThroughput}}{\text{TheoreticalPeakThroughput} \times \digamma_{\text{DRAMThr}}}$$

The theoretical peak throughput is weighted through the $\digamma_{DRAMThr}$ characterisation function. If the memory throughput value is close to 1, the throughput/occupancy criterion cannot be further improved. Otherwise, the throughput/occupancy criterion is calculated as the potential improvement of the memory throughput metric by using all device threads $(1 - occ)$. This criterion is particularly useful in such applications that do not achieve the theoretical occupancy of the device. As proved in [271], a high theoretical GPU occupancy is not necessary to reach the peak performance. In contrast, a high theoretical occupancy and a low value of the throughput/occupancy criterion suggests optimizing the application kernel through a re-configuration to increase the occupancy.

13.4 Performance Prediction

Improving any of the optimization criteria presented in Section 13.3 impacts on the overall application speedup. A speedup increasing is proportional to the criterion improvement. The potential speedup of the *host synchronization, divergence, warp and SM load balancing* and *throughput/occupancy* criteria are defined as follows:

$$\begin{aligned}
\textsc{HostSync}^{\text{SP}} &= \frac{1}{\textsc{HostSync}}\\
\textsc{Divergence}^{\text{SP}} &= \frac{1}{\textsc{Divergence}}\\
\textsc{LoadBalanc}_{WARP}{}^{\text{SP}} &= \frac{1}{\textsc{LoadBalanc}_{WARP}}\\
\textsc{LoadBalanc}_{SM}{}^{\text{SP}} &= \frac{1}{\textsc{LoadBalanc}_{SM}}\\
\begin{matrix}\textsc{Throughput}/\\ \textsc{Occupancy}^{\text{SP}}\end{matrix} &= \frac{1}{\textsc{Throughput}/\textsc{Occupancy}}
\end{aligned}$$

The potential speedup of the *device synchronization* criterion also depends on the fraction of time spent in stall state over the total kernel time:

$$\textsc{DeviceSync}^{SP} = \left(1 - \frac{\text{TotActiveWarps }/|\text{Warps}|}{\text{CLK\_cycles}}\right) \cdot \text{StallSync}$$

$|Warps|$ represents the maximum number of thread warps of the device, while CLK_cycles represents the total number of GPU clock cycles elapsed to execute the kernel. The value in the round brackets represents the overall percentage of inactivity of the GPU warps (i.e., warps in stall state).

The potential speedup definition of *L1,L2 granularity* and *shared memory efficiency* criteria also depends on the percentage of time the application uses the L1, L2, and shared memory, respectively, over the total execution time:

$$\begin{aligned}
\textsc{L1\_Granularity}^{\text{SP}} &= \frac{1}{\textsc{L1\_Granularity}} \cdot L1^{\%}\\
\textsc{L2\_Granularity}^{\text{SP}} &= \frac{1}{\textsc{L2\_Granularity}} \cdot L2^{\%}\\
\textsc{ShMemEfficiency}^{\text{SP}} &= \frac{1}{\textsc{ShMemEfficiency}} \cdot \text{ShMem}^{\%}
\end{aligned}$$

$L1^{\%}$, $L2^{\%}$, and $ShMem^{\%}$ are evaluated as follows. The model classifies the application activity in terms of DRAM accesses, cache accesses, shared memory accesses, arithmetic instructions, and idle states. The profiler provides the accurate evaluation of the idle states, the exact amount of memory transactions for each memory typology, and the number of arithmetic instructions. Twelve microbenchmarks (eight for memory accesses considering both load and store operations and four for arithmetic instructions) allow estimating the memory latencies and the arithmetic instruction throughputs. $L1^{\%}$, $L2^{\%}$, and $ShMem^{\%}$ are calculated by comparing the sum of such latencies spent in a specific memory level with the total cycles elapsed during the kernel execution.

Finally, the overall potential speedup of the application is defined as follows:

$$\text{PotentialSpeedup} = \begin{cases} \dfrac{1}{\textsc{MemThr}} & \text{if memory-bounded} \\ \dfrac{1}{\textsc{ArithThr}} & \text{if compute-bounded} \end{cases}$$

where $MemThr$ has been defined in Section 13.3.8, while $ArithThr$ is defined as follows:

$$\textsc{ArithThr} = \frac{AchivedThroughput}{TheoreticalPeakThroughput \times F_{ARITHThr}}$$

The formula expresses the potential speedup of the application under tuning as the inverse of the memory throughput or the arithmetic throughput depending on whether the application is memory-bounded or compute-bounded. Such an information is provided by the profiler.

13.5 Experimental Results

The proposed performance model has been applied for tuning and improving the performance of three different CUDA applications, *reduction*, *matrix transpose*, and *BFS* for an NVIDIA (Kepler) GEFORCE GTX 780 device. The experiments have been run on such a device with CUDA Toolkit 7.0, AMD Phenom II X6 1055T (3GHz) host processor, Debian 3.2.60 operating system, and NVIDIA `nvprof` profiler.

13.5.1 Case study 1: Parallel Reduction

Given a vector of data $\{x_1, x_2, \ldots, x_n\}$, the reduction applies an operator $\oplus$ to all elements and returns a single element $R = x_1 \oplus x_2 \oplus \ldots \oplus x_n$. We analysed the reduction implementation provided in [119], and we applied two tuning iterations with the proposed model.

Figure 13.4 shows the results. In the left-most side, the columns represent the optimization value [0-1] for each criterion at each tuning iteration. The dot in a column represents the potential contribution of an improvement of such a criterion in the predicted overall speedup. In the right-most side, the figure reports the overall potential speedup of the application (see Section 13.4), which is calculated for the original code (Version1) as well as for the two optimized versions of the code. The *L1* and *L2 granularity* criteria has the same value and are thus reported in Figure 13.4 as a single item.

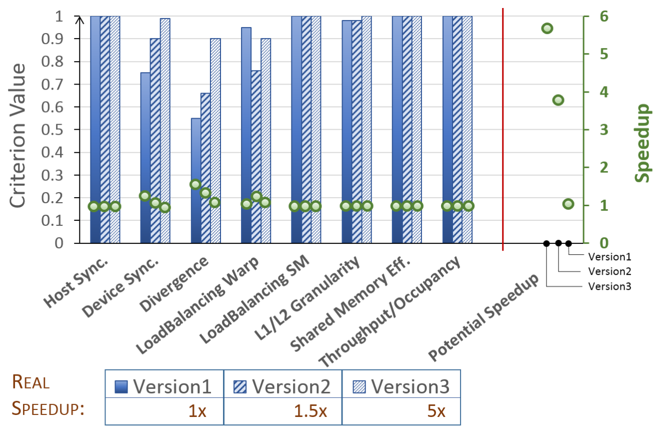


FIG. 13.4: *Experimental results of case study 1.*

In the analysis of the original code (Version1), the model predicted a potential speedup of 5.8x. The criteria values underline that the application bottlenecks are mainly due to high thread divergence, inadequate synchronization of GPU threads, and unbalancing at warp level. We first optimized the code by focusing on synchronization. We organized the threads by using the warp-centric method proposed by [125], which allowed us to reduce the number of barriers from log(BLOCKSIZE) to one.

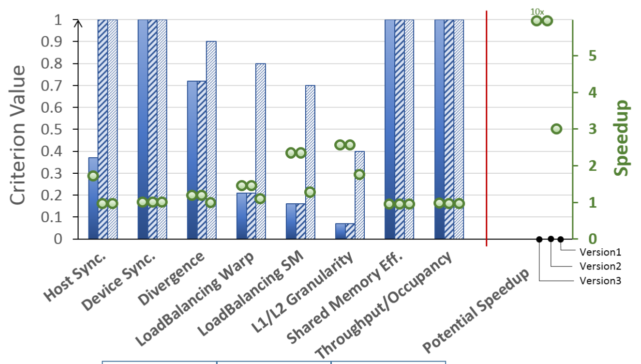
The analysis of such a first optimization (Version2) confirmed the improvement on the thread synchronization (see *device synchronization* criterion), which influenced (positively) the divergence level of threads. Nevertheless, the results underlined a slight improvement of the *memory troughput* metrics, which motivates the marginal increasing of the Version2 speedup (1.48x). On the other hand, the model predicted a further potential improvement of the speedup up to 3.9x, by suggesting to optimize the divergence aspect.

We addressed the divergence issue in the second optimization (Version3) by increasing the number of elements computed by a single thread. We also applied instruction-level parallelism techniques to increase the arithmetic throughput. The analysis of Version3 shows that all the optimization criterion values are close to the maximum and the potential speedup is close to 1x. These values suggest that any further optimization on the considered criteria on the adopted GPU device would not improve the current speedup. We measured the Version 3 speedup equals to 4.98x, while the potential speedup predicted by the model was 5.8x.

13.5.2 Case study 2: BFS

In parallel computing, BFS is one of the most representative irregular application that involves thread divergence, workload imbalance, and poorly coalesced memory accesses. We analysed the BFS implementation provided in [59].

Figure 13.5 shows the results obtained by applying the proposed performance model for two optimization steps of the code. In the original imlementation (Version1), the results indicate many different causes of performance bottlenecks and a



Real Speedup:	■ Version1	▨ Version2	▨ Version3
	1x	1.6x	4.1x

FIG. 13.5: *Experimental results of case study 2.*

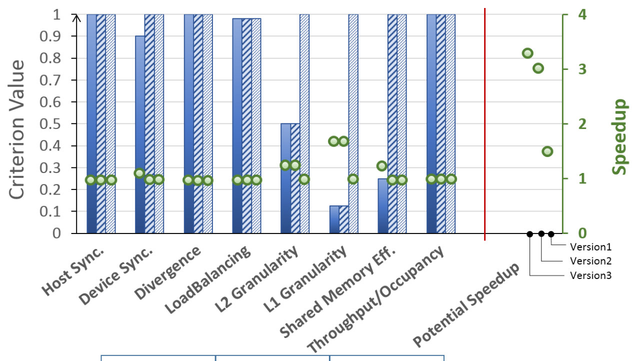
potential speedup up to 10x. We first focused in the low value of the *host synchronization* criterion, which was due to a high number of kernel calls. We optimized the code by enabling the *inter-block synchronization* [281], which allows the device and the host execution to be completely separated and, thus, the application to be organized into one single kernel.

The analysis of such a first optimization (Version2) confirmed the total elimination of the host synchronization overhead thanks to the single kernel implementation. This allowed reaching a fair speedup (2.44x). On the other hand, the results show that the optimization didn't impact on the other criterion values. The results underline that the code suffers from *L1/L2 granularity*, for which the criterion value is the lowest and the potential contribution ($\approx 2.5x$) in the overall speedup is the highest. Nevertheless, to the best of my knowledge, we could not have removed such a bottleneck, which is mainly due to the irregular data structures on which the implemented algorithm works. We focused on the low values of the warp/SM load balancing criteria, which suggest to better organize the GPU thread allocations. We optimized the code (Version3) by re-arranging the threads in groups with the aim of cooperative visiting single vertices instead of sets of vertices. Version3 provides a speedup of 4.1x w.r.t. the original code. The analysis of Version3 underlines that improving the *L1/L2 granularity* would be the main important optimization to double the speedup and to reach the predicted 10x value.

13.5.3 Case study 3: Matrix Transpose

We analysed the matrix transpose implementation presented in [227], which is characterized by data tiling in shared memory and thread organization in 2D hierarchical grids and blocks.

Figure 13.6 shows the results. The original code already provides values close to the maximum for the *host* and *device synchronizations*, *divergence*, and *Warp/SM load balancing* criteria. For all the other criteria, even though they have very low values (between 0.1 and 0.5), the model predicts marginal potential speedups.



Real Speedup:	■ Version1	▨ Version2	▨ Version3
	1x	1.1x	2x

FIG. 13.6: *Experimental results of case study 3.*

This is due to the fact that the application algorithm relies on very regular and independent tasks. This justifies the limited potential overall speedup (≈3.3x) predicted by the model.

In the first optimization (Version2), we focused on improving the shared memory bank conflicts (*shared memory efficiency* criterion) by applying the *memory padding technique* [227]. As expected, since such a technique has more impact on the NVIDIA Fermi than on the NVIDIA Kepler architecture [227] the gained speedup is marginal.

In the second optimization (Version3) we taken into account the memory access patterns to improve the *L1* and *L2 granularity* criteria. Their low values suggest that the memory accesses do not match the granularity of the respective caches, thus involving a waste of the memory bandwidth. We fully optimized both the criteria by simply re-organizing the thread block configuration and by resizing the memory *tiles* (as shown by the third columns of the three criteria in Figure 13.6). The Version3 implementation provides a speedup of 3x against the 3.3x predicted by the model.

13.6 Remarks

This Section presented a fine-grained performance model for GPU architectures. It relies on microbenchmarks to characterize the GPU device and on several application criteria to measure the implementation quality, to give interpretable hints, and to accurately calculate potential performance. The Section presented the results obtained by applying the proposed model for tuning different GPU applications, by underlining how the advanced profiling results have been effectively used to focus the tuning effort in specific code optimizations.

14

Power/Performance/Energy Microbenchmarking

GPU-accelerated applications are becoming increasingly common in high-performance computing as well as in low-power heterogeneous embedded systems. Nevertheless, GPU programming is a challenging task, especially if a GPU application has to be tuned to fully take advantage of the GPU architectural configuration. Even more challenging is the application tuning by considering power and energy consumption, which have emerged as first-order design constraints in addition to performance. Solving bottlenecks of a GPU application such as high thread divergence or poor memory coalescing have a different impact on the overall performance, power and energy consumption. Such an impact also depends on the GPU device on which the application is run. This Section presents a suite of microbenchmarks, which are specialized chunks of GPU code that exercise specific device components (e.g., arithmetic instruction units, shared memory, cache, DRAM, etc.) and that provide the actual characteristics of such components in terms of throughput, power, and energy consumption. The suite aims at enriching standard profiler information and guiding the GPU application tuning on a specific GPU architecture by considering all three design constraints (i.e., power, performance, energy consumption). The Section presents the results obtained by applying the proposed suite to characterize two different GPU devices and to understand how application tuning may impact differently on them.

14.1 Introduction

With the growth of computational power and programmability, Graphic Processing Units (GPUs) have become increasingly used as general-purpose accelerators. They not only provide high peak performance, but also excellent energy efficiency [196]. As a consequence, besides supercomputers, GPUs are quickly spreading in low-power and mobile devices like smartphones. NVIDIA Tegra X1 [8] and Qualcomm Snapdragon [11] are some among the several system-on-chip examples available in the mobile market that integrate GPUs with other processing units (i.e., CPUs, FPGAs, DSPs).

On the other hand, the large number of operating hardware resources (e.g., cores and register files) employed in GPUs to support the massive parallelism leads

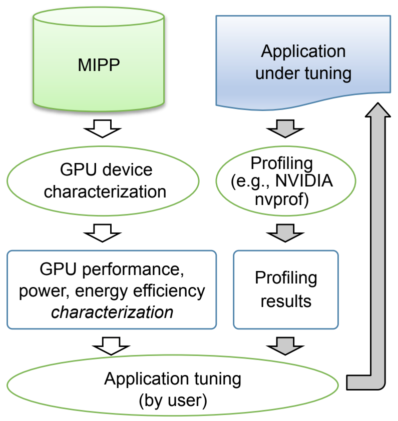

MIPP
Application under tuning
GPU device characterization
Profiling (e.g., NVIDIA nvprof)
GPU performance, power, energy efficiency *characterization*
Profiling results
Application tuning (by user)

FIG. 14.1: *Overview of application tuning through MIPP.*

to a significant power consumption. The elevated levels of power consumption have a sensible impact on such many-core device reliability, aging, economic feasibility, performance scaling and deployment into a wide range of application domains. Different techniques have been proposed to manage such high levels of power dissipation and to continue scaling performance and energy. They include approaches based on dynamic voltage/frequency scaling (DVFS) [147], CPU-GPU work division [173], architecture-level/runtime adaptations [274], dynamic resource allocation [127], and application-specific (i.e., programming-level) optimizations [288]. Particularly in this last category, it has been observed that source-code-level transformations and application specific optimizations can significantly affect the GPU resource utilization, performance, and energy efficiency [247].

In this context, even though profiling tools (e.g., CUDA nvprof, AMD APP) exist to help programmers in the application analysis and optimization, they do not provide a complete view of the GPU features (especially on power consumption and energy efficiency) neither they provide a correlation among these design constraints.

This Section presents *MIPP*, a suite of microbenchmarks that aims at characterizing a GPU device in terms of performance, power, and energy consumption. In particular, it aims at understanding how application bottlenecks involving selected functional components or underutilization of them can affect the code performance, power consumption, and energy efficiency on the given device. The functional components include arithmetic instruction units, memories (shared, cache, DRAM, constant), scheduling and synchronization components. Fig. 14.1 shows how the suite can be applied during an application tuning. First, the microbenchmarks are run on the GPU device to *characterize* the device in terms of performance, power and energy over its main functional components. Then, the application under tuning is profiled by using a standard profiling tool. The profiling results provide information on bottlenecks and underutilization of functional components. The microbenchmark results extend such an information by quantitatively showing how such bottlenecks and underutilization affect performance, power and energy. The proposed model allows the flow (underlined by grey arrows in Fig. 14.1) to be iterated for incremental tuning of the application.

The suite has been applied to characterize two different GPU devices (i.e., an NVIDIA Kepler GTX660 and a low power embedded system NVIDIA Jetson TK1). The results show how the same code optimizations have a different impact on the design constraints on the two GPU architectures.

The work is organized as follows. Section 14.2 presents the microbenchmark suite. Section 14.3 reports the experimental results, while Section 14.4 is devoted to concluding remarks.

14.2 The Microbenchmark Suite

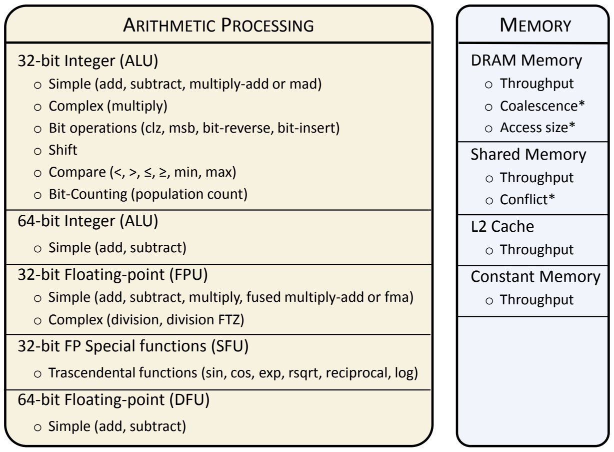

Arithmetic Processing
32-bit Integer (ALU)
Simple (add, subtract, multiply-add or mad)
Complex (multiply)
Bit operations (clz, msb, bit-reverse, bit-insert)
Shift
Compare (<, >, ≤, ≥, min, max)
Bit-Counting (population count)
64-bit Integer (ALU)
Simple (add, subtract)
32-bit Floating-point (FPU)
Simple (add, subtract, multiply, fused multiply-add or fma)
Complex (division, division FTZ)
32-bit FP Special functions (SFU)
Trascendental functions (sin, cos, exp, rsqrt, reciprocal, log)
64-bit Floating-point (DFU)
Simple (add, subtract)
Memory
DRAM Memory
Throughput
Coalescence*
Access size*
Shared Memory
Throughput
Conflict*
L2 Cache
Throughput
Constant Memory
Throughput

FIG. 14.2: *Microbenchmark Classes*

We developed a suite of microbenchmarks to selectively study the behaviour of a wide range of GPU functional components. Fig. 14.2 gives an overview of such a microbenchmark suite by reporting, for each microbenchmark, the exercised GPU component, the involved specific instructions, and the considered features.

A microbenchmark consists of a GPU kernel code that exercises a specific functional component of the architecture and whose instructions can be evaluated at a clock-cycle accuracy. The generic structure of the microbenchmark main procedure consists of a long sequence of one or more selected instructions (e.g., arithmetic instructions, memory accesses) that executes without any interference deriving from other instructions. The microbenchmarks have been implemented to stress only a specific functional component at a time, while affecting the others as little as possible to obtain reliable and accurate feedback.

The microbenchmark code is written by combining the CUDA C/C++ language with inline intermediate assembly to avoid compiler side effects that may elude the target properties. The parallel thread execution (PTX) is a GPU machine-independent language that allows expressing general purpose computation through virtual ISA. We exploited the PTX language to force a specific operation on a data type, to avoid compiler optimizations, to prevent caching/local-storage mechanisms, and to drive the memory accesses. We adopted several arrangements to preserve the code functionality. As an example, registers are initialized with dynamic values to avoid constant propagation in arithmetic benchmarks.

Finally, to perform an extensive computation, we applied template metaprogramming and nested loops. Such programming techniques are required to prevent compiler optimizations (e.g., loop collapsing and dead code elimination). The same full control on the compiler cannot be obtained by simply handling the compilation flags, since they apply only to arithmetic instruction optimizations (such as floating-point precision and floating-point multiply-add enabling. The code controls the intensity variability, the amount of computation, and other aspects through parameterized procedures.

Each microbenchmark run returns information like *execution time*, *actual throughput* (to compare with the theoretical throughput from the device specifications), *average* and *max power consumption*, *energy consumption* and *energy efficiency*. Some microbenchmarks (marked with "$*$" in Fig. 14.2) are also applied to exercise functional components with different intensity. As an example, a microbenchmark allows analysing the shared memory throughput by generating a different amount of bank conflicts, from zero to the maximum value, and by measuring the corresponding access time.

The microbenchmarks provide a quantitative model of the target GPU architecture based on performance and power, and provide important guidelines for the application optimization. As shown in Fig. 14.2, the microbenchmarks are grouped into two classes: *Arithmetic processing* and *memory hierarchy.*

14.2.1 Arithmetic processing benchmarks

This class of microbenchmarks targets the complete set of arithmetic instructions natively supported by the GPU, by distinguishing between integer and floating point over 32 and 64-bit word sizes.

The benchmarks perform a long sequence of instructions to stress the ALU components. All PTX instructions of arithmetic benchmarks have a direct translation into the native ISA, called SASS (Shared ASSembly), except 64-bit integer operations and floating-point divisions that are compiled into multiple instructions. A SM executes native instructions in one clock cycle, providing a throughput (instructions per clock cycles) limited by the concurrency of the exercised ALU component. Depending on the compute capability of the device and on the architecture, the implementation of non-native instructions may correspond to a different number and type of ISA instructions. Arithmetic benchmarks include also four different types of division operations classified by approximation (IEEE754 Compliance and fast hardware approximation) and normalization (normal and de-normal numbers).

As an example, Fig. 14.3 summarizes the microbenchmark developed to analyse the 32-bit integer arithmetic processing unit (simple `add`). The code implements dynamic value assignments to registers (see rows 1 and 2 in the upper side of the figure) to avoid the *constant propagation* optimization by the compiler[1]. The code

[1] Static value assignments to registers are generally solved and substituted by the compiler optimizations through inlining operations.

```
__global__ Add_throughput()
int R1 = clock();   // assign dynamic values to R1,R2 to ↵
int R2 = clock();   // avoid constant propagation
int startTimer = clock();
Computation<N>(R1, R2);   // call the function N times
int endTimer = clock();
```

```
template<int N>()   // template metaprogramming
__device__ __forceinline__ Computation(int R1, int R2)
#pragma unroll 4096   // maximum allowed unrolling
for (int i = 0; i < 4096; i++) do
    asm volatile("add.s32 : "=r"(R1) : "r"(R1), "r"(R2));
end   // volatile: prevent ptx compiler optimization
Computation<N-1>(R1, R2);   // recursive call
```

FIG. 14.3: *Example of microbenchmark code. The code aims at measuring the maximum instruction throughput of the* `add` *operation.*

also adopts *recursive* and *template-based* metaprogramming. This allows generating an arbitrarily long sequence of arithmetic instructions without any control flow instructions. ($4096 \times N$ `add` instructions in the example[2]).

14.2.2 Memory benchmarks

This class of benchmarks focuses on the impact of throughput and access patterns on DRAM, shared, constant, and L2 cache memories. The *DRAM throughput* benchmark executes several global accesses to different memory locations with a stride of 128 bytes between grid threads to avoid L1 coalescing. The *L2 benchmark* repeats a compile-time sequence of store instructions on the same memory address. We use cache modifiers [205] to avoid L1 cache hits that can occur in the store operations. *Shared* and *constant memory* benchmarks consist of a sequence of load/store instructions respectively. In the *coalescing* benchmark, we vary the number of threads within a warp that access to continuous locations. For example, to test the impact of the worst memory access pattern (no coalescence) we apply the same stride of the DRAM throughput benchmark, to evaluate 1/16 of coalescence we divide the warp threads into 16 groups of two threads where each group accesses in different addresses. The *access size* benchmark copies one large array into another multiple times and, in each execution, we vary the data type size.

Fig. 14.4 summarizes the microbenchmark developed to measure the impact of shared memory bank conflicts on the memory access throughput. The procedure first computes, for each thread of a warp, the address offsets of shared memory that can lead to bank conflicts (`line 2` in the upper side of Fig. 14.4, where `LANE_ID` represents the thread id in the warp, and `CONFLICTS` represents the number of conflicts to generate). Fig. 14.5 shows, for example, the offsets generated to lead to no and to one bank conflict. Then, it performs a long sequence of store operations with no interrupt or intermediate operation. The code implements *volatile*

[2] In the example, 4,096 unrolls are a good compromise between loop body replication and template recursion. Over a fixed number of loop unrolling iterations the compiler would insert control statements in the loop to reduce the size of the binary code.

```
   __device__ clock_t devClocks[RESIDENT_WARPS];
   __device__ int devTMP;
   template<int CONFLICTS>
   __global__ SharedMemConflicts()
1: __shared__ volatile int SMem[1024];
2: volatile int* Offset = SMem + LANE_ID * (CONFLICTS+1);
3: clock_t startTimer = clock64();
4: Computation<N>(Offset);// call the function N times
5: clock_t endTimer = clock64();
6: if (LANE_ID == 0) then
7:     devClocks[WARP_ID] = endTimer - startTimer;
8: if (THREAD_ID == 1024) then
9:     devTMP = SMem[0]; // never executed
```

```
   template<int N>   // template metaprogramming
   _device_ _forceinline_ Computation(volatile int* Offset)
1: #pragma unroll 4096
2: for (int i = 0; i < 4096; i++) do
3:     asm volatile("st.volatile.s32 [%0], %1;" : :
4:                 "l"(Offset), "r"(i) : "memory" );
5:            // asm volatile: prevent PTX compiler optimization
6: end
7: Computation<N-1>(Offset);   // recursive call
```

FIG. 14.4: *Example of the microbenchmark code to measure the impact of shared memory bank conflicts*

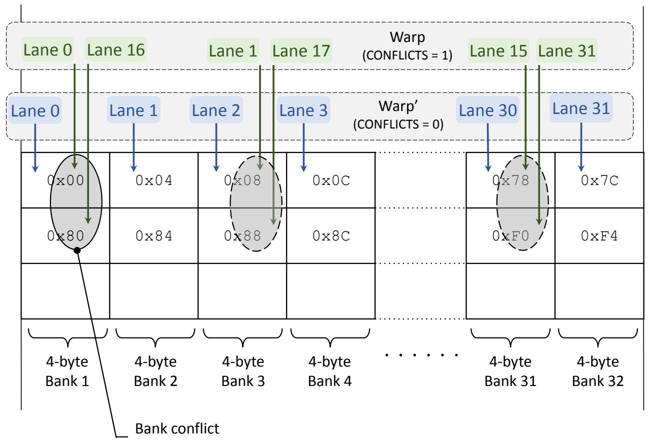


FIG. 14.5: *Example of memory accesses with no and one bank conflict.*

quantifiers (`lines 1,2` in the upper side of Fig. 14.4) to avoid *local-storage* optimization by the compiler. As for the simple `add` example, the code adopts *recursive* and *template-based* metaprogramming to generate an arbitrarily long sequence of arithmetic instructions ($4,096 \times N$ `store` instructions in the example). In the last step (`Line 7` in the upper side of the figure) each warp sends the timing results

	Integer 32-bit						**Integer 64-bit**	**FP 32-bit**		**FP 64-bit**
	Simple	Complex	Pop. Count	Shift	Bit OP.	Compare	Simple	Simple	Special	Simple
Execution Time (ms)	9.6	34.3	34.3	34.3	36.7	9.8	24.5	9.7	36.6	137.0
Spec Throughput (OPs × Cycle) × SM	160	32	32	32	32	160	n.a.	192	32	8
Real Throughput (OPs × Cycle) × SM	126.1	34.7	34.7	34.7	34.3	122.2	51.2	126.0	33.9	8.8
Avg. power (W)	54.4	54.4	53.7	52.5	56.8	54.3	57.6	55.7	60.6	53.3
Max Power (W)	62	59	56	56	60	59	62	62	65	59
Energy (J)	0.1	0.4	0.4	0.4	0.4	0.1	0.3	0.1	0.4	1.4
Energy efficiency (MIPS × Watt) × SM	2,057.1	576.3	583.8	597.0	514.6	2,020.0	760.2	1,990.3	483.9	146.9
nanojoule per instruction	0.1	0.3	0.3	0.3	0.4	0.1	0.2	0.1	0.4	1.3

Table 14.1: *GTX 660 - Characterization with Arithmetic Processing benchmarks.*

to the host through global memory. `Line 8,9` ensure that the result is stored in global variable with a fake write instruction (that is never executed since the higher thread id in a block is 1023) to prevent dead code elimination by the compiler.

Similarly, other microbenchmarks of this class exercise their corresponding functional components with different intensity. This allows characterizing the components behavior under different workloads. For example, the global memory throughput is affected by access pattern that involves a different number of memory transactions or by saturation of the instruction pipeline. Varying the intensity of a microbenchmark allows us to better understand the main factors that affect the performance and the power consumption in real-world applications, where functional components show a wide range of utilization values.

In general, the microbenchmarks have been developed to guarantee enough computation time (i.e., at least of some milliseconds) to overcome the limitation of the sampling frequency in the measurement of the power features ($P_{\max}, \mathrm{P}_{\mathrm{total}}, \mathrm{P}_{\mathrm{avg}}$) and to minimize the RLC effect in the kernel starting/ending phases.

14.3 Experimental Results

We run the suite to characterize two different GPU devices. The first is an NVIDIA Kepler GeForce GTX 660 with CUDA Toolkit 7.5, AMD Phenom II X6 1055T (3GHz) host processor, and Ubuntu 14.04 OS. The second is a Tegra K1 SoC (Kepler architecture) on an NVIDIA Jetson TK1 embedded system, with CUDA Toolkit 6.5, 4-Plus-1 NVIDIA host multi processor (four ARM Cortex-A15 cores on the general cluster and one ARM Cortex-A7 in the low power cluster), and Ubuntu 14.04 OS.

	Integer 32-bit						**Integer 64-bit**	**FP 32-bit**		**FP 64-bit**
	SIMPLE	COMPLEX	POP. COUNT	SHIFT	BIT OP.	COMPARE	SIMPLE	SIMPLE	SPECIAL	SIMPLE
Execution Time (ms)	140.8	483.0	489.9	486.2	515.5	145.1	399.0	143.3	508.5	1,878.8
Spec Throughput (OPs × Cycle) × SM	160	32	32	32	32	160	n.a.	192	32	8
Real Throughput (OPs × Cycle) × SM	123.0	32.1	32.1	32.1	32.1	121.0	40.6	123.1	30.7	8.0
Avg. Power (W)	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.2	3.3	3.2
Max Power (W)	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0
Energy (J)	0.5	1.5	1.6	1.5	1.6	0.5	1.3	0.5	1.7	6.0
Energy efficiency (MIPS × Watt) × SM	2,376.2	703.1	688.3	699.1	658.7	2,318.6	844.5	2,355.2	646.2	180.2
nanojoule per instruction	0.4	1.4	1.5	1.4	1.5	0.4	1.2	0.4	1.5	5.5

TABLE 14.2: *Jetson TK1 - Characterization with Arithmetic Processing benchmarks.*

Performance information has been collected through the `clock64` device instruction for all microbenchmarks to provide high accuracy. The microbenchmark kernels are organized in such a way that each warp stores the starting and the ending clock counters for each execution and sends the difference to the host, which computes the average among all device warps.

Power and energy consumption information has been collected through the *Powermon2* power monitoring device [35]. The device allows measuring the voltage and the current values from different sources at the same time with a frequency of 1024 Hz for every sensors. The GTX 660 requires five 12V pins, three for the pci-express power connectors and two for auxiliary connectors. We used a pci-express *interposer* to isolate the GPU power connectors from the motherboard. The Jetson TK1 requires only a DC barrel connector adapter to enable the power monitoring. We designed specific API and procedures to allow microbenchmarks to communicate and to synchronize with the Powermon device. In particular, we ensured that the kernel calls are synchronized with the first power measurement. Each microbenchmark is repeated ten times with two seconds of idle activity between consecutive executions. The analysis has been performed with the default GPU frequency setting on both devices. To measure, as much accurately as possible, the power consumption of the Jetson TK1 SoC, in which is not possible to physically isolate the GPU (Tegra K1) from the rest of the system, we operated as follows. We implemented the communication with the platform remotely without any additional connected peripherals, we disabled the Linux display manager (lightdm) and the HDMI port from the OS, we physically disabled the general CPU cluster, and we forced the low power CPU cluster to run with the lowest frequency.

Tables 14.1 and 14.2 report the results obtained by running the *Arithmetic Processing* benchmarks on the GTX 660 and TK1, respectively. The benchnmarks, which are organized over columns, consist each one of 10^9 instructions per SM (i.e., consider that the GTX 660 consists of 5 SMs, while the TK1 consists of 1 SM). For each benchmark, the tables report the execution time, the theoretical peak throughput of the corresponding functional unit provided in the device specifications [213] (*Spec Throughput*) and that measured through the proposed benchmark (*Real throughput*). The device specifications do not include the theoretical peak throughput of the Integer 64-bit simple unit since such an operation has not an embedded hardware implementation (it is performed by combining different hardware units). The tables also show the average power, the peak power, the energy consumption for a single SM, the energy efficiency [99], i.e. performance per watt (Million Instructions Per Second per Watt - MIPS/Watt) and the energy consumption for a single instruction in nanojoules.

For both the GTX 660 and TK1, the benchmarks underline that the theoretical peak throughput of several functional units (e.g., complex multiply, population count, shift Integer 32-bit etc.) can be actually reached. The measured peak values often exceed the theoretical values provided by the device specifications. We assume this is due to the fact that the theoretical values refer to the declared compute capability of the GPU rather than actual GPU manufacturing of the vendor. In any case, the difference between the two value is negligible. In contrast, the peak throughput of selected functional units, such as the simple 32-bit either Integer or Floating Point `add` cannot be actually fully exploited. We assume this is due to the actual latency of the fetching subunits, which do not support the throughput of the computation subunits. Even though the two devices are fairly different (desktop-oriented GTX 660, and low-power embedded system TK1), the benchmarks underline they rely on equivalent Kepler SMs, whose peak performance are fully comparable.

Finally, Tables 14.1 and 14.2 report information about power and energy consumption, which are not provided with the device specifications. Power and energy characteristics refer to the whole GPU device and underline the structural characteristics of the two GPU device architectures (5 SMs vs. 1 SM). The tables also show that FP 64-bit operations have a significant impact on the energy consumption of both the devices (15 times higher than Integer/FP 32-bit simple and Integer compare instructions).

Tables 14.3 and 14.4 report the results obtained by running the microbenchmarks of the *memory* class to evaluate the throughput of the different device memories. The results allow understanding how the throughput differs among memories and how it differs between the two devices. As an example, an application running on the TK1 accesses the constant memory 4 times faster than in DRAM. The same application running in the GTX 660 accesses the constant memory 20 times faster than in DRAM. Moreover, the table shows that DRAM accesses strongly affect the average and the max power of GTX 660 and TK1 devices while, the on-chip memories (shared and constant memories) have sightly higher average power than arithmetic instructions.

Figure 14.6a shows the impact of thread *coalescence* in DRAM memory accesses on the GTX 660 performance, power and energy. The figure shows the effect starting from no coalescence (one memory transaction per warp thread access), 1/16 coalescence (one transaction per two warp threads), until FULL coalescence (one

	DRAM	**L2**	**Shared**	**Constant**
Execution Time (ms)	1,170	1,013	220.2	60.8
Real Throughput (OPs per Cycle)	1.0	1.1	5.3	19.6
Avg. power (W)	92.1	71.5	59.2	63.3
Max power (W)	102.0	80.0	62.0	68.0
Energy (J)	107.8	72.5	13.0	3.8
Energy efficiency (10^6**Transactions/Watt**)	10.0	14.8	82.4	279.1
nano Joule per mem. transaction	100.4	67.5	12.1	3.6

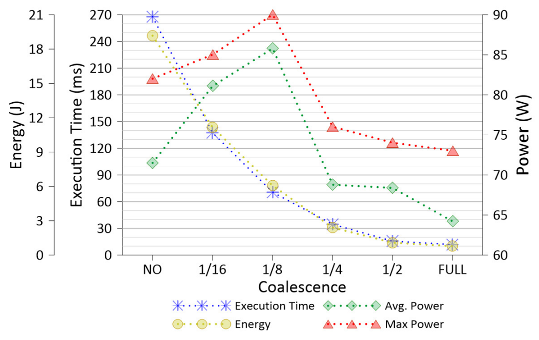
Table 14.3: *GTX 660 - Characteristics of accesses on DRAM, L2, shared and constant memories.*

	DRAM	**L2**	**Shared**	**Constant**
Execution Time (ms)	24,703	22,339	14,915	3,905
Real Throughput (OPs per Cycle)	0.6	0.7	1.0	3.8
Avg. power (W)	4.1	3.8	3.3	3.3
Max power (W)	5.0	5.0	5.0	4.0
Energy (J)	100.4	85.4	49.2	12.9
Energy efficiency (10^6**Transactions/Watt**)	10.7	12.6	21.8	83.3
nano Joule per mem. transaction	93.5	79.5	45.8	12.0

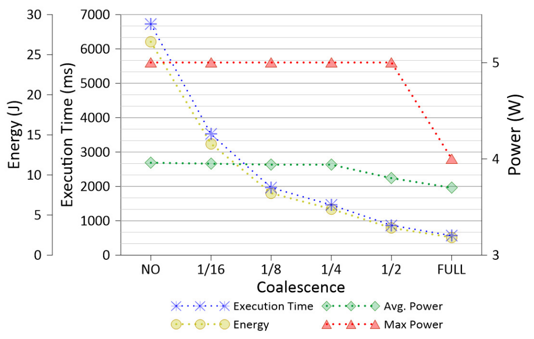
Table 14.4: *Jetson TK1 - Characteristics of accesses on DRAM, L2, shared and constant memories.*

transaction per a whole 32-threads warp). The figure shows how performance and energy are proportional to the reached coalescence. In contrast, max and average power reach the highest values at 1/8 coalescence, and they decrease until FULL coalescence. This is due to the fact that from NO to 1/8, the coalescence is incrementally supported by the 32-Byte L2 memory banks, which saturate at 1/4 coalescence (i.e., each set of 4 transactions per warp, each one 32-Byte large, saturate a 32-Byte L2 bank). From 1/4 on, the coalescence relies on the 128-Byte L1 memory banks. Figure 14.6b reports the same analysis on the TK1, for which the decreasing of the max power can be observed at the FULL coalescence state only.

Figures 14.7a and 14.7b quantify the impact of bank conflicts in shared memory on power, performance, and energy consumption. They underline that the bank conflicts similarly impact on performance and energy on the two devices. In contrast the analysis underlines that up to 7 conflicts do not affect the max power on the GTX 660, while up to 7 conflicts strongly affect the max power on the TK1.

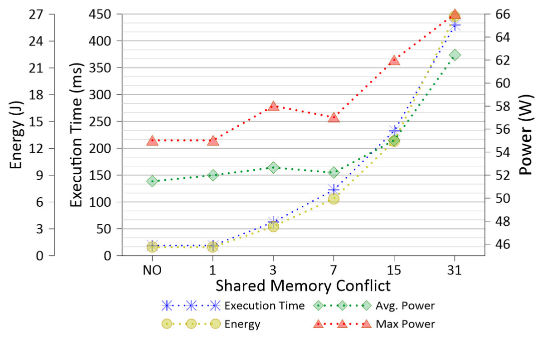


(a) *GTX 660*

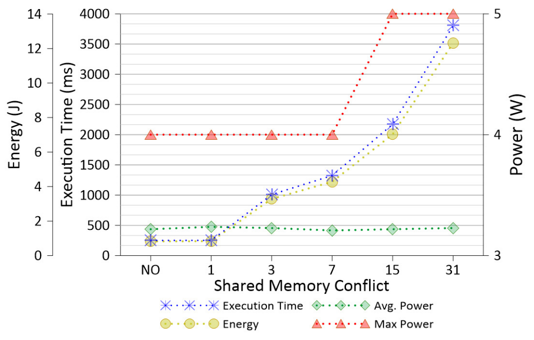


(b) *Jetson TK1*

FIG. 14.6: *DRAM coalescence*



(a) *GTX 660*



(b) *Jetson TK1*

FIG. 14.7: *Shared Memory Conflict*

Overall, the results obtained by running the proposed suite on a given GPU device allows understanding the specific impact of a tuning step on the design constraints (performance, power, and energy consumption). Improving the code performance that affect a specific functional component (e.g., coalescence of memory accesses) may violate a design constraint on a device, while it may not on a different device (see for instance the different effect on peak power on GTX660 and TK1 by increasing the memory coalescence). Combined with the standard profiler information, the proposed microbenchmark suite can efficiently guide developers in choosing among the possible optimizations during the whole iterative tuning flow.

14.4 Conclusions

This Section presented *MIPP*, a suite of microbenchmarks that aims at characterizing a GPU device in terms of performance, power, and energy consumption. *MIPP* aims at understanding how application bottlenecks involving selected functional components or underutilization of them can affect code performance, power consumption, and energy efficiency on a given device. The Section presented the results obtained by applying the microbenchmark suite to characterize two different GPU devices, i.e., an NVIDIA Kepler GTX660 and a low power embedded system NVIDIA Jetson TK1. The results showed how the same code optimizations have a different impact on the design constraints on the two GPU architectures.

15

Power-aware Performance Tuning of GPU Applications Through Microbenchmarking

Tuning GPU applications is a very challenging task as any source-code optimization can sensibly impact performance, power, and energy consumption of the GPU device. Such an impact also depends on the GPU on which the application is run. This Section presents a suite of microbenchmarks that provides the actual characteristics of specific GPU device components (e.g., arithmetic instruction units, memories, etc.) in terms of throughput, power, and energy consumption. It shows how the suite can be combined to standard profiler information to efficiently drive the application tuning by considering the three design constraints (power, performance, energy consumption) and the characteristics of the target GPU device.

15.1 Introduction

Graphic Processing Units (GPUs) have become increasingly used as general-purpose accelerators thanks to their computational power and programmability. Besides providing high performance, they also achieve excellent energy efficiency [196]. This makes them well suited to a variety of architectures, ranging from supercomputers to low-power and mobile devices [8].

On the other hand, the large number of operating hardware resources (e.g., cores and register files) employed in GPUs to support the massive parallelism can lead to a significant power consumption. The elevated levels of power consumption have a sensible impact on such many-core device reliability, ageing, performance scaling and deployment into a wide range of application domains. Different techniques have been proposed to manage the high levels of power dissipation and to continue scaling performance and energy. They include approaches based on dynamic voltage/frequency scaling (DVFS) [147], CPU-GPU work division [173], architecture-level/runtime adaptations [274], dynamic resource allocation [127], and application-specific (i.e., programming-level) optimizations [288]. Particularly in this last category, it has been observed that source-code-level transformations and application specific optimizations can significantly affect the GPU resource utilization, performance, and energy efficiency [247].

In this context, even though profiling tools (e.g., *CUDA nvprof*) exist to help programmers in the application analysis and optimization targeting performance,

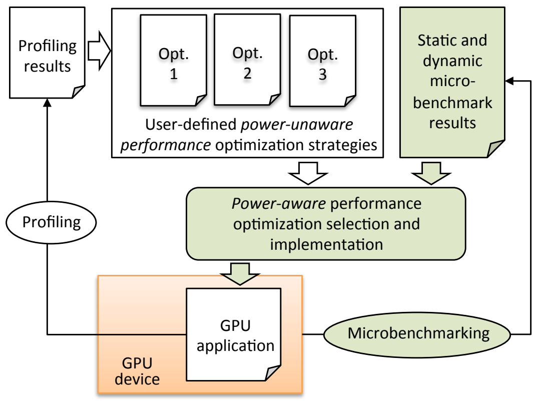

Profiling results
Opt. 1
Opt. 2
Opt. 3
User-defined *power-unaware* *performance* optimization strategies
Static and dynamic micro-benchmark results
Profiling
Power-aware performance optimization selection and implementation
GPU application
GPU device
Microbenchmarking

FIG. 15.1: *Overview of the proposed approach.*

they do not provide a complete view of the GPU features (especially on power consumption and energy efficiency) neither they provide a correlation among these design constraints. What is missing is a solution to measure, on a given GPU architecture, the potential effects of code optimizations on every design constraint before implementing them.

To overcome this limitation, this Section presents a suite of microbenchmarks, which aims at *characterizing* a GPU device in terms of performance, power, and energy consumption. The microbenchmark suite has been designed to be compiled and run on any CUDA GPU device, with the aim of quantitatively characterizing, *statically* and *dynamically*, all the functional components of the device. The functional components include arithmetic instruction units, memories (shared, cache, DRAM, constant), scheduling and synchronization units.

Fig. 15.1 shows how the proposed microbenchmarking can be combined with the standard profiling for a power-aware performance tuning of GPU applications. Given a GPU application, the standard profiling information allows defining a set of potential optimizations targeting performance. The microbenchmarks are run once for all in the target GPU device. By considering the functional components involved by an optimization strategy, the microbenchmark results on such components allow classifying the potential and the useless optimization strategies for the target design constraint before implementing them. The model allows the flow to be iterated for incremental tuning of the application.

The suite has been applied to characterize two different GPU devices (i.e., NVIDIA Kepler GTX660 and Maxwell GXT980), which are representative of the respective architectures, and to efficiently guide the tuning of two representative and widely used parallel applications.

The work is organized as follows. Section 15.2 presents the suite and how it is used to characterize a given device. Section 15.3 reports the experimental results, while Section 15.4 draws the conclusions.

Componet	Benchmark	Instructions	Thoman et al. [264]	Lemeire et al. [158]
ALU	**32-bit Integer Simple**	`add, sub`	✗	✓(ND)
ALU	**32-bit Integer Complex**	`mul`	✗	✓(ND)
ALU	**32-bit Integer Bit operations**	`clz, mbs, brev, bfi, bfe`	✗	✗
ALU	**32-bit Integer Shift**	`shl, shr`	✗	✗
ALU	**32-bit Integer Pop. count**	`popc`	✗	✗
ALU	**32-bit Integer Remainder**	`rem`	✗	✗
ALU	**64-bit Integer Simple**	`add, sub`	✗	✗
FPU	**32-bit FP Simple**	`add, sub, mul`	✓(74.0%)	✓(ND)
FPU	**32-bit FP Complex**	`div, div.ftz, div.approx, div.approx.ftz`	✓(85.8%)	✓(ND)
SFU	**32-bit FP Transcendental op.**	`sin, cos, exp, rsqrt, rcp, log`	✓(45.4%)	✓(ND)
DFU	**64-bit FP Simple**	`add, sub`	✓(74.1%)	✓(ND)
DRAM	**DRAM**	`load, store`	✗	✓(ND)
L2	**L2**	`load, store`	✗	✓(ND)
L1/Shared mem.	**Shared memory**	`load, store`	✗	✓(ND)
Constant mem.	**Constant memory**	`load, store`	✗	✓(ND)

Table 15.1: Microbenchmarks for static characteristics

15.2 The Microbenchmark Suite

A microbenchmark is a GPU kernel that exercises a specific functional component of the device and whose instructions can be evaluated at a clock-cycle accuracy. A microbenchmark main procedure consists of a long sequence of one or more selected instructions (e.g., arithmetic instructions, memory accesses) that executes without any interference deriving from other instructions. Each microbenchmark selectively stresses a functional component without or minimally affecting the others to provide reliable and accurate feedback. To do that, we implemented the microbenchmarks by combining common CUDA C/C++ language with inline intermediate assembly to avoid compiler side-effects.

The PTX language has been exploited to force a specific operation on a data type, to avoid compiler optimizations (which cannot be avoided by simply setting compiler flags like `-O0` in both C/C++ and PTX compilation), to prevent caching/local-storage mechanisms, and to restrict the memory access space.

Tables 15.1 and 15.2 summarize the GPU components and the corresponding low-level instructions statically and dynamically exercised by the proposed suite. The tables compare the completeness and the accuracy of the suite with the best and more complete suites at the state of the art (i.e., [158, 264]). The accuracy is essential for a correct characterization of the timing features of a GPU component, and even more to understand how a generic application can affect power and energy consumption of such a component. The accuracy is measured as the number of useful ("pure") instructions for a given component microbenchmarking over the total number of the microbenchmark instructions. Each microbenchmark of the proposed suite reaches an accuracy value equal to 99.99%. Such an accuracy is not reached by the counterparts (as reported in brackets). We derived the accuracy of the suite proposed by [158] experimentally (see Section 15.3), since the suite is not released with the source code.

Componet	Benchmark	Thoman et al. [264]	Lemeire et al. [158]
ALU	Loop unrolling	✗	✗
ALU	ILP	✗	✗
DRAM	Coalescence	✗	✗
DRAM	Access size	✗	✗
Shared memory	Bank conflicts	✗	✗
Streaming Multiprocessor	Device occupancy (SM)	✗	✗
SM scheduler	Device synchronization	✗	✗
SM scheduler	Thread divergence	✗	✗

Table 15.2: Microb. for dynamic characteristics

15.2.1 GPU static characteristics

The suite allows analysing the peak characteristics of the arithmetic and memory components of the GPU by applying extensive workloads on them. The *arithmetic* microbenchmarks target the complete set of arithmetic instructions natively supported by the GPU, by distinguishing between integer and floating-point over 32 and 64-bit word sizes. The *memory* microbenchmarks give information on the throughput (bandwidth) of DRAM, L1/shared, constant, and L2 cache memories. The *DRAM microbenchmark* executes several accesses at different memory locations with a stride of 128 bytes between grid threads to avoid L1 cache interferences. The *L2 microbenchmark* repeats a compile-time sequence of store instructions on the same memory address. We used cache modifiers [205] to avoid L1 cache hits in the store operations. *Shared* and *constant memory microbenchmarks* consist of a sequence of store/load instructions.

```
__device__ clock_t devClocks[RESIDENT_WARPS];
__device__ int devTMP;
__device__ volatile int devMemory[SIZE];
template<int TH_GROUP_SIZE>
__global__ Coalescence()
int thread_group_id = GLOBAL_THREAD_ID / TH_GROUP_SIZE;
int L1_bank_offset = thread_group_id · CACHE_LINE_SIZE;
volatile int* pointer = devMemory + L1_bank_offset + (GLOBAL_THREAD_ID%TH_GROUP_SIZE);
int R1 = threadIdx.x;                                          // assign dynamic value
clock_t start_tm = clock64();
InstrSeq<N>(pointer, R1);                                      // call the function N times
clock_t end_tm = clock64();
if (LANE_ID ==0) then devClocks[WARP_ID] = end_tm - start_tm;
if (THREAD_ID == 1024) then devTMP = R1;                       // never executed

template<int N>                                                // template metaprogramming
__device__ __forceinline__ InstrSeq(volatile int* pointer, int& R1)
const int STRIDE = RESIDENT_WARPS · CACHE_LINE_SIZE;
#pragma unroll                                                 // loop unrolling
for (int i = 0; i < 4096; i++) do
    asm volatile("ld.volatile.s32 %0, [%1]" : "=r"(R1) :
                  "l"(pointer + i * STRIDE) : "memory");
end
InstrSeq<N-1>(pointer, R1);                                    // recursive call
```

FIG. 15.2: *Example of the microbenchmark code to measure the impact of global memory coalescence.*

15.2.2 GPU dynamic characteristics

The suite includes dynamic microbenchmarks, which analyse the dynamic characteristics of the device by exercising the functional components with different intensity.

The memory microbenchmarks analyse how the memory access pattern of threads affects the memory throughput. This includes memory coalescence, memory access size, and bank conflicts involved by the implemented access pattern. As an example, the microbenchmark in Fig. 15.2 measures the impact of global memory *coalescence* on the memory throughput. To do that, the microbenchmark implements different patterns of memory accesses, where each pattern guarantees a different coalescence degree. Considering a base address (`devMemory`), each thread calculates the own *L1 bank offset* through the global identifier (`GLOBAL_THREAD_ID`), the size of the L1 cache bank (`CACHE_LINE_SIZE`), and through the `TH_GROUP_SIZE` variable (lines 1, 2 in the upperside of Fig. 15.2). This allows forcing different thread accesses to be grouped into the same L1 cache banks and, as a consequence, to group such thread accesses into coalesced global memory transactions. Then, each thread calculates the final pointer through base address, L1 bank offset, and thread offset in the bank. Fig. 15.3 shows an example, which underlines how grouping threads into coalesced transactions is parametrized through the `TH_GROUP_SIZE` variable. The microbenchmark dynamically sets such a variable to control the coalescence degree.

The code implements *volatile* quantifiers (`devMemory` definition and line 3 in the upper side of Fig. 15.2). This allows avoiding *local-storage* optimizations by the compiler, which may change the coalescence degree forced by the proposed strategy. The code adopts *recursive* and *template-based* meta-programming to generate an arbitrarily long sequence of arithmetic instructions ($N \times 4,096$ `store` instructions in the example). This allows improving the accuracy of the functional characteristics measurement. In the bottom side of Fig. 15.2, the `STRIDE` variable represents the minimum value of memory address offset that allows preventing false positive L1 cache hits. Such an offset guarantees that any thread cannot calculate the same pointer in two different loop iterations. Both the `STRIDE` and the global memory offsets (`i·STRIDE`) are computed at compile time to guarantee that the measured memory throughput is not distorted by such value computation.

The *arithmetic* microbenchmarks analyse the dynamic characteristics of the GPU computational units over two optimization aspects: unrolling and instruction-level parallelism (ILP). The *unrolling* microbenchmark iteratively executes a chunck of code in a loop, where the loop iterations/loop unrollings are set dynamically and increasingly from the minimum to the maximum. The microbenchmark returns the effect of eliminating conditional statements in terms of performance and power. The *ILP* microbenchmark executes a sequence of unrelated instructions, where the sequence length is set dynamically and incrementally.

The suite includes the *shared memory* microbenchmark, which generates a different amount of bank conflicts, from zero to the maximum value. The *access size* microbenchmark copies one large array into another multiple times by varying, at each iteration, the size of the data block.

The *thread scheduling and synchronization* microbenchmarks aim at studying the dynamic behavior of the device by varying the streaming multiprocessor oc-

	Integer 32-bit						**Integer 64-bit**	**FP 32-bit**		**FP 64-bit**
	Simple	Complex	Pop. Count	Shift	Bit OP.	Rem	Simple	Simple	Special	Simple
Execution Time (ms)	8.6	31.5	29.6	15.5	15.5	965.2	18.8	8.6	32.5	223.9
Spec Throughput OPs per Cycle per SM	128	n.a.	32	64	64	n.a.	n.a.	128	32	4
Real Throughput OPs per Cycle per SM	116.3 (66.3†)	31.8	32.0	63.4	63.5	1.0	51.6	116.3 (101.3*) (104.4†)	29.8 (33.9*)	4.0 (7.6*) (err†)
Avg. Power (W)	75.2	88.2	69.4	70.8	79.1	100.2	80.9	72.2	84.4	76.8
Max Power (W)	86	93	72	77	85	114	88	86	99	88
Energy (J)	0.7	2.8	2.1	1.1	1.2	96.7	1.5	0.6	2.74	17.20
Energy efficiency MIPS per Watt	26,564	6,190	8,370	15,672	13,995	178	11,269	28,120	6,262	999
nano Joule per instruction	0.04	0.16	0.12	0.06	0.07	5.63	0.09	0.04	0.16	1.0

Table 15.3: *GTX980 - Characterization of arithmetic instructions (static characteristics).* **results of Thoman et al. [264],* †*results of Lemeire et al. [158]*

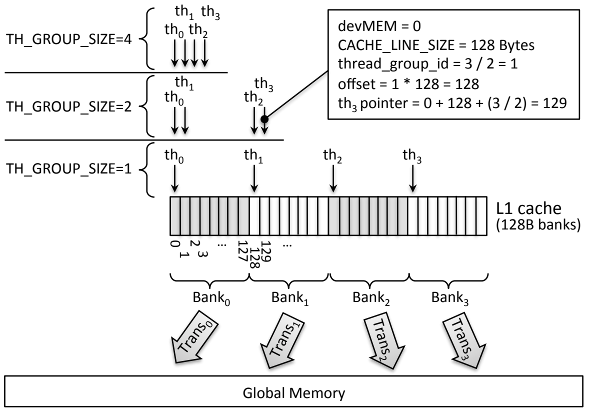


FIG. 15.3: *Controlled memory coalescence*

cupancy and the degree of thread divergence. They also aim at characterizing the device by considering the synchronization overhead caused by thread block barriers over the whole kernel. The *device occupancy (SM)* microbenchmark evaluates the contribution of different number of active SMs on the computation, while the *device synchronization* one analyses the impact of synchronization barriers in the code.

The arithmetic processing benchmarks allowed us to measure the impact of the `div` precision in GPU applications running on the GTX 660 and GTX 980 devices, respectively (see Tables 15.4 and 15.5). In both cases, the division precision can be set at compile time, (`prec-div=false` enables fast approximation mode breaking IEEE754 Standard, `ftz=true` flushes de-normal values to zero). The tables show that in both devices, the execution time, the peak throughput, and the energy are proportional to the selected precision. The table underlines that the throughput and the energy consumption of the division operation with the lowest precision is comparable with FP 32-bit special instructions.

	IEEE754		Fast Approximation	
	Div	Div FTZ	Div	Div FTZ
Execution Time (ms)	223.7	222.4	53.0	36.6
Real Throughput (OPs×Cycle)×SM	5.3	5.3	24.1	34.3
Avg. power (W)	68.5	68.3	60.7	56.4
Max power (W)	73.0	72.0	64.0	59.0
Energy (J)	15.3	15.2	3.2	2.1
Energy efficiency (MIPS×Watt)×SM	70.1	70.7	334.2	519.5
nano Joule per instruction	14.3	14.1	3.0	1.9

Table 15.4: *660 GTX - Impact of* `div` *precision.*

	IEEE754		Fast Approximation	
	Div	Div FTZ	Div	Div FTZ
Execution Time (ms)	276.0	274.7	59.3	32.4
Real Throughput (OPs×Cycle)×SM	4.9	4.9	16.8	29.7
Avg. power (W)	100.8	89.5	81.8	81.3
Max power (W)	128.0	117.0	86.0	86.0
Energy (J)	1.74	1.54	0.30	0.17
Energy efficiency (MIPS×Watt)×SM	617	698	3,539	6,512
nano Joule per instruction	1.62	1.43	0.28	0.15

Table 15.5: *GTX 980 - Impact of* `div` *precision.*

15.3 Experimental Results

15.3.1 GPU Device Characterization

We run the microbenchmarks on an NVIDIA GeForce GTX660 and on a GTX980, which are representative of the Kepler and Maxwell architectures, respectively.

The devices have been evaluated with CUDA Toolkit 7.5, AMD Phenom II X6 1055T (3GHz) host processor, and Ubuntu 14.04 operating system. The proposed suite is independent from the specific CUDA-enabled GPU device and from the adopted CUDA Toolkit version.

Performance information has been collected through the CUDA runtime API to measure the execution time and through the `clock64()` device instruction for throughput values to ensure clock-cycle accuracy of time measurements.

Power and energy consumption information has been collected through the *Powermon2* power monitoring device [35]. The analysis has been performed with the default GPU frequency setting and by disabling any PCI/GPU adaptive frequency or thermal throttling mechanisms (i.e., *GPUBoost*).

Table 15.3 reports the results obtained by running the static microbenchmarks on the GTX980 device. For the sake of space and without loss of generality, we do not report the *static* characteristics of the GTX660 device, since they are not necessary for understanding the focus of the Section. We only report, for the GTX660, the most useful *dynamic* benchmarking results.

The static microbenchmarks are organized over columns and consist of 10^9 instructions per SM. For each microbenchmark, the table reports the execution time, the theoretical peak throughput of the corresponding functional unit provided in the device specifications [213] (*Spec. throughput*) and that measured through the proposed microbenchmark (*Real throughput*). The static microbenchmarks measure the maximum arithmetic instruction throughput of simple integer operations (`add`, `sub` , etc.), complex integer operations (`mul`, `mad`, etc.), integer population count, shift, remainder, bitwise (bit insert, bit reverse, etc.), simple single precision floating-point operations (`add`, `mul`, etc.), complex single precision floating-point operations (transcendental functions such as `sin`, `rcp`, etc.) and double precision floating-point operations. The device specifications do not include the theoretical peak throughput of the integer 64-bit and integer 32-bit remain operation (`rem`) and integer 32-bit complex operation since such operations have not an embedded hardware implementation. They are performed through a combination of different hardware units.

The results of the static microbenchmarking allow understanding the microbenchmark accuracy by comparing the measured throughput with the throughput reported in the device specification.

They also underline the accuracy difference between each microbenchmark of the proposed suite and the corresponding microbenchmark, when provided, of the best suites at the state of the art (i.e., [158, 264]). It is worth noting that the accuracy of the state of the art microbenchmarks for simple instructions is very low. This is due to the compiler activity on the source code (unavoidable even disabling any optimization flag), which inserts ”spurious” instructions in the executable code. Such optimizations lead the throughput measured on the FP instructions to be even higher than the real throughput

Table 15.3 also reports information about power and energy consumption, which is not reported in the device specifications. The energy efficiency (or performance per watt) [99] is defined as the number of operations/instructions computed per second per Watt. We refer to million instructions per second (MIPS) for arithmetic benchmarks and million of memory transactions for memory benchmarks. Finally, the table shows the power consumption (nJ) per single instruction/memory transaction.

Table 15.6 reports the results of the static microbenchmarks on the GPU memories. They allow understanding how the throughput differs among memories. As an example, an application running on the GTX980 accesses the shared memory 11 times faster than the DRAM (the corresponding microbenchmarking on the GTX660 reports that the same application running on the GTX660 accesses the

	Dram	L2	Shared	Constant
Exec Time (ms)	9,536.2	2,522.5	847.6	226.4
Real Throughput **(Trans×Cycle)×SM**	0.09 ($0.08^{\dagger}$)	0.33 ($0.14^{\dagger}$)	1.02 ($0.85^{\dagger}$)	4.06 ($3.08^{\dagger}$)
Avg. power (W)	113.3	105.8	94.1	76.0
Max power (W)	117	112	105	87
Energy (J)	67.54	16.7	5.0	1.1
Energy efficiency **(10^6Transactions/Watt)**	15.8	64.4	215.3	998.0
nano Joule per mem. transaction	62.9	15.5	4.6	1.0

TABLE 15.6: *Characteristics of mem. accesses on the GTX980.* †*results of Lemeire et al. [158]*

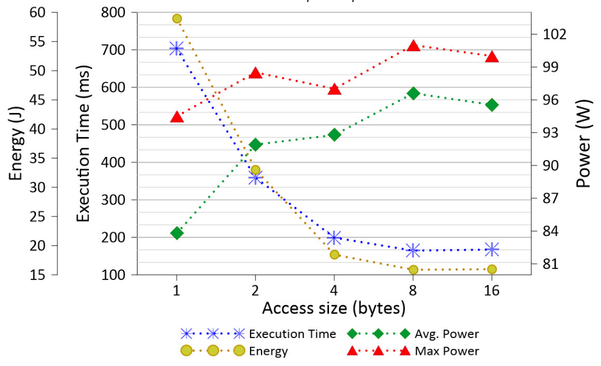


FIG. 15.4: *GTX980 DRAM access size*

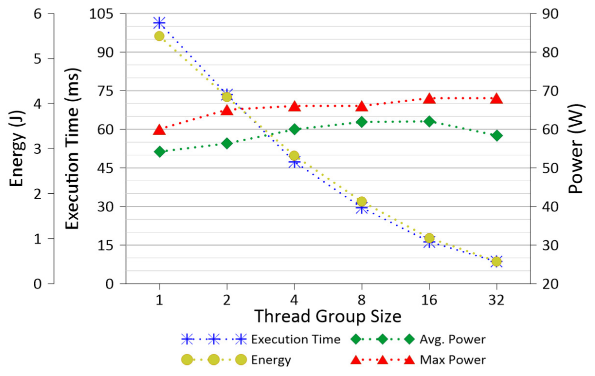
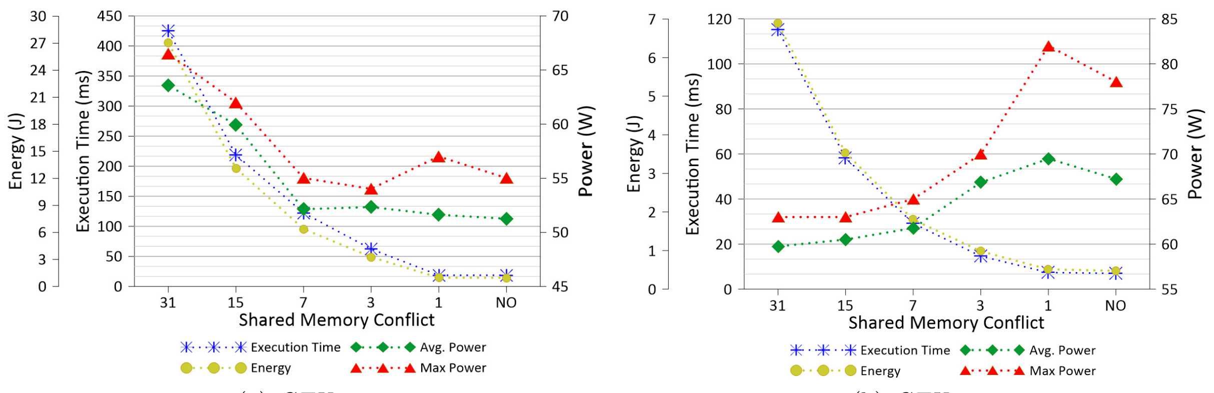


FIG. 15.5: *GTX980 thread divergence*

shared memory 5 times faster than the DRAM). The DRAM accesses strongly affect the average and peak power. The constant memory, which presents the best energy efficiency, is 3.3 times more energy efficient than the L2 cache in the GTX980 (1.5 times in the GTX660). The microbenchmarks of the state of the art suites (i.e., [158, 264]) do not allow measuring power and energy consumption since they are too fast also for the best sampling frequency available in literature provided by the *Powermon2* device.



(a) *GTX660* (b) *GTX980*

FIG. 15.6: *Shared memory bank conflicts*

Figures 15.4-15.6 report some of the dynamic microbenchmarking results (the most relevant for the case study presented in Section 15.3). Fig. 15.4 reports the impact of the thread access size in DRAM, starting from 1-byte to 16-byte blocks per thread. Increasing the access size sensibly improves both performance and energy consumption at the cost of slightly more average and peak power in the GTX980. The results are proportionately similar in the GTX660.

Fig. 15.5 reports the analysis of thread divergence. Performance and energy consumption linearly improve by moving from the maximum divergence (1-sized thread groups in the left) to the minimum divergence (32-sized thread groups in the right), at the cost of a slight increase of peak power.

Figures 15.6a and 15.6b quantify the effect of bank conflicts in shared memory for both the GTX660 and GTX980. They underline that the bank conflicts similarly impact on performance and energy on the two devices, while they affect average and peak power in the opposite way. In the GTX660, average and max power sensibly decrease (up to 20%) by decreasing the bank conflicts from the maximum (31) to 7. After that (from 7 to 0) there is no meaningful effect on them. In the GTX980, there are no meaningful variations on the power by decreasing the bank conflicts from the maximum to 7, while further reducing the conflicts (from 7 to 0) involves the most sensible power increase. This is due to the advanced instruction scheduler of the Maxwell architecture that, differently from that in the GTX660, keeps up the throughput of few or no bank conflicts.

The *memory coalescence* microbenchmark analyses the impact of the coalescence in DRAM memory accesses on the device performance, power, and energy consumption. The results, which are not reported for the sake of space, quantitatively show that performance, power, and energy are linearly proportional to the coalescence degree.

We used the proposed microbenchmarks combined with the standard profiler information to drive the tuning of two widespread parallel applications, *vector reduction* and *matrix transpose*. Each application tuning has been performed for both the GTX660 and GTX980 devices.

				GTX660				GTX980			
Opt. branch	Ver.	Profiler metrics and features	Related microbenchmark	Exec. time (ms)	Avg. power (W)	Peak power (W)	Energy (J)	Exec. time (ms)	Avg. power (W)	Peak power (W)	Energy (J)
-	orig.	inst_per_warp= 347 integer_rem	-	260.8	83.2	101	21.7	108.7	165.3	169	17.9
1	v1.0	inst_per_warp= 130	Divergence Arith. throughput	184.7	73.7	77	13.6	52.6	167.5	135	6.7
2	v2.0	gmem_throughput=75GB/s (GTX660) gmem_throughput=145GB/s (GTX980)	Access size	37.4	77.7	82	2.9	17.2	132.2	151	2.1
	v2.1	gmem_throughput=111GB/s (GTX660) gmem_throughput=165GB/s (GTX980)	Access size	23.2	77.1	85	1.7	15.2	133.5	153	1.6

Table 15.7: *Vector reduction characteristics on GTX660 and GTX980 devices.*

15.3.2 Vector Reduction

Vector reduction is one of the most common and important application cores in parallel computing. It consists of performing a binary and associative operation over all elements of a data vector to obtain a single final value.

We started from the implementation presented in [69], which applies, as associative operator, the addition to a vector of integers. The goal was to generate two distinct variants of the original code, the first (branch 1) targeting a lower peak power, the second (branch 2) targeting performance speedup.

For the first branch, the best code optimization we identified to lower the peak power without losing performance was reimplementing the following code pattern to control the thread execution paths:

```
if (threadIdx.x % (2 * stride) == 0)
    Mem[threadIdx.x]+=Mem[threadIdx.x+stride];
```

The idea was to replace the `rem` PTX operations used in the original code with `add` and `mul` PTX operations (see Table 15.3 for the peak power comparison among such arithmetic instructions). On the other hand, the identified modification potentially reduces the *thread divergence*, since it forces only the neighbouring threads to compute the reduction body (second line of the code above). However, by analysing the microbenchmark results on the thread divergence (Fig. 15.5 for the GTX980, and similar for the GTX660), we expected no meaningful increase of peak power as a *side-effect* of any thread divergence reduction.

According to the results of such an analysis, we expected slightly higher performance and much lower peak power in this code version w.r.t. the original code in both the devices. We also expected a stronger peak power reduction and a lower speedup in the GTX660 while a weaker power reduction and a higher speedup in the GTX980, as then confirmed by the results (-24% peak power and 1.4x speedup in GTX660, -20% peak power and 2.1x speedup in GTX980). Table 15.7 summarizes the obtained results and reports, for each code version, the used profiler metrics, the most relevant features that characterize the code, the analysed microbenchmarks, the execution time, the average and peak power, and the energy consumption.

Opt. branch	ID	Profiler metrics and features	Related microbenchmark	GTX660 Exec. time (ms)	GTX660 Avg. power (W)	GTX660 Peak power (W)	GTX660 Energy (J)	GTX980 Exec. time (ms)	GTX980 Avg. power (W)	GTX980 Peak power (W)	GTX980 Energy (J)
-	orig.	`global_mem_eff`= 0.125 `ipc`= 0.3 (GTX660) `ipc`= 0.4 (GTX980)	-	52.2	75.0	79	3.9	55.8	86.5	92	4.8
1	v1.0	`global_mem_eff`= 0.5 `ipc`= 0.6 (GTX660) `ipc`= 0.6 (GTX980)	DRAM Coalescence	9.3	69.3	87	0.6	26.8	94.2	103	2.5
2	v2.0	`shared_mem_eff`= 6%	Bank conflicts	10.4	69.5	81	0.7	16.3	90.8	97	1.4
	v2.1	`shared_mem_eff`= 100%	Bank conflicts	6.3	61.4	74	0.4	10.4	90.2	105	0.9

TABLE 15.8: *Matrix transpose characteristics on the GTX660 and GTX980 devices.*

In the other optimization branch (v2.x) we identified a different memory access strategy [170], which allows applying a variety of memory access sizes in the application. We thus analysed the microbenchmark results on the memory access sizes (Fig. 15.4) and observed that increasing the access size can lead to a sensible performance improvement with no meaningful side-effects on peak power. We applied the technique proposed in [213] to cast the inputs to a data type of larger size, thus forcing vectorized memory accesses. The technique improved both the performance and the peak power w.r.t. the original code and, as expected, the increasing of memory access size led to a further speedup with no significant power increase in both the devices.

15.3.3 Matrix Transpose

We analyzed the matrix transpose implementation provided in [227] and, by considering the profiling information, we identified two optimization branches targeting memory coalescence for global memory accesses and bank conflict reduction for shared memory accesses, respectively. Coalescence and memory access pattern are two of the most important factors to be considered in the tuning phase of any GPU application, especially if the application, like the matrix transpose, is memory-bound.

In particular, the memory coalescence optimization allows sensibly improving the performance speedup, while the bank conflict reductions allows tuning the tradeoff between performance and peak power (see Fig. 15.6).

By combining the profiling information of the original code, which underlined a low global memory accesses efficiency (`global_mem_eff`=0.125) and the dynamic characteristics of the memory coalescence provided by the corresponding microbenchmark, we expected, for the first branch, an increase of both performance speedup and peak power proportional to the memory coalescence improvement on both the devices. We thus optimized the memory coalescence by re-organizing the thread block configuration in v1.0. Table 15.8 shows the results, which confirm the expected positive speedup at the cost of a higher peak power (5.6x speedup and +10% peak power in GTX660, 2.1x speedup and +12% peak power in GTX980).

In the other branch (v2.x), we identified a different memory access pattern, which allows taking advantage of the shared memory to locally transpose a tile of the whole matrix and to optimize the memory load/store operations of the matrix elements. The implementation of such an optimization (v2.1) provided a further tuning opportunity, since the profiling of such a code version indicated bad access patterns in shared memory. This was underlined by a low value of `shared_mem_eff`[1], which involves a waste of the memory bandwidth. By analysing the results of the microbenchmarks on the bank conflicts (Figs. 15.6a and 15.6b), we expected, as a consequence of solving such a bottleneck, an improvement on both performance and peak power in the GTX660. We thus applied the *memory padding* technique [227] to reduce the bank conflicts, which led to 8.3x speedup and -5% peak power w.r.t the original version. As suggested by the microbenchmarks, we would not have applied such a time consuming optimization to reduce the peak power in the GTX980. The uselessness of such an optimization on the GTX980 has been then confirmed by the experimental results (+14% peak power w.r.t. the original version).

15.4 Conclusions

This Section presented a suite of microbenchmarks to statically and dynamically characterize GPU devices in terms of performance, power, and energy consumption. The Section showed how the microbenchmark results can been combined with the standard profiler information to efficiently tune any parallel application for a given GPU device and for a given design constraint (performance speedup or peak power reduction). Experimental results have been conducted on two widespread parallel applications for two representative GPU devices. They showed how the proposed microbenchmarking can improve the efficiency of the tuning task by identifying the potential from the useless optimization strategies before implementing them.

[1] A value of `shared_mem_eff` equal to 6% corresponds to 31 bank conflicts in the shared memory microbenchmark (see Figs. 15.6a,15.6b):

$$\frac{1\ access}{total\_accesses=32} \cdot \frac{smem\_bank\_size=8byte}{data\_size=4byte} = 6\%.$$

16

A Performance, Power, and Energy Efficiency Analysis of Load Balancing Techniques for GPUs

Load balancing is a key aspect to face when implementing any parallel application for Graphic Processing Units (GPUs). It is particularly crucial if one considers that it strongly impacts on performance, power and energy efficiency of the whole application. Many different partitioning techniques have been proposed in the past to deal with either very regular workloads (static techniques) or with irregular workloads (dynamic techniques). Nevertheless, it has been proven that no one of them provides a sound trade-off, from the performance point of view, when applied in both cases. More recently, a dynamic multi-phase approach has been proposed for workload partitioning and work item-to-thread allocation. Thanks to its very low complexity and several architecture-oriented optimizations, it can provide the best results in terms of performance with respect to the other approaches in the literature with both regular and irregular datasets. Besides the performance comparison, no analysis has been conducted to show the effect of all these techniques on power and energy consumption on both GPUs for desktop and GPUs for low-power embedded systems. This chaper shows and compares, in terms of performance, power, and energy efficiency, the experimental results obtained by applying all the different static, dynamic, and semi-dynamic techniques at the state of the art to different datasets and over different GPU technologies (i.e., NVIDIA Maxwell GTX 980 device, NVIDIA Jetson Kepler TK1 low-power embedded system).

16.1 Introduction

Graphic Processing Units (GPUs) have become increasingly used as general-purpose accelerators thanks to their computational power and programmability. Besides providing high peak performance, they also achieve excellent energy efficiency. This makes them well suited to a variety of architectures, ranging from supercomputers to low-power and mobile devices [196].

Nevertheless, the current GPU programming paradigm does not allow developers to automatically address issues like load balancing and GPU resource utilization. A meaningful example is the CUDA scheduler, which cannot handle the unbalanced workload efficiently. Particularly with problems that do not exhibit enough parallelism to fully utilize the GPU, employing the canonical GPU programming paradigm easily leads to underutilization of the computation power. These issues are essentially due to fundamental limitations on the current data parallel programming methods [67]. Indeed, the workload decomposition and allocation strategies are let to the application designer. How the application implements such a mapping can have a significant impact on the overall application performance. In addition, the load balancing strategy implemented in the GPU application strongly affects also the power consumption and energy efficiency, which are becoming fundamental design constraints in addition to performance [128].

Different techniques for GPU applications have been presented in literature to decompose and map the workload to threads [30, 59, 83, 113, 117, 125, 190]. All these techniques differ in the complexity of their implementation and from the overhead they introduce in the application execution to address the most irregular workloads. In particular, the simplest solutions [117,125] apply well to very regular workloads while they cause strong unbalancing and, as a consequence, lost of performance in case of irregular workloads. More complex solutions [30,59,83,113, 190] best apply to irregular problems through semi-dynamic or dynamic workload-to-thread mappings. Nevertheless, the overhead introduced for such a mapping often worsens the overall application performance when run on regular problems. More recently, a partitioning and mapping technique called *Multi-phase* [58] has been proposed to address the workload unbalancing problem in both regular and irregular problems. It implements a dynamic allocation of work-units to threads through an algorithm whose complexity is sensibly reduced with respect to the other dynamic approaches in the literature.

Although all these techniques have been compared in terms of performance over very different datasets [61], no analysis has been conducted to prove (i) whether the most efficient in terms of performance can also guarantee the best power and energy consumption (ii) the performance-power trade-off of such techniques when applied on low-power embedded GPUs.

This chaper presents an experimental analysis of all the most efficient load balancing techniques at the state of the art applied on different benchmarks and over different GPU architectures (i.e., NVIDIA Maxwell GTX 980 device, NVIDIA Jetson Kepler TK1 low-power embedded system) to understand when and how each technique best applies in terms of performance, power, and energy consumption.

The chaper is organized as follows.Section 16.1.1 summaries the Multi-Phase Search algorithm for GPUs. Section 16.2 presents the load balancing analysis, while Section16.3 is devoted to the conclusions.

16.1.1 The *Multi-phase* technique

Multi-phase aims at exploiting the balancing advantages of the two-phase algorithms while overcoming the limitations of the scattered memory accesses. It consists of a *hybrid partitioning phase* and an *iterative coalesced expansion.*

Hybrid partitioning

Differently from all the other dynamic techniques in literature, which strongly rely on the binary search, *Multi-phase* relies on a hybrid partitioning strategy by which each thread searches the own work-items. Such a hybrid strategy dynamically switches between an *optimized binary search* and an *interpolation search* depending on the benchmark characteristics.

Optimized binary search: In the standard implementation of the binary search, each thread finds the searched element, on a prefix-sum array of N elements, through one memory access in the best case or through $2\log N$ memory accesses in the worst case. Indeed, at each iteration, each thread performs two memory accesses, to check the lower bound (value at the left of the index) and the upper bound (value at the right of the index) to correctly update the index for the next iteration. Nevertheless, in the context of binary search on prefix-sum, since all threads must be synchronized by a barrier before moving to the next iteration, and since at least one thread executes all iterations involving $2\log N$ memory accesses, each binary search actually has a time complexity equal to $2\log N$ memory accesses. In *Multi-phase*, each thread checks, at each iteration, only the lower bound, thus involving only one memory access per iteration. On the other hand, this approach requires all threads to perform all iterations ($\log N$) indistinctly. Overall, such an optimization halves the binary search complexity to $\log N$ memory accesses.

Interpolation search: In case of uniformly distributed inputs (i.e., low standard deviation of work-item size) and a low average number of work-units, *Multi-phase* implements an *interpolation search* [219] in alternative to the optimized binary search. The interpolation search has a very low complexity ($O(\log\log N)$) at the cost of additional computation. The algorithm pseudocode is the following:

```
**Interpolation Search** $(Array, left, right, S)$
1:  **while** $S \geqslant Array[\text{left}]$ and $S \leqslant Array[\text{right}]$ **do**
2:      $K = \text{left} + (S - Array\,[\text{left}]) \cdot \frac{\text{right}-\text{left}}{Array[\text{right}]-Array[\text{left}]}$
3:      **if** $Array\,[K] < S$ **then**
4:          $left = K + 1$
5:      **else if** $Array\,[K] > S$ **then**
6:          $right = K - 1$
7:      **else**
8:          **return** $K$
9:      **end**
10: **end**
```

The idea is to use information about the underlying distribution of data to be searched in a human-like fashion when searching a word in a dictionary. Given a chunk of prefix-sum elements ($Array$) and the item to be searched (S), the procedure iteratively calculates the next search position K (row 2 of the algorithm) by mapping S in the distribution $Array[\text{left}], \text{Array}[\text{right}]$. The algorithm shows an average number of comparisons equal to $O(\log\log n)$ that increase to $O(N)$ in the worst case, differently to the binary search that shows complexity $O(\log N)$ in all cases.

The main drawback is the higher computational cost to calculate the next index of the search (row 2), which involves double precision floating-point operations (division, multiplication, and casting). Such operations present a very low arithmetic throughput in GPU devices compared with single precision operations. To limit such a cost, *Multi-phase* implements the computation by minimizing the expensive double precision operations and by replacing them with 64-bit integer operations when possible.

Multi-phase switches between interpolation and binary search depending on the benchmark characteristics. In particular, the interpolation search runs if both the following conditions hold:

$$Std\_Dev\_WI_{size} \leqslant Threshold_{SD}$$

and

$$Average\_WI_{size} \leqslant Threshold_{AVG}$$

where the standard deviation of the work-item size and the average work-item size of the benchmark are calculated runtime. The switching between the two search methods is parametrized through the two thresholds that have been heuristically set to $Threshold_{SD} = 5$ and $Threshold_{AVG} = 3$ for all the analyzed benchmarks.

Iterative Coalesced Expansion

In the expansion phase, all threads of each block load the corresponding chunks into the shared memory. Then, each thread performs a binary search (optimized as in the partitioning phase presented in Section 16.1.1) in such a local partition to get the assigned work-unit. Then, the expansion phase consists of three iterative sub-phases, by which the scattered accesses of threads to the global memory are reorganized into coalesced transactions. This is done in shared memory and by taking advantage of local registers:

1. *Writing on registers.* Instead of sequentially writing on the work-units in global memory, each thread sequentially writes a small amount of work-units in the local registers.
2. *Shared mem. flushing and data reorganization.* After a thread block synchronization, the local shared memory is flushed and the threads move and reorder the work-unit array from the registers to the shared memory.
3. *Coalesced memory accesses.* The whole warp of threads cooperate for a coalesced transaction of the reordered data into the global memory. This step does not require any synchronization since each warp executes independently on the own slot of shared memory.

Steps two and three iterate until all the work-units assigned to the threads are processed. Even though these steps involve some extra computation with respect to the direct writings, the achieved coalesced accesses in global memory significantly improve the overall performance.

Workload Source	Avg. work-item size	Std. Dev. work-item size	Max work-item size
great-britain_osm	2.1	0.5	8
web-Notredame	5.2	21.4	3,445
cit-Patents	4.8	7.5	770
circuit5M	10.7	1,356.6	1,290,501
as-Skitter	13.1	136.9	35,455

TABLE 16.1: Benchmark Characteristics

16.2 Load balancing analysis

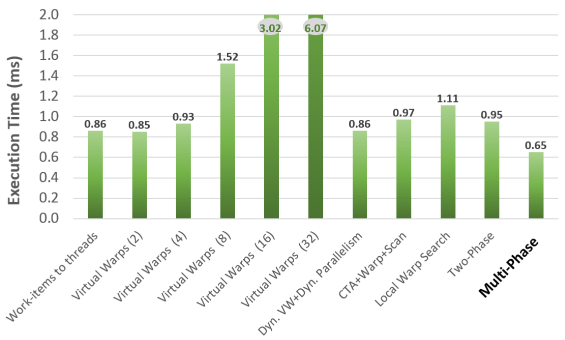
16.2.1 Characteristics of datasets, GPU devices, and equipment for performance, power, energy efficiency measurement

We tested the load balancing efficiency in terms of performance, power consumption and energy efficiency of all the more representative techniques in the literature (presented in Section 4.1) over a dataset of different benchmarks (see Table 16.1). The dataset, consists of five representative benchmarks selected from *The University of Florida Sparse Matrix Collection* [84], which consists of a huge set of data representation from different contexts (e.g., circuit simulation, molecular dynamic, road networks, linear programming, vibroacoustic, web-crawl). The five benchmarks have been selected among the collection to cover very different data characteristics in terms of maximum work-item size, average, and standard deviation from the item size. As summarized in the table, they span from very regular to strongly irregular workloads. The *great-britain_osm* benchmark represents a road network with very uniform distribution and low average. *web-NotreDame* is a web-crawl with a slightly higher average and middle-sized standard deviation. *Cit-patent* represents the U.S. patent dataset, which has moderate average and not-uniform distribution. *Circuit5M* represents a circuit simulation instance, while *as-skitter* is an autonomous system. The last two benchmarks are characterized both by highly not-uniform distribution and middle-sized average.

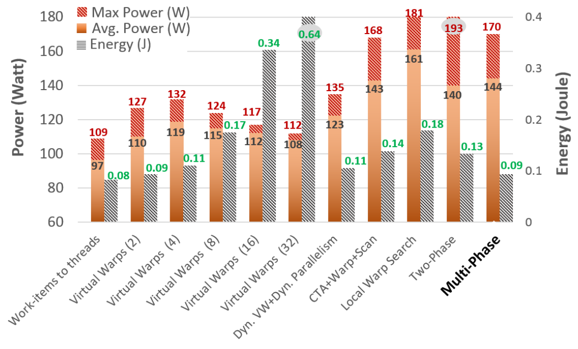
All the analyzed balancing techniques have been integrated in a reference application, in which the threads access and update, in parallel, each work-unit of the benchmark workload. We run the experiments on two different GPU devices. The first is an NVIDIA Maxwell GeForce GTX 980 with CUDA Toolkit 7.5, AMD Phenom II X6 1055T (3GHz) host processor, and Ubuntu 14.04 OS. The second is a Tegra K1 SoC (Kepler architecture) on an NVIDIA Jetson TK1 embedded system, with CUDA Toolkit 6.5, 4-Plus-1 NVIDIA-ARM host multi processor and Ubuntu 14.04 OS.

Performance information has been collected through the CUDA runtime API to measure the execution time and through the `clock64()` device instruction for throughput values to ensure clock-cycle accuracy of time measurements.

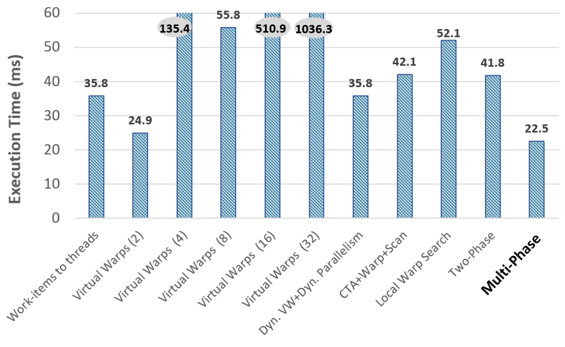
Power and energy consumption information have been collected through the *Powermon2* power monitoring device [35]. It allows measuring the voltage and the current values from different sources at the same time with a frequency of 3,072 Hz multiplexed across a subset of the 8 channels. We used a *interposer/riser card* to isolate the GPU pci-express power connector from the motherboard, while we directly connected the *Powermon2* device to the ATX power connectors. For each power supply source, we measured the instantaneous current and voltage to compute the power values. The analysis has been performed at a constant 21.0°C temperature to avoid temperature-related current leakage variations. The analysis has been performed with the default GPU frequency setting and by disabling any PCI/GPU adaptive frequency or thermal throttling mechanisms (i.e., *GPU-Boost*). The measurement protocol consists of executing each run several times to validate the corresponding results. The procedures ensure the measuring of the first voltage/current sample at the same instant the GPU kernel starts. We forced five seconds delay across different runs to avoid RLC effects and the corresponding interference with the subsequent experiment.



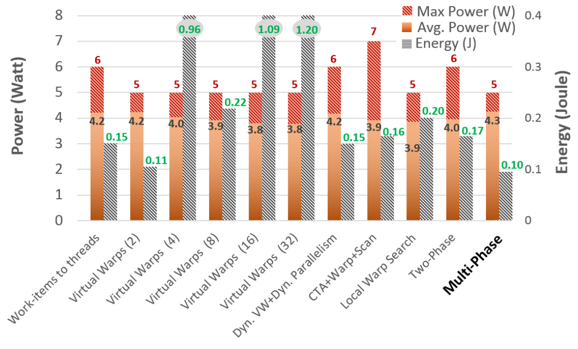
(a) *GTX980 - Execution Time*



(b) *GTX980 - Power and energy consumption*

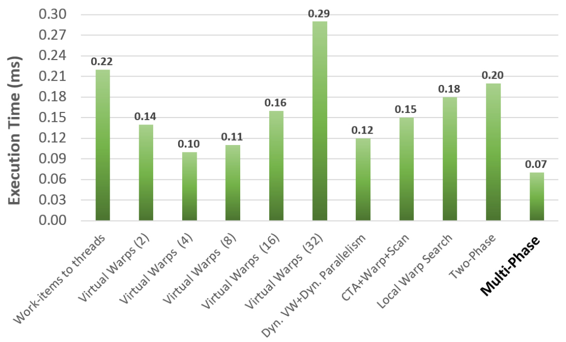


(c) *Jetson TK1 - Execution Time*

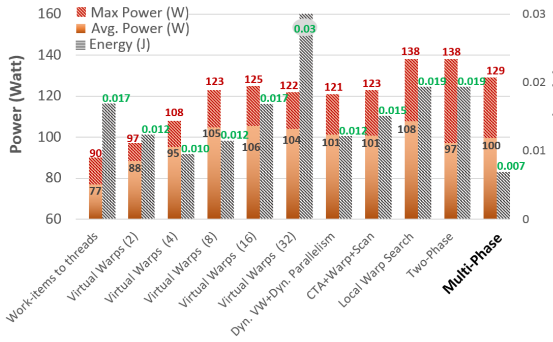


(d) *Jetson TK1 - Power and energy consumption*

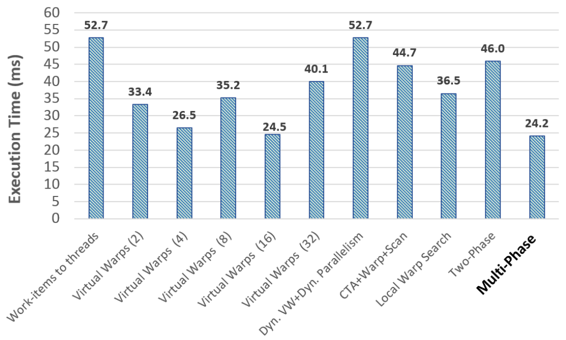
FIG. 16.1: *Comparison of execution time, power and energy consumption on great-britain_osm.*



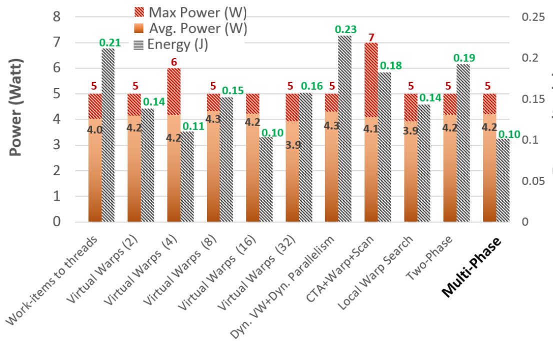
(a) *GTX980 - Execution Time*



(b) *GTX980 - Power and energy consumption*



(c) *Jetson TK1 - Execution Time*



(d) *Jetson TK1 - Power and energy consumption*

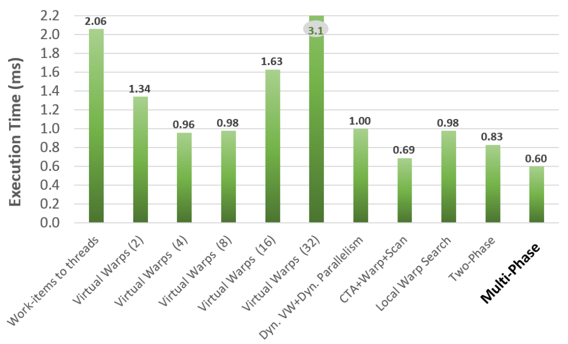
FIG. 16.2: *Comparison of execution time, power and energy consumption on web-Notredame.*

16.2.2 Performance, power, energy efficiency analysis

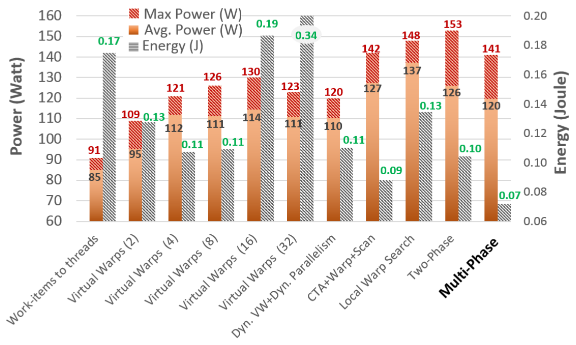
Figures 16.1, 16.2, 16.3, 16.4, and 16.5 report the obtained experimental results in terms of execution time, peak power, and energy consumption. In particular, the reported values are the best results of each technique we obtained by tuning the kernel configuration in terms of number of threads per block. For the two GPU devices used in this analysis, the best results have been reached with 128-256 threads per block for all the techniques, which provide the maximum occupancy of the device and the lowest synchronization overhead.

Work-item to threads [117] and *Virtual warps* [125] represent the static techniques (*Virtual warp* has been evaluated with different warp sizes). *Dyn.VW+Dyn.Parallelism* [59] and *CTA+Warp+Scan* [190] represent the semi-dynamic techniques, while *Local Warp Search* [83], *Two-Phase*, and *Multi-Phase* represent the dynamic ones. For the *Two-Phase* algorithm, we used the well-know *ModernGPU* library [30], which is based on the GPU algorithm proposed by Green [113].

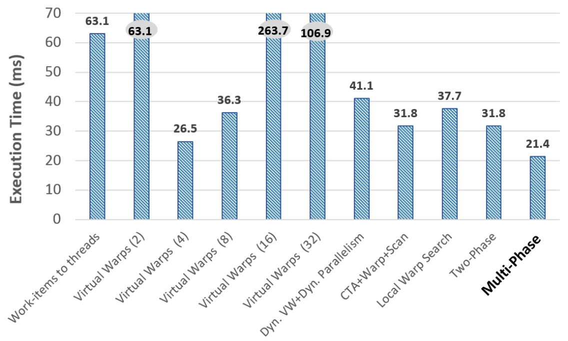
In the first benchmark, *great-britain_osm*, as expected, *Work-items to threads* and *Virtual Warps*(2) are the approaches, among those in the literature, with the



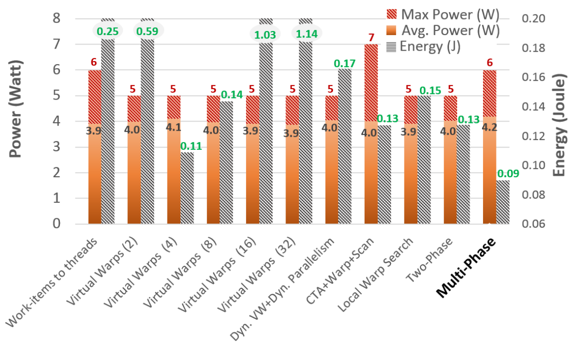
(a) *GTX980 - Execution Time*



(b) *GTX980 - Power and energy consumption*



(c) *Jetson TK1 - Execution Time*

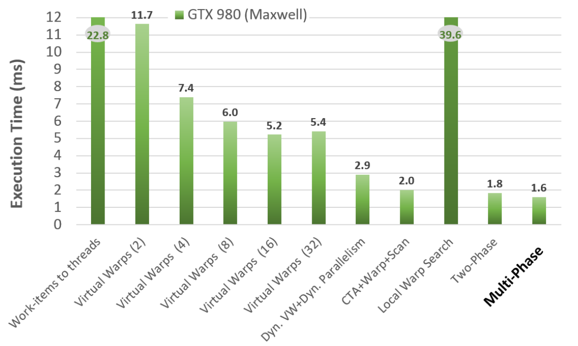


(d) *Jetson TK1 - Power and energy consumption*

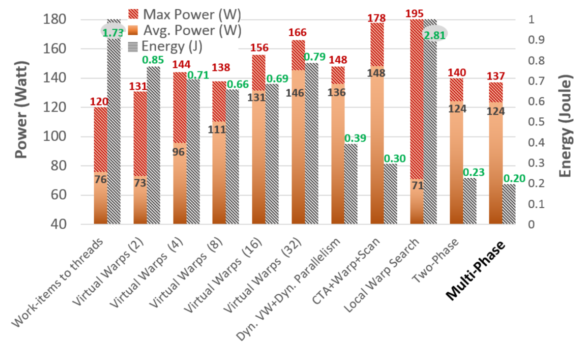
FIG. 16.3: *Comparison of execution time, power and energy consumption on Cit-Patents.*

best performance in both the GPU architectures (see Figs. 16.1a and 16.1c). This is due to the fairly regular workload and the small average work-item size. In this benchmark, the overhead for the dynamic item-to-thread mapping compromises the overall algorithm performance. However, *Multi-Phase* outperforms all the existing techniques, including the static ones. This is due to the reduced amount of overhead introduced by such a dynamic technique, which well applies also in case of very regular workloads.

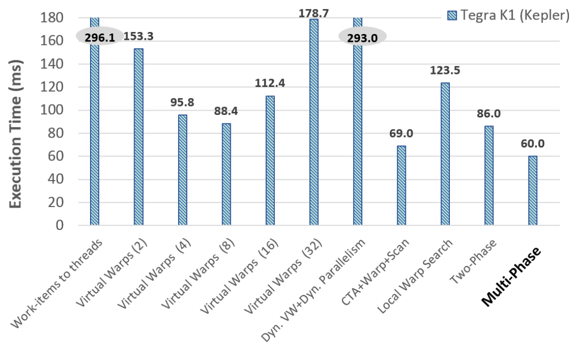
Figs. 16.1b and 16.1d report the average, maximum power and energy consumption of the load balancing applications for the same first benchmark. The static techniques show low average and maximum power on the GTX 980 device, while the semi-dynamic and dynamic techniques present the highest values, which are proportional to the technique complexity. The same characteristics show low variability on the Jetson TK1 device, except for *CTA+Warp+Scan*, due to the regular workload. On the other hand, *Multi-Phase* presents, on both devices, the lowest energy consumption, which is two times lower than the other dynamic techniques in most cases, at the cost of higher *peak* power in devices with multiple SMs like the Maxwell GTX980.



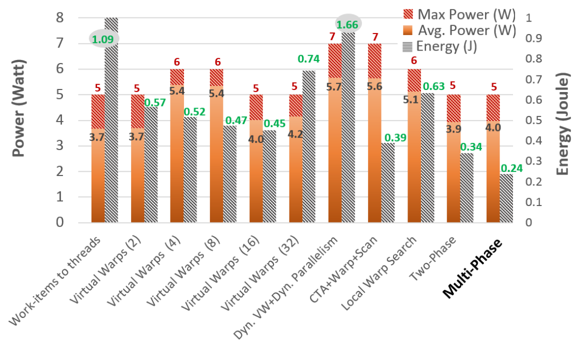
(a) *GTX980 - Execution Time*



(b) *GTX980 - Power and energy consumption*



(c) *Jetson TK1 - Execution Time*



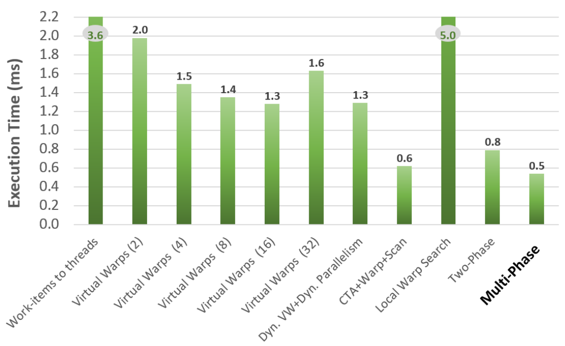
(d) *Jetson TK1 - Power and energy consumption*

FIG. 16.4: *Comparison of execution time, power and energy consumption on Circuit5M.*

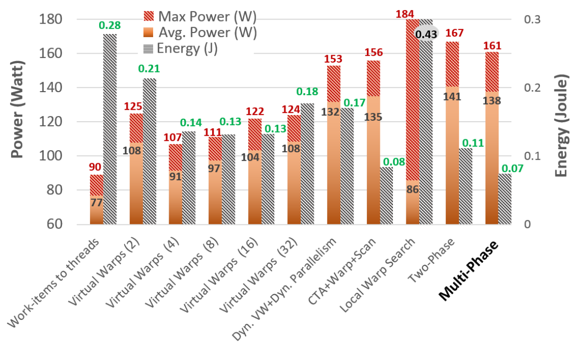
In the *web-NotreDame* benchmark, *Multi-Phase* is the most efficient technique and provides almost twice the performance with respect to the second best technique (*Virtual Warps* and three times faster than *Two-Phase* on GTX 980), while it shows performance comparable with *Virtual Warps(16)* on the Jetson TK1 (see Figs. 16.2a, 16.2c). It is important to note that *Virtual Warps* provides good performance if the virtual warp size is properly set, while it sensibly worsens with wrongly-sized sizes. The virtual warp size has to be set statically. For the obtained results in these two benchmarks, we noticed that the optimal virtual warp size is proportional and follows approximately the average of work-item sizes. In these first two benchmarks, *CTA+Warp+Scan*, which is one of the most advanced and sophisticated balancing technique at the state of the art, provides low performance. This is due to the fact that the CTA and the Warp phases are never or rarely activated, while the activation controls involve strong overhead.

The power and energy consumption (Fig. 16.2b, 16.2d) follows the behaviour of the first benchmark, but with lower values due to a lower number of benchmark work-units.

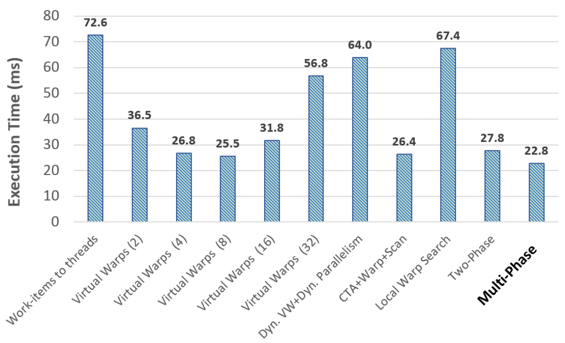
The efficiency of *Multi-Phase* becomes sharply evident as soon as the benchmark becomes more irregular, as for *Cit-Patents* and *Circuit5M* (see Figs. 16.3 and 16.4). In these benchmarks, we observed that the dynamic techniques



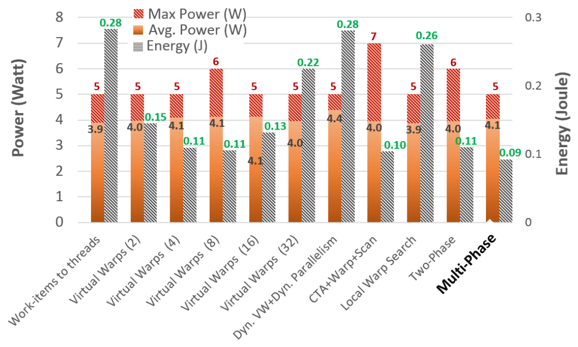
(a) *GTX980 - Execution Time*



(b) *GTX980 - Power and energy consumption*



(c) *Jetson TK1 - Execution Time*



(d) *Jetson TK1 - Power and energy consumption*

FIG. 16.5: *Comparison of execution time, power and energy consumption on Skitter.*

(*CTA+Warp+Scan*, *Two-Phase Search*, and *Multi-Phase*) are one order of magnitude faster than the static approaches in most cases. In these benchmarks, *Multi-Phase* shows the best results due to the low average (less than warp size) and high standard deviation.

In the *Cit-Patents* (Fig. 16.3b, 16.3d) and *circuit5M* benchmarks (Fig. 16.4b, 16.4d), *Multi-Phase* shows good values of average and maximum power consumption, which are comparable with the static-mapping techniques. On the other hand, *Two-Phase* and *Multi-Phase* techniques present the best power consumption on both the devices, which are three times lower on GTX 980 and two times lower on Jetson TK1 compared to the static techniques.

In the last benchmark, *as-skitter* (Fig. 16.5a, 16.5c), *Multi-Phase* and *CTA+Warp+Scan* provide the best results. *CTA+Warp+Scan* shows low execution time since the CTA and Warp phases are frequently activated and exploited. *Virtual Warps 16* and *Dynamic parallelism* techniques present quite good performance on GTX 980, while the overhead involved by the dynamic kernels heavily decreases the execution time on Jetson TK1 device.

As for *Great-Britain_osm* and *Web-Notredame*, the average and maximum power (Fig. 16.5b, 16.5d) of the dynamic techniques are higher than the static mapping ones. However, all the dynamic techniques, except for *Local-Warp Search*, show almost half energy consumption of the static techniques on GTX 980 and slightly lower on the Jetson TK1 device. This underlines the suitability of the dynamic approaches for application running on energy bounded environment.

Finally, we observed that the *Dynamic Parallelism* feature provided by the NVIDIA Kepler and Maxwell architectures, implemented in the corresponding semi-dynamic technique, finds the best application only when the work-item sizes and their average are very large. In any case, all the dynamic load balancing techniques, and in particular *Multi-Phase*, perform better without such a feature in all the analyzed benchmarks.

In general, we found that *Multi-Phase* provides the best trade-off between performance and power/energy consumption in all the benchmarks. This is due to the fact that such a dynamic technique implements a high-throughput energy-efficient workload balancing by minimizing the data movement throughout the memory space hierarchy[1], by exploiting fine-grained memory locality, and by organizing the computation at different memory hierarchy levels (shared memory, registers, caches).

16.3 Conclusion

This Section presented a survey of the most important and widely used load balancing techniques for GPUs. It summarized the main important aspects that characterize the overall complexity of each technique. This allows better understanding which one of them provides the best performance on different dataset characteristics. More importantly, the Section presented an analysis of average, peak power and energy consumption of each single technique over such a dataset. This allows considering additional dimensions in the technique evaluation, by providing a comprehensive trade-off between performance and power/energy consumption of each technique when applied on the different benchmarks and over different GPU architectures (i.e., NVIDIA Maxwell GTX 980 device, NVIDIA Jetson Kepler TK1 low-power embedded system).

[1] In GPU architectures, the off-chip global memory accesses consume a large amount energy, while on-chip memory accesses show lower latencies, higher bandwidth, and lower energy consumption.

Part IV

Dynamic Graph Processing

Introduction

Dynamic data structures are becoming increasingly important as many real-world problems evolve over the time. Sparse data computation and graph analytics can greatly benefit from dynamic features of underlying data structures that can be updated at high rates. On the other hand, as evolving data structures arise, also the algorithms build on top of them should be reconsidered and reformulated to fully exploit such dynamic behavior. In particular, in the context of dynamic graph analytics, it translates in avoiding redundant computation compared to reiterate the same process on a new snapshot of the static data structure. As consequence, implementations of dynamic graph algorithms may achieve execution times orders of magnitude lower respect to static counterparts.

This part of the thesis focuses on dynamic graph data structures and algorithms. First, in Section 17, it introduces some preliminary concepts and the related work. Then it describes an efficient data structure for dynamic graph analytics and sparse linear algebra for GPU architectures called *Hornet* (Section 18). Later, it presents how this data structure has been applied to a special case of subgraph isomorphism called*k-truss* (Section 19). Thanks to the great flexibility of the data structure, the dynamic algorithm has been designed to efficiently manipulate the graph without needing to rebuild it after each change.

17 Related Work

This section describes the related work on dynamic data structures and the main techniques in the literature to efficiently support graphs and sparse linear algebra problems which can evolve over the time.

17.0.1 Dynamic Sparse Formats

While some of the static solutions allow for graph updating, they can only support a limited number of updates, may have a large update time, or have an unacceptable overhead due to data structure re-allocation and re-initialization.

In order to fully support *dynamic* graph algorithms, more advanced and complex data structures have been recently proposed. They aim at *efficiently* supporting dynamic operations in graphs or matrices like edge/node insertions, deletions, or value/weight updates.

The *STINGER* data structure [95] was first introduced as a dynamic graph structure for both temporal and spatial graphs with meta-data (such as vertex and edge types) for multi-core architectures. It supports rapid graph updates at rates of 3 million updates per second [95]. It outperforms both shared-memory and distributed graph platforms and databases in supporting graph updates and analytics computation [182].

cuSTINGER [110] extends the *STINGER* data-structure to GPU applications. While the *STINGER* and *cuSTINGER* implementation of many features is similar (e.g., meta-data support, temporal support), their underlying data structures are very different. *STINGER* relies on blocked linked lists, while *cuSTINGER* uses arrays for the neighbor lists. cuSTINGER use of arrays improves locality during the parallel accesses to the memory and increases scalability.

GraphIn [239] and its extension for GPUs, *Evograph* [238], allow for incremental graph processing on CPU-based architectures by combining two static graph data structures: CSR for the original input and a dynamic edge-lists (COO) to store new edges. These frameworks are constrained to a limited number of updates (predefined by the users). Also, COO can lead to scattered memory accesses in case of large updates.

AIM [278] implements a block linked-list data structure for GPUs by using a *STINGER*-like data structure. It allocates a single array that is partitioned and used by an internal memory manager. *AIM significantly*[1] over allocates the amount of memory to ensure fast initialization and update rate. Such an allocation strategy allows for fast updates but, on the other hand, strongly limits the AIM adoption to support any analytics computation.

Dynamic CSR (DCSR) [144] is a CSR variant for supporting dynamic updates. When initialized, the DCSR format is nearly equivalent to the basic CSR representation with some additional storage overhead. Any update to the graph requires a *concatenation* to the initial CSR data structure. Concatenations involve significant memory overhead, they require knowing the number of updates a priori, and they require a data structure reorganization after each update.

[1] According to [278], *AIM* allocates the entire available GPU memory.

TABLE 17.1: Comparison of sparse graph and matrix representations. m_e represents the total number of available/extra edges in the graph. Insertions and deletions complexity is presented for single updates.

Format	**Storage**	**Duplicate checking**	**Insertion**	**Deletion**	**Re-set frequency**	**Memory reclama-tion**	**Fixed mem size alloca-tion**	**Support for additional graph properties**	**Notes**
CSR	$n+m$	/	/	/	For every update	No	Yes	SoA	
COO	$2\cdot(m+m_e)$	Enabled	$\mathcal{O}(m)$	$\mathcal{O}(m)$	After m_e updates	No	Yes	SoA	Poor locality
		Disabled	$\mathcal{O}(1)$	$\mathcal{O}(m)$					
GraphIn [239] CSR+COO	$n+m+2\cdot m_e$	No t supported	$\mathcal{O}(1)$	Not supported	After m_e updates or single deletion	No	Yes	Statically fixed	Reduced locality. Complex API.
Evograph [238] CSR+COO	$n+m+2\cdot m_e$	No t supported	$\mathcal{O}(1)$	Not supported	After m_e updates or single deletion	No	Yes	Statically fixed	Reduced locality. Complex API.
DCSR [144]	$2K*n+m+m_e$	No t supported	$\mathcal{O}(1)+\mathcal{O}(m)$	Not supported	After K batches or m_e edges	No	Yes	No	Complex API
AIM [278]	whole available GPU memory	Always enabled	$\mathcal{O}(deg_{max})$	$\mathcal{O}(deg_{max})$	Whenever exceed allocated memory	No	Yes	Statically fixed	
cuSTINGER [110]	$\mathcal{O}(n+2m+m_e)$	Always enabled	$\mathcal{O}(deg_{max})$	$\mathcal{O}(deg_{max})$	No	No	No	Statically fixed	
Hornet	$\mathcal{O}(n+2m)$	Enabled	$\mathcal{O}(1)^1$	$\mathcal{O}(1)^1$	No	Yes	No	Fully supportive and adaptive SoA/AoS	
		Disabled	$\mathcal{O}(1)^1$	$\mathcal{O}(1)^1$					

17.0.2 K-Truss

The k-truss was first introduced by Cohen [72] as a relaxation of a clique (due to the reduced complexity of the truss) while still ensuring that if two vertices are in a given truss it is quite likely their common neighbors will be in the truss. Several different approaches for finding trusses and the maximal k-truss are also discussed in [72]. Yet, these share common algorithmic properties: 1) the algorithm is executed in an iterative fashion and 2) in each iteration of the algorithm, a subset of edges is removed from the graph. Edges are removed from the graph if their support (i.e. number of triangles they participate in) is not large enough based on the given iteration.

In [73], Cohen discusses the benefits of implementing graph algorithms in the Map-Reduce framework which enables the analysis of large networks. It is worth noting that the work in [73] preceded the creation of the Pregel [177] framework. The introduction of Pregel improved expressibility and simplicity of implementing graph algorithms in a Map-Reduce framework, though performance of these algorithms did not improve as much. While the work in [73] showed the ability to scale to larger networks, the work by Wang and Cheng [272] showed that an optimized algorithm designed for a single shared-memory system can easily outperform the Map-Reduce implementation.

Wang and Cheng [272] show several different optimizations for finding the maximal k-truss , though these algorithms are sequential[2]. Further, Wang and Cheng [272] discuss several iterative approaches for finding trusses, including a bottom-up approach and a top-down approach. The bottom-up approach is closer to the approach taken by [72], whereas the top-down approach works in the reverse direction (starting from the edge with the largest number of triangles and working its way down). The top-down approach is ideal for cases when there is a need to find either the maximal k-truss or for k's close to the maximal k-truss . In practice, the top-down approach has a performance penalty making it more expensive than the bottom-down approach in many instances.

Sariyuce *et al.* [232] present a new approach for decomposing a graph into a forest of nuclei. These nuclei are dense sub-graphs with clique-like properties. One nuclei subgraph used in the decomposition is the maximal k-truss.

Kabir and Madduri [137] show a parallel algorithm for k-truss decomposition for multi-core systems. Their algorithm also uses an efficient triangle counting phase that avoids unnecessary graph intersections as well as two edge queues for storing the list of active edges in the graph. The active edges are used for updating the support of edges and filtering edges not matching the necessary support.

Gadepally *et al.* present the Graphulo [104] framework which enables implementing graph algorithm using linear algebra operators over the Apache Accumulo NoSQL database. The formulation for finding a k-truss in Graphulo is similar to the formulations of the baseline benchmarks of the HPEC Graph Challenge, which are implemented in Matlab, Julia, and Python using highly optimized libraries.

[2] One of the main challenges associated with a parallel implementation of their algorithms is the need for correct triangle counting in parallel and dynamic environment. A solution to the problem was recently given in [175] and is discussed in additional detail in Section 17.0.5.

The linear algebra based algorithm in [104] presents one iteration of [72, 272] for a specific k, though this can be extended to find the maximal k-truss . Huang *et al.* [130] show how to maintain the various trusses of a graph in a dynamic environment.

17.0.3 Triangle Counting

Triangle counting is a building block for numerous applications. Therefore, it is not overly surprising that numerous algorithms and optimizations have been designed to efficiently compute it. Some libraries and implementations have focused on good system utilization with good load-balancing [111, 273], others have focused on data scalability to support larger graphs GraphX [283], GraphLab [167]. Techniques such as vertex re-ordering have been shown to help reduce the number of cache misses [220, 243]. Other algorithms have used vertex covers to reduce the number of necessary intersections [109].

17.0.4 GPU Triangle Counting

Leist *et al.* [157] show the first GPU algorithm for triangle counting. In this approach each GPU thread is responsible for a different intersection. Green *et al.* [112] offer a different parallelization scheme for the GPU that uses numerous GPU threads for each adjacency intersection and extends the Merge-Path formulation [113, 214] to Intersect-Path. Intersect-Path improves the performance over [157] by an order of magnitude. Wang *et al.* [273] analyze the performance of several different approaches for triangle counting on the GPU.

17.0.5 Streaming and Dynamic Triangle Counting

Similar to the static graph triangle counting algorithms, numerous algorithms have been designed for streaming graphs [34, 55, 152]. Streaming graphs are graphs were the edges are inserted or removed one at a time (typically at high rates) and the number of vertex and edges memory accesses per update is limited to $O(1)$ operations. In the case of triangle counting, many streaming graph algorithms focus on approximating the number of triangles. Furthermore, many streaming graph algorithms focus on the easier case of edges insertions [152]. Becchetti *et al.* [34] note that there are numerous applications where these approximations are not good enough - this is also true for the case of finding the exact and largest k-truss in a graph.

In addition to streaming graphs algorithms, dynamic graph triangle counting algorithm can be found in [94, 175, 239]. Ediger *et al.* [94] use the STINGER [95] dynamic graph data structure for updating the graph and analytics in batches. For a single update this is simple, however, then the update consists of multiple edges (combined into a single batch) a situation can arise where numerous edges in a batch can create a triangle - such a triangle can go undetected in a parallel environment. Therefore the approach taken in [94] and GraphIn [239] is to recompute the triangles of a vertex from scratch even if only one of its edges is affected.

Recently, a dynamic graph triangle counting algorithm was presented by Makkar *et al.* [175] that shows a new inclusion-exclusion formulation for detecting triangles within a given batch, thereby reducing the amount of work required to update the number of triangles per vertex. This new algorithm does not require recomputing the number of triangles for a whole vertex as required by previous approaches. This algorithm, with its ability to support a batch of edge deletion, is extremely useful for finding the k-truss .

18

Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices

Sparse data computations are ubiquitous in science and engineering. Unlike their dense data counterparts, sparse data computations have less locality and more irregularity in their execution, making them significantly more challenging to optimize. Even more challenging is their optimization for parallel applications. Most of the existing formats for sparse data representations on parallel architectures are restricted to static data sets, while those for dynamic data suffer from inefficiency both in terms of performance and memory footprint. This work presents Hornet, a novel data representation that targets dynamic graph analytics and linear algebra based problems. While the data structure API is simple for users, the data structure includes an optimized memory manager that is responsible for memory allocation, reclamation, and for ensuring low overhead to represent the data during its evolution. Hornet is scalable with the input data, flexible in representing data set properties, and does not require any data re-allocation or re-initialization during the data evolution. The Section presents a Hornet implementation for GPU architectures, and it shows the experimental results obtained on several representative sparse graphs and matrices. It analyses and compares the proposed solution with the most important static and dynamic data structures at the state of the art in terms of memory utilization efficiency, initialization time and update rates.

18.1 Introduction

Dynamic sparse data applications are now ubiquitous and can be found in many domains. *Dynamic* refers to the fact that the data is changing at very high rates. For example, updates might represent the change in the current-flow of a power network or the road-congestion for a transportation network. The number of applications is considerably high and many formulations of these problems end up being either graph-based or linear-algebra based. The sparsity of the data has led to the development of several data representations, which are common for both problem formulations: Compressed Sparse Row *(CSR)*, Coordinate (*COO*), Compressed Sparse Column (*CSC*), and ELL (*Ellpack*). Unlike a dense adjacency matrix, which may be potentially filled with “0”-values, these formats avoid storing these trivial values. As such, these data-structures are cost-effective in terms of memory yet lack the flexibility to support growth. While a linked-list can also offer similar memory bounds and give the flexibility to change the graph (by adding or removing nodes from the linked-list), in practice the lists lack of locality and the irregularity of the accesses limit its scalability and performance.

As real-world networks continue to grow at extremely high-rates, the issue of whether the problem is presented as a graph formulated problem or in a linear-algebra formulation becomes of second-order. The main challenge is to ensure that the data structure for any sparse dataset can be updated at real-world rates. In this context, even though some attempts have been recently done to design a data structure that is scalable, high-performing, and flexible enough to support rapid updates [20, 95, 238, 278], they have shown to be unable to meet all three criteria.

This Section presents *Hornet*, a data structure for efficient computation on dynamic sparse graph and matrices. It relies on an internal memory manager that is responsible for both the memory allocation and memory reclamation. This allows the data structure to grow to very large sizes (i.e., over $1,000x$ the initial dataset) without requiring any data re-allocation or re-initialization during the whole dynamic evolution of data.

Due to the internal memory organization it outperforms dynamic graph data structures on several fronts: *Hornet* provides better memory utilization than AIM [278] and cuSTINGER [110], faster initialization (from $3.5x$ to $26x$), and faster update rates (over 200 million updates per second) than the state of the art dynamic data structures. It comparison to several widely used and static graph data structures, Hornet also offers good storage utilization and efficient initialization. While Hornet may require 5% to 35% additional memory in contrast to CSR, it also uses 30% less memory than COO. Initializing Hornet is $1.7x$ to $12.7x$ slower that the initialization time of CSR; however, this can be considered negligible for dynamic graph analytics running for an extended amount of time.

Hornet is also designed to support *property graphs*, that is, graphs in which meta-data (i.e., structured data) can be associated to each vertex and edge. This allows for great flexibility and applicability when the graph properties (number and type of information per edge/vertex) cannot be know a-priori.

The Section presents the *Hornet* implementation for GPU architectures, the experiment analysis, and its comparison with the state of the art dynamic approaches. On the other hand, it underlines that *Hornet* does not have any architecture dependencies that limit its application only to GPUs.

The work is organized as follows.Section 18.2 presents the *Hornet* data structure and its implementation for GPUs. Sections 18.3 and 18.4 present the experimental setup and a detailed empirical analysis, respectively. Section 18.5 is devoted to the concluding remarks.

18.2 Hornet Overview

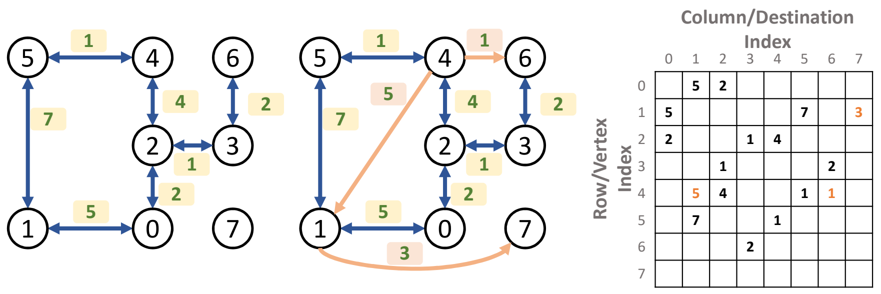
The *Hornet* data structure has been designed with both *dynamic* graphs and *property* graphs in mind[1]. Unlike the state of the art data structures, it efficiently

[1] Due to the different terminologies used by these applications, we have the terms used for *graphs*.

TABLE 18.1: List of symbols and notations.

Symbol	**Description**
$G(V, E)$	Input graph.
V, E	Vertices/Edges in input graph.
n, m	Number of vertices/edges.
B	Batch updates (list of edges).
$adj(v)$	Adjacency list of vertex v.
$deg(v)$	Degree of vertex v.
deg_{max}	Maximum degree of a vertex in a graph.
$BA_{k,id}$	*Block array* for vertices s.t. $2^{k-1} \leqslant deg(v) < 2^{k}$
$\|BA\|$	Number of *blocks* within a *block array*.
$\text{BSIZE}(v)$	Number of edges (*block*) size allocated for v
$v.ptr$	A pointer to the edge list of vertex v

Notations	***Hornet* Data Structure fields - User's API**
$v.used$	Degree of the vertex v
$v.pointer$	A pointer to the edge list of vertex v



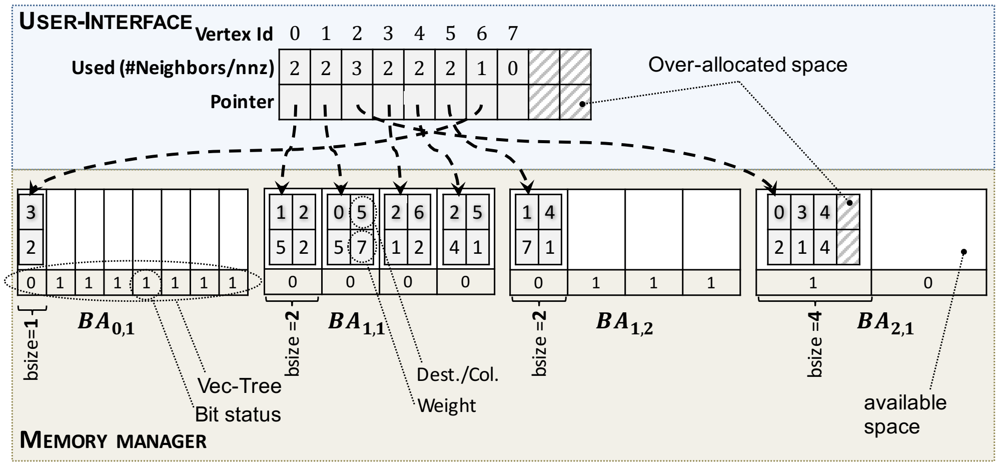
(a) *Initial input graph.* (b) *Graph after changes.* (c) *Matrix representation of graph.*

FIG. 18.1: *Graph example.*

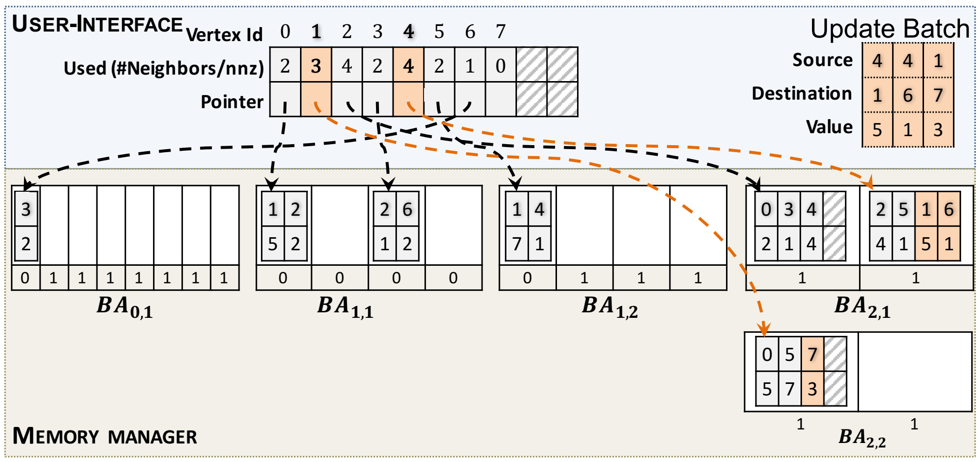
supports evolving graphs by storing additional meta-data per edge. While storing meta-data per edge is fairly simple for static graphs (i.e., it requires allocating an array of size $|E|$ edges), it is far more challenging and crucial for dynamic graphs where $|E|$ is constantly changing as is are the indices of the edges in an array. Hornet resolves this problem by allowing each to store additional needed meta-data.

Fig. 18.2 gives an overview of *Hornet*, which consists of two layers: The user interface and a memory manager. Such a double-layer structure allows providing a very simple API to the user; while hidden to the user is an advanced and optimized memory manager.

From the user's perspective, each vertex is associated with two main fields: The *number of current neighbors* (i.e., *Used* in the figure) that represents the adjacency list size, and a *pointer* to a dedicated adjacency list. Instead of using standard



(a) *The initial graph of Fig.* **??***(a).*



(b) *The updated graph of Fig.* **??***(b).*

FIG. 18.2: *Hornet layout.*

memory allocation function calls for each adjacency list, which would be extremely inefficient[2], *Hornet* implements a dedicated memory manager. It consists of three main components: *Block arrays* for storing multiple adjacency lists, a *vectorized bit tree* for efficiently finding empty memory *blocks* for the adjacency lists, and B^+ *trees* to manage the *block arrays*.

[2] On modern systems, memory allocation function calls like `malloc` or `cudaMalloc` can take from a few hundred microseconds to several milliseconds, which would make the approach infeasible for graphs with millions of vertices.

18.2.1 Block arrays

Hornet represents the graph through a hierarchical data structure, which consists of an adjacency lists, *blocks*, and *block arrays.* A *block array* is an array of equally-sized memory chunks, called *blocks.* Each *block* contains a number of adjacency lists equal to a power of two (we refer to this number as the *bsize*). Blocks sizes are $2^b size$. The bsize for each vertex, v is determined as follows:

$$bsize(v) = 2^{\lceil log_2(deg(v)) \rceil} \tag{18.1}$$

Fig. 18.2(a) shows, as an example, the *Hornet* layout of the initial graph of Fig. 18.1(a), which consists of four *block arrays*: $BA_{0,1}$ (*bsize=1*), which contains one adjacency list; $BA_{1,1}$ and $BA_{1,2}$ (*bsize=2*), which contain four and one adjacency list, respectively; $BA_{2,1}$ (*bsize=4*), which contains one adjacency list.

Fig. 18.2(b) shows the *Hornet* layout after the insertion of the three new edges of Fig. 18.1(b) (the details of the insertion procedure are discussed in Section 18.2.5). The insertion of edge $1 \rightarrow 7$ involves increasing the size of the adjacency list for vertex 1 as it cannot store additional edges in it's pre-allocated *block* in $BA_{1,1}$. As a consequence, the memory manager allocates a new *block array* for blocks of $bsize$ = 4 ($BA_{2,2}$) and moves the whole *block* containing the adjacency list in it.

By placing adjacency lists in blocks using the *bsize* mechanism discussed above, we can place an upper bound on the amount of space allocated for each adjacency list. As a consequence, it allows identifying the worst case upper bound of memory allocated for the entire graph evolution: $2 \cdot |E|$. In practice, the average memory allocated for the graph edges, as shown in Section 18.3, is close to $1.4 \cdot |E|$.

Differently from cuSTINGER [110], the memory allocation scheme implemented in *Hornet* does not allow for a full control over the amount of memory used since, as experimented in cuSTINGER, such a full control would lead to slower updates and memory fragmentation.

Finally, the number of *blocks* in a *block array* is also a power of two. Section 18.2.3 discusses more in details the trade-off between different sizes of *blocks* and *block arrays.* In Section 18.3 we show the difference in memory utilization as a function of the *block array* size.

18.2.2 Vectorized Bit Tree

Sizing the block arrays in a power-of-two number of blocks as well as run-time deletions of vertices/edges leads to *empty blocks* (white spaces in Fig. 18.2(a), (b)). The *Hornet* memory manager relies on the *vectorized bit tree* data structure (*Vec-Tree* in the following) to efficiently use such empty blocks for new allocations. The implemented data structure allows the memory manager to fulfill three key requirements: 1) ensuring that a new block array is not allocated until all block arrays for a given block size are fully utilized, 2) involving a reduced memory footprint for the Vec-tree implementation, and 3) finding an empty block in an efficient manner. *Hornet* satisfies the first requirement by associating one Vec-Tree per block array. Each Vec-Tree consists of a tree of boolean values in which each tree node stores the value of the *logic OR* of its two children. The leafs of the

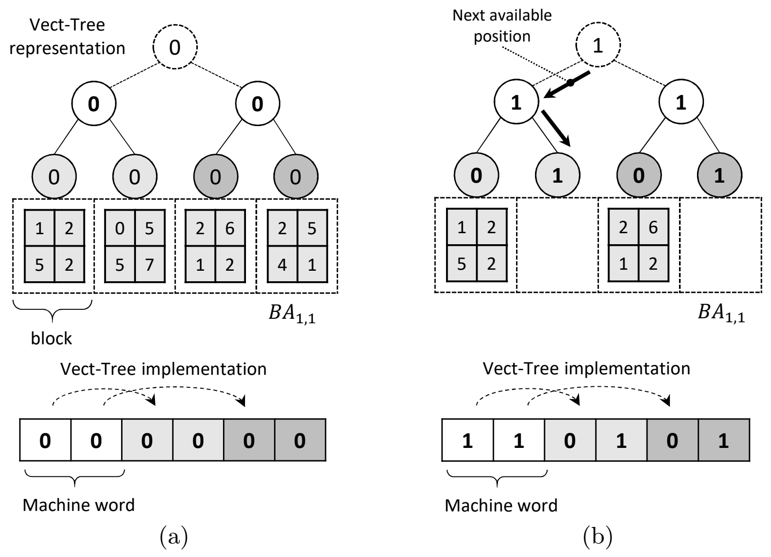


FIG. 18.3: *Vectorized Bit Tree of block array $BA_{1,1}$. The figure shows the data structure before (a) and after (b) the batch update. The top part of each subplot illustrates the representation, while the bottom size the actual* "vectorized" *implementation.*

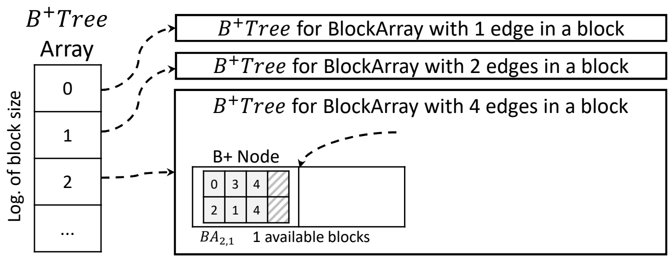
tree represent the state of the blocks (1 is empty, 0 if used). Fig. 18.2 shows the Vec-Trees of all block arrays before and after the graph update, while Fig. 18.3 shows in details the representation and actual vectorized implementation of the Vec-Tree of $BA_{1,1}$ before and after an update.

In general, starting from the Vec-Tree root, the memory manager checks whether there is at least one free block within any block array in $\mathcal{O}(1)$ steps. It finds the actual free block within $\mathcal{O}(log_W(|BA|))$ steps, where W is the machine word size (this satisfies the third requirement). The same time is spent for an empty block reclamation. Assuming block i as the block of interest, the address of block i is calculated as follows:

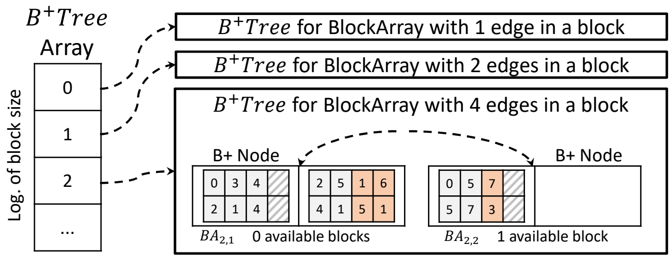
$$address(i) = address(BA_{k,id}) + i \cdot 2^k \qquad (18.2)$$

where 2^k is the size of each block.

Such a reduced complexity of both find operations is particularly important to guarantee high update rates even in cases of large block arrays. The memory overhead introduced by the Vec-Trees can be calculated as follows. For a block array with $|BA|$ blocks, the corresponding Vec-Tree requires: $|BA|$ bits for the lowest level of the tree, $|BA|/2$ for the second lowest level, and so forth upto the root. In total, a Vec-tree requires $2|BA| - 1$ bits. Overall, the storage requirements for the Vec-Trees is negligible in comparison to the memory required by the blocks themselves (i.e., one bit vs. information on 2^i edges for storing destination, weights, and additional meta-data per block).


B^+Tree Array
Log. of block size
0
1
2
...
B^+Tree for BlockArray with 1 edge in a block
B^+Tree for BlockArray with 2 edges in a block
B^+Tree for BlockArray with 4 edges in a block
B+ Node
0 3 4
2 1 4
$BA_{2,1}$ 1 available blocks

(a) *Layout of the initial graph of Fig. 18.1(a).*


B^+Tree Array
Log. of block size
0
1
2
...
B^+Tree for BlockArray with 1 edge in a block
B^+Tree for BlockArray with 2 edges in a block
B^+Tree for BlockArray with 4 edges in a block
B+ Node
B+ Node
0 3 4 2 5 1 6
2 1 4 4 1 5 1
0 5 7
5 7 3
$BA_{2,1}$ 0 available blocks
$BA_{2,2}$ 1 available block

(b) *Layout after the update process of Fig. 18.1(b).*

FIG. 18.4: *Block array memory manager with an emphasis on blocks of size 4 edges (also referred to as $BA_{2,i}$).*

18.2.3 B$^+$Trees of block arrays

Beside the Vec-Tree layer that allows reclaiming empty blocks on a specific block array, the memory manager uses the *B$^+$ Tree* data structure, which allows finding empty blocks over different block arrays. *Hornet* allocates an array of B$^+$Trees, where each B$^+$Tree (one for each block size) manages all the block arrays of a given block size. Fig. 18.4 shows the B$^+$Tree array for the example of Fig. 18.2 (initial and after the update), which consists of three B$^+$Tree for blocks of size 1, 2, and 4.

Each node of a B$^+$Tree consists of a couple `<data, key>`. The `data` field points to the block array and the `key` stores the number of free blocks within that block array. Searching for empty blocks in a B$^+$Tree takes logarithmic time with respect to the tree size. Considering that the size of block arrays is generally big (see Section 18.2.1), the number of block arrays of a given block size is relatively small. This means that the lookup operations are extremely fast and that, in general, the overhead of this data structure is relatively low. As a consequence, when a new block is needed, rather than iterating through all the block arrays and their corresponding Vec-Trees, the memory manager queries the B$^+$Trees to find a block array with an empty block. Several B$^+$Trees implementations already exist in literature and offer this search (e.g., [41, 133]).

18.2.4 Data structure initialization

Hornet allows for graph initialization by starting from an empty data structure and by adding edges and vertices one at a time. It also supports the initialization

Algorithm 13 Pseudo-code for the Hornet initialization.

```
    Input: $G_{CSR}$ : CSR representation of graph $G(V, E)$,
           MaxBASize : maximum number of edges in a block array
    Output: Hornet graph hgraph
          → CopyAdjcencyList(SRC, DEST, SIZE)
          → CopyToDevice(SRC, DEST, SIZE)
 1: CreateMemManager(Device: GPU, MaxBASize)
 2: for all v ∈ $G_{CSR}$ do
 3:     ⟨host_ptr, dev_ptr⟩ = MemManager.getEmptyBlock(deg(v))
 4:     CopyAdjcencyList(v.ptr, host_ptr, deg(v))
 5:     deg_array[v]      = deg(v)
 6:     dev_prt_array[v] = dev_ptr
 7: end
 8: for all  BA ∈ MemManager  do          → BA : Block array
 9:     CopyToDevice(BA.host_ptr, BA.device_ptr, MaxBASize)
10: end
11: CopyToDevice(deg_array, hgraph.used, $|V|$)
12: CopyToDevice(dev_prt_array, hgraph.pointer, $|V|$)
13:         → the actual implem. copies the vertex data in a single operation
```

by starting from a CSR representation and by converting such a static format into the *dynamic-ready Hornet* format. Alg. 13 shows the pseudo code of such a process and focuses on the memory allocation by using the whole hierarchy of data structures presented in the previous sections. It is important to note that the pseudo code is architecture independent and applies to both CPU and GPU architectures.

The data structure initialization consists of three steps. First, for all vertices in the graph, the memory manager finds empty *blocks* for the corresponding adjacency lists (`lines 2-6`). If such blocks are the first of their block sizes or all the previous block arrays are full, the memory manager allocates a new block array. In this phase, for performance reasons, all the adjacency lists are temporarily stored in block arrays and maintained in the host-side (`lines 2-6`) rather than being directly copied to the device memory. In `lines 7-8`, all block arrays are actually copied to the device. Copying the whole block array instead of single blocks greatly improves the initialization time, since it avoids many small memory transfers yet maximizing the PCI-Express bandwidth. Finally, also the vertex data (degree and adjacency list pointers) are copied to the device (`lines 9-10`).

18.2.5 Dynamic Updates

Hornet supports different graph updates: (a) insertion and deletion of vertices, (b) insertion and deletion of edges, and (c) update of values of existing vertices and edges. The first two types (a, b) change the graph topology, while the last one only changes the data values of the network. Vertex insertions and deletions are implemented through series of edge insertions and deletions, respectively.

Hornet supports graph updates through *batches* [95, 110, 144, 238, 239, 278], by which different updates are grouped into a batch to maximize the throughput and to avoid the latency involved by the sequential updates.

[3] The *Hornet* implementation is based on the binary-search load balancing algorithm [63].

Algorithm 14 Pseudo-code for updating the data-structure after a batch of updates. The pseudo-code for deletions is almost identical to the insertion code by replacing line 4-5.

```
1: $Q \leftarrow$ empty queue            $\rightarrow Q : \langle$ old_ptr, new_ptr, size $\rangle$
2: $\hat{B} \leftarrow$ CSR representation of $B$        $\rightarrow$ require sorting: $O(B \cdot log(V))$
3: parallel for v $\in \hat{B}$ do          $\rightarrow O(B)$
4:     new_degree $\leftarrow$ hgraph[v].used + $deg_{CSR}$(v)
5:     if (new_degree > BSIZE(hgraph[v].used) then
6:         new_ptr $\leftarrow$ MemManager.GETEMPTYBLOCK(new_degree)
7:         ENQUEUE ($\langle$ hgraph[v].pointer, new_ptr, hgraph.used[v] $\rangle$, $Q$)
8:         hgraph.used[v]    $\leftarrow$ new_degree
9:         hgraph.pointer[v] $\leftarrow$ new_ptr
10:
           $\rightarrow$ Load-balancing is required for efficient copies$^3$.
11: parallel for q $\in Q$ do          $\rightarrow$ COPYADJCENCYLIST(SRC, DEST, SIZE)
12:     COPYADJCENCYLIST(q.old_ptr, q.new_ptr, q.size)
13: for q $\in Q$ do          $\rightarrow O(B)$
14:     MemManager.RECLAIMOLDBLOCK(q.old_ptr)
15: parallel for v $\in \hat{B}$ do          $\rightarrow$ Only for batch insertion          $\rightarrow O(B)$
16:     COPYADJCENCYLIST(v.ptr, hgraph[v].pointer, $deg_{CSR}$(v))
```

```
Deletion:
4: new_degree $\leftarrow$ hgraph[v].used - v.degree
5: if (new_degree < BSIZE(hgraph[v].used) / 2 then
6:
```

Algorithm 15 Pseudo-code for batch duplicate removing.

```
Input: hgraph : Hornet graph,
Input: $B$ : Update Batch,
Output: $\tilde{B}$ : batch without cross duplicates
1: parallel for (u,v) $\in B$ do          $\rightarrow$ Compute the workload
2:     deg_array[u] = $deg_G$(u)
3: sum = PARALLELEXCLUSIVEPREFIXSUM(deg_array)
4: Parallel for thread_id from 0 to sum do          $\rightarrow$ LB: Load Balancing
5:     batch_pos, offset = BINARYSEARCHLB(deg_array, thread_id)
6:     src = $B$[batch_pos].src          $\rightarrow$ offset $\in$ [0, deg(src)]
7:     if ($adj_{hgraph}$(src)[offset] = $B$[batch_pos].dst) then
8:         MARK(flag_array, batch_pos)          $\rightarrow$ mark duplicate
9:
10: end
11: FLAGGEDCOMPACT($B$, flag_array)          $\rightarrow$ Compact the batch by flags
```

Algorithm 14 shows the pseudo-code for completing an update batch for *edge insertions*. The insertion of new elements in the structure consists of several important yet *parallel* phases. First, all the single operations in the batch are sorted to improve locality during the updates and to count the number of appearances for each row/source (the update batches are converted to CSR format). Then, the vertices requiring additional storage (e.g., vertices 1 and 4 in Fig. 18.2(b)) are enumerated and queued. A new block is allocated for each of them ($BA_{2,1}$, $BA_{2,2}$), the contents of the old blocks are copied into the corresponding new blocks in parallel, the old block pointers are given back to the memory manager, and the pointers are updated.

The algorithm for edge deletions is implemented in a similar manner, and can be obtained by replacing lines 4 and 5 in Algorithm 14 with the lines placed at the bottom of the pseudo code.

Differently from the state-of-the-art solutions for dynamic updates (*cuSTINGER*, *DCSR*, *AIM*), *Hornet* does not rely on expensive atomic operations to update the vertex degrees (see `Algorithm 2: line 8`). In addition, implementations in literature lead to sequential and random memory accesses when the adjacency lists are read; in contrast *Hornet* involves only coalesced memory accesses thanks to advanced load-balanced copy operations (see `Algorithm 2: lines 12, 16`).

Like other approaches in literature (cuSTINGER and AIM), *Hornet* supports *cross duplicate removing* between a batch and the target graph. The goal is to ensure that the final graph, after the update process, does not contain edge duplicates, which may lead to wrong results in the computation of important graph algorithms (e.g. triangle counting or betweenness centrality).

Algorithm 15 shows the pseudo-code of this process, which takes as input the update batch and the graph, for removing the cross duplicates. Given a single edge, the basic idea is to span a set of threads (with continuous ids) equal to the degree of the edge source. Each thread maps to a different element in the adjacency list of the source vertex (coalesced accesses) and checks if a duplicate is present ($adj(src)[i] = batch\_edge.dst$). In such a case, the edge is marked to be removed. The pseudo-code shows the entire parallel procedure for a batch of edges. At the beginning, the workload is computed for all edge sources in the batch (`lines 1-3`). In `lines 4-8`, all duplicate edges are marked in an external array. Finally (`line 9`), the batch is in compacted in parallel using parallel prefix operations.

Lastly, unlike the other state of the art solutions, *Hornet* implements the removal of *intra-batch* duplicates (i.e., edge duplicates within a batch) by sorting the edges within a batch. Note, the sorting operation is already applied in Algorithm 14 (`line 2`) and both procedures expose high parallelism and efficiency.

18.2.6 Handling graphs with extra properties

Property graphs are graphs in which meta-data (i.e., structured data) can be associated to each vertex and edge rather than a single value. Implementing the support for handling such extra properties is fairly simple for static graphs as it requires duplicating the data structure used for storing edge weights. In contrast, duplicating the data structure in dynamic graphs, while the data is *evolving* (e.g. edge insertion timestamps) is far more challenging. Table 17.1, second to last column, summarizes how the different data structures, in particular for dynamic graphs, can support property graphs. Unlike dynamic solutions like *AIM*, *STINGER*, and *cuSTINGER*, which require the additional properties (number and type) to be known and set statically, *Hornet* implements a full support and adaptive system for extra properties. *Hornet* allows associating an *arbitrary* number of additional fields to each edge or vertex. They are implemented as additional rows to the block data structures, which are allocated and handled by the memory manager.

Given the set of additional fields, *Hornet* automatically morphs this internal representation by selecting the best data layout at compile time. For example, in the context of GPU computing, *Hornet* adopts an Array-of-Structure (AoS) if the data types associated with the additional fields of an edge or vertex can be vectorized (single memory transaction for multiple data), otherwise it uses

TABLE 18.2: Graphs and matrices used in the experimental results.

Matrix/Graph	Source	Context (Matrix/Graph)	Symm.	Rows, Vertices (M)	NNZ, Edges (M)	Avg.
dblp-2010	[85]	Collaboration (G)	Y	0.03	1.6	5.0
Cantilever	[276]	FEM (M)	Y	0.06	4.1	65.2
Protein	[276]	Protein (M)	Y	0.03	4.3	120.3
Spheres	[276]	FEM (M)	Y	0.08	6.1	73.1
Ship	[276]	FEM (M)	Y	0.14	7.6	56.5
Wind	[276]	Wind tunnel (M)	Y	0.21	11.6	54.4
in-2004	[85]	Web crawl (G)	N	1.38	16.7	12.2
soc-LiveJournal1	[161]	Social Network (G)	N	4.85	69.0	14.2
cage15	[85]	DNA (G)	N	5.15	99.2	19.2
europe_osm	[85]	Road (G)	Y	50.91	108.1	2.1
kron_g500-logn21	[85]	Synthetic (G)	Y	2.1	182.1	86.8
indochina-2004	[85]	Web crawl (G)	N	7.41	194.1	26.2
uk-2002	[85]	Web crawl (G)	N	18.5	298.1	16.1
com-livejournal	[161]	Ground-truth comm. (G)	Y	4.00	69.3	17.3
com-orkut	[161]	Ground-truth comm. (G)	Y	3.07	234.3	76.2

Structure-of-Array (SoA). This guarantees high flexibly and the best performance while accessing the data.

18.3 Experimental Results

While *Hornet*, as underlined in the previous sections, is architecture independent, in this Section we present the experimental results of its implementation for GPU architectures. The *Hornet* implementation for multi-core architectures is part of ongoing work.

We conducted the efficiency analysis and the comparison with the corresponding state of the art data structures for GPUs. We considered different key factors for the evaluation, which include *memory utilization*, *initialization time*, and *update rates*.

The experimental analysis has been conducted on a NVIDIA Tesla P100 device (Pascal micro-architecture) with Xeon E5-2650 v4 host processor. The CPU runs at 2.8GHz with 30MB L3 cache. The P100 consists of 56 SMs with a total of 3,840 CUDA cores and 16GB DRAM memory.

Table 18.2 reports the set of sparse graphs and matrices used in the experiments and their main characteristics. They have been taken from the University of Florida Sparse Matrix Collection [85], the SNAP dataset [161] and the sparse matrix problems used by Williams et al. [276].

18.4 The *Hornet* data structure

18.4.1 Memory utilization efficiency

The *Hornet* memory utilization has been evaluated and compared with *cuSTINGER*, which is the reference dynamic data structure for sparse datasets

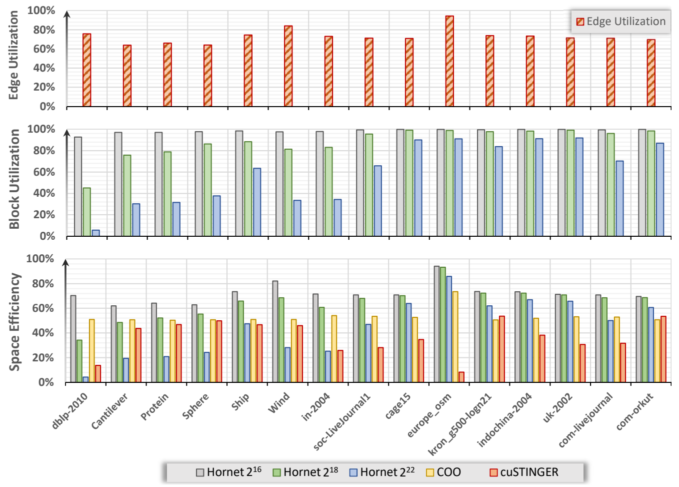


FIG. 18.5: *Upper-side: block-level fragmentation analysis. Middle-side: Fragmentation analysis at block array level. Bottom-side: Overall memory utilization efficiency.*

that does not rely on static user-defined memory allocation scheme (as required by *GraphIn*, *DCSR*, and *AIM*). Instead, we compare the memory utilization to two efficient static graph data structures (*CSR* and *COO*). This allows us to understand the total amount of memory overhead involved by the dynamic feature w.r.t. any format with static allocation.

We first analyzed the *block-level* fragmentation, that is, the unused edges within the blocks due to the power-of-two block sizing (Eqn. 18.1). Fig. 18.5 (upper subplot) reports the results, in which 100% represents no fragmentation, the bar value represents the total memory utilization in the allocated blocks (e.g., 78% for *dblp-2010*), while the difference (22% for *dblp-2010*) represents the over-allocated memory. The results are identical for all block array sizes. As formulated by the analysis of Section 18.2.1, the over-allocated memory never goes beyond twice the number of edges (i.e., below the 50% utilization). For most cases, the bock-level fragmentation is less than 30%. It is important to note that such an over-allocated memory can be used by the memory manager during dynamic operations (i.e., edge insertions) or deallocated during any memory reclamation step.

We then analyzed the fragmentation at *block array level*, that is, the unused over-allocated memory within block arrays due to empty blocks (see Section 18.2.2). Fig. 18.5 (middle subplot) reports the results, by considering different *block array* sizes: $2^{16}, 2^{19}$, and 2^{22} edges. As expected, the more the block array size increases, the more the fragmentation increases (storage utilization decreases). This is especially evident for the smaller graphs. For large graphs, the utilization of the allocated memory is usually well over 80% as the vertices are well balanced across the multiple block arrays. On the other hand, as shown in Section 18.4.3, the more the block array size increases, the better the dynamic update rate.

Finally, Fig. 18.5 (bottom subplots) shows the comparison between *Hornet*, CSR, COO, and *cuSTINGER* in terms of overall memory utilization efficiency. We chosen, as reference point, the memory utilization of *CSR*, since it is the

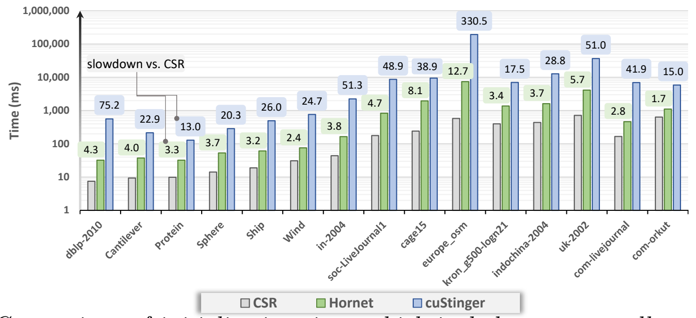


FIG. 18.6: *Comparison of initialization time, which includes memory allocation on GPU and CPU-GPU data transfer time (Hornet with block array size=2^{16} edges).*

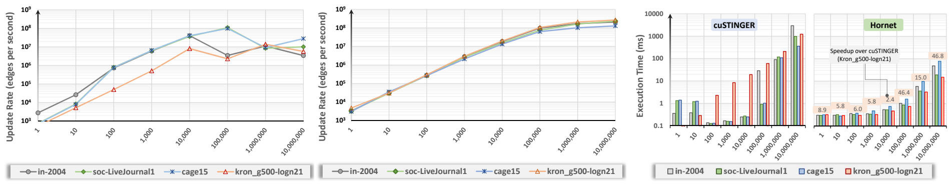
most compact state of the art data structure. *CSR* is represented by the 100% utilization in the figure. We also considered *COO* as it is the second most efficient data structure in terms of memory utilization with some support (even though very limited) for graph updates.

The overall comparison of Fig. 18.5 underlines that *Hornet* strongly improves (almost twice) the memory utilization efficiency with respect to the best dynamic counterpart at the state of the art (*cuSTINGER*). It also shows that, if properly configured, *Hornet* provides better memory efficiency then the static *COO*.

The results also underline that the total memory utilization (bottom subplot) is proportional to the *block-level* fragmentation (upper side plot). This is especially clear with *Hornet* 2^{16}, for which the block array utilization is close to 100%. Such a correlation underlines that, despite the high utilization of the block arrays, the overall memory utilization strongly depends on the block utilization (middle subplot).

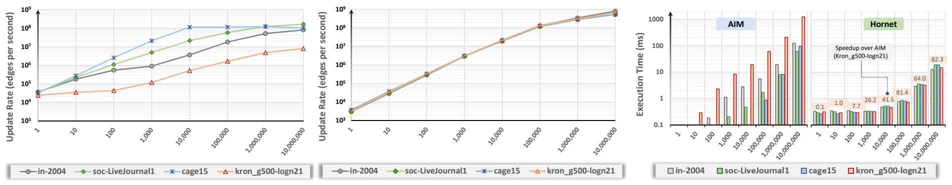
18.4.2 Initialization Time

Fig. 18.6 shows the results of the initialization time analysis. The time includes both the memory allocation on the GPU and the host-to-device data transfer. For *Hornet* and *cuSTINGER*, the time also includes all the overhead involved by the additional operations for initializing the dynamic data structure. Fig. 18.6 that *Hornet* is significantly faster than *cuSTINGER* - one order of magnitude faster in numerous instances. Also, the *Hornet* initialization is on average less than $4x$ more expensive than *CSR* initialization. This overhead can be considered negligible for many graph analytic applications. In addition, the *Hornet* initialization is a one-time penalty that enables the support for dynamic graphs, which can grow, shrink, and be updated for an indefinite amount of times without requiring any reset or re-allocation. For the sake of space and clarity, the figure does not include the initialization time of *GraphIn*, *DCSR*, and *AIM* since, similarly to the static counterparts, they allocate a static (and user-defined) amount of memory and requires reset and re-initialization as the graph evolves. Their initialization time is close (but not faster) to that of *CSR*. In addition, the *AIM* allocation strategy, which relies on allocating the entire available GPU memory is quite restrictive for



(a) *Update rate for cuSTINGER [110].* (b) *Update rate for Hornet.* (c) *Execution time of Hornet and cuSTINGER [110].*

FIG. 18.7: *Analysis of update rate of Hornet against cuSTINGER. Hornet is configured in an equivalent manner to cuSTINGER (minimum edges per block = 8, and block array size =* 2^{21}*).*



(a) *Update rate for AIM [278].* (b) *Update rate for Hornet.* (c) *Execution time of Hornet and AIM [278].*

FIG. 18.8: *Analysis of update rate of Hornet against AIM. Hornet is configured in an equivalent manner to AIM to ensure the same interaction with the memory manager and avoid new memory allocations (minimum edges per block = 256, and block array size =* 2^{22}*).*

almost all applications (e.g., all the applications that require additional storage at runtime).

18.4.3 Update rates

We evaluated the update rates (expressed as updates per second) the dynamic data structure can handle for batches from 1 to 10^7 updates.

Hornet, similar to cuSTINGER, STINGER, and AIM, verifies that all new edges do not exist in the graph before insertion. Other data structures, including EvoGraph and GraphIn, do not perform this in their update phase; as such their update phase is potentially shorter. Secondly, it is worth noting that most of the execution time of the batch update for *Hornet* and cuSTINGER is spent in the memory allocation. This is in contrast to the simpler approaches adopted by AIM, EvoGraph, and GraphIn which use static and predefined allocation methodologies making them less flexible.

Fig. 18.7 shows the insertion update rate for four different graphs for both cuSTINGER (a) and *Hornet* data structure (b). Fig. 18.7 (c) summarizes the speedup of the new data structure compared to the cuSTINGER implementation. For small batches, 10^4 edges and smaller, cuSTINGER outperforms *Hornet* by about $2X$. However, for larger batches the cuSTINGER has a performance dip (up to 61X) due to communication overhead between the CPU and GPU - *Hornet* does not. *Hornet* reduces the communication between the CPU and GPU. This also includes fewer memory transfers when allocating new *blocks* on the device.

To perform a fair comparison of *Hornet* with AIM, we configure *Hornet* to use a minimal block size similar to the one found in AIM. In many cases *Hornet* is faster than AIM. This is even despite the fact AIM pre-allocates a large and fixed amount of memory; where *Hornet* grows based on need. Fig. 18.8 depicts the update rate of AIM (a) and *Hornet* (b), and the speedup of *Hornet* compared to AIM (c). Also in this case, *Hornet* shows lower update rates than AIM for small batches, and in particular for graphs with regular degree distribution (*cage15*). On the other hand, *Hornet* outperforms AIM for larger batches up to 82X thanks to a faster memory manager and an efficient update algorithm (see Sec. 18.2.5). Note, for the *kron_g500-logn21* graph, *Hornet* is especially faster than AIM as *Hornet* stores the entire adjacency array in a single block vs. the multiple blocks used by AIM. This improves locality for *Hornet*.

The reduced performance of *Hornet* for small batches is in part due to a preprocessing phase that converts the batch to a CSR representation. This is applied to all batch sizes. While this conversion is relatively costly for small batches, it greatly improves the performance for large batches.

Using the AIM configuration in *Hornet*, *Hornet* can process up to 800 millions updates per second (Fig. 18.8(b)). This can be further increased to 1 billion updates per second if the duplicate testing is disable as was done in GraphIn and Evograph.

18.4.4 Dynamic Triangle Counting

18.4.5 Breadth-first search

BFS is a fundamental graph operation and building block for most graph algorithms. We compare the performance of *Hornet* and CSR data structures for the BFS graph traversal. In addition, we evaluate the proposed solution with the state-of-the-art CSR implementation provided by the Gunrock library [275] to better underlines the efficiency of the approach. As for SpMV, the CSR and *Hornet* implementations are identical except for the data structures used. Fig. 18.9 shows the speedup and the performance (millions traversed edges per seconds, MTEPS) of CSR and *Hornet* in comparison to the Gunrock implementation[4]. The proposed solutions provide performance always higher and up to 5.5X in comparison to Gunrock. In contrast to SpMV, *Hornet* shows slightly higher performance than CSR (up to 10%) thanks to a better locality of vertices with the similar degree within the same block array.

18.4.6 SpMV

Sparse matrix-vector multiplication (SpMV) is a core primitive in linear algebra and widely used in numerous real-world applications. We evaluate the perfor-

[4] To perform a fair comparison, we evaluate all implementations by using the same source vertex and by forcing traversing exactly the same number of edges (`atomicCAS`/no idempotent status lookup).

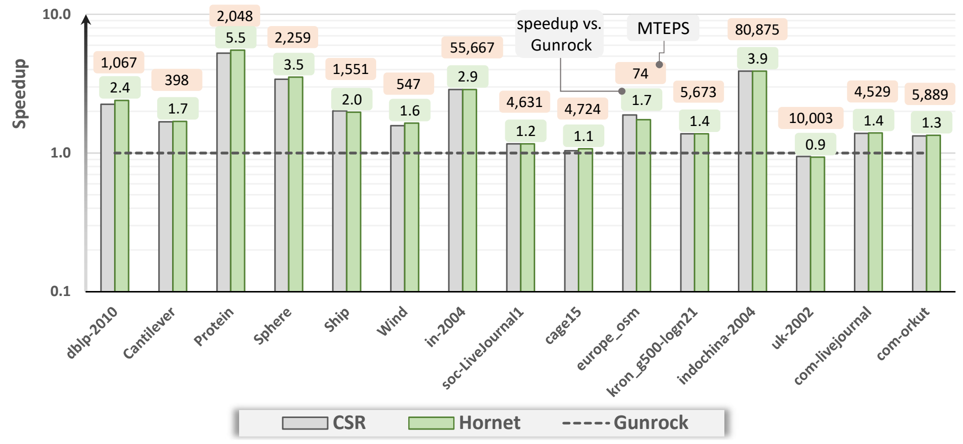


FIG. 18.9: *Performance comparison of BFS between CSR, Hornet, and Gunrock (CSR). The figure depicts the speedup over Gunrock and traversal throughput (MTEPS).*

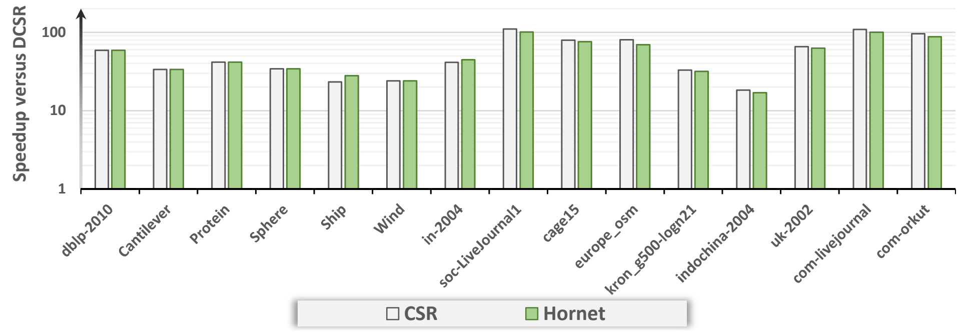


FIG. 18.10: *Performance comparison of SpMV between CSR, DCSR, and Hornet. The figure depicts the normalized speedup over DCSR.*

mance of *Hornet* for SpMV in contrast to CSR and DCSR implementations. Fig. 18.10 compares the performance of SpMV for these three data representations. For DCSR, we use the implementation provided in [144]. The CSR and *Hornet* SpMV implementations are identical except for the data structures used. We are aware of additional SpMV implementations such as yaSpMV [285] (COO based), pOSKI [132] (multi-level CSR), and low-level optimizations (such as those found in CUB, CUSP, and cuSparse). These optimization are orthogonal to the proposed representation. The goal is to show that replacing CSR with *Hornet* does not change the performance of a SpMV.

Specifically, Fig. 18.10 depicts the speedup of the CSR and *Hornet* in comparison to DCSR. First of all, we note that *Hornet* is at least $10X$ faster than DCSR and in some cases as much as $100X$ faster. Second, the difference in execution time between static CSR and *Hornet* is relatively small - less than 10% in most cases. Note, that there cases where *Hornet* is faster than CSR, this has to do with the data placement in the locality of the SpMV computation.

18.5 Conclusions

In this work, we presented a new dynamic data structure, called Hornet, for representing sparse dynamic graphs and matrix problems. The proposed data structure supports both insertion, deletions, and value updates. Unlike past attempts at designing dynamic graph data structures, the new solution does not require restarting due to a large number of edge updates. In Sections 18.3 and 18.4 we discussed a GPU implementation of this data structure. Thought, we note that the data structure is not limited to only GPU architectures.

Hornet, unlike cuSTINGER, has a memory manager that supports memory reclamation; thus reducing memory fragmentation. This is in part due to the block and block array concepts which store adjacencies in arrays of predetermined sizes. By having a relatively small number of array sizes, blocks that were previously freed can be reused in the future. We showed that the new data structure can deal with updates at $4X - 40X$ a higher update rate than cuSTINGER. Further, this new design enables to put an upper bound on the amount of allocated memory. In most cases, the storage overhead of the new data structure is only about 30% higher than the storage requirements of CSR and is usually better than the storage requirements for COO.

Lastly, to show the efficacy of the data structure, we showed that it performs well on an important building blocks for linear algebra and graph processing: SpMV and BFS. By simply replacing CSR in an SpMV implementation, without any additional code tuning, the new data structure is able to perform at the same efficiency of CSR. In most cases, both CSR and cuSTINGER offer an SpMV implementation that is orders of magnitude faster than DCSR.

The experimental results in this Section primarily focused on the analysis of a GPU based implementation of this data structure. In future work, we will extend this data structure to work for CPU systems as well, and we will implement all the necessary functionality to best utilize massively multi-threaded systems.

19

Quickly Finding a Truss in a Haystack

The k-truss of a graph is a subgraph such that each edge is tightly connected to the remaining elements in the k-truss. The k-truss of a graph can also represent an important community in the graph. Finding the k-truss of a graph can be done in a polynomial amount of time, in contrast finding other subgraphs such as cliques. While there are numerous formulations and algorithms for finding the maximal k-truss of a graph, many of these tend to be computationally expensive and do not scale well. Many algorithms are iterative and use static graph triangle counting in each iteration of the graph. In this work we present a novel algorithm for finding both the k-truss of the graph (for a given k), as well as the maximal k-truss using a dynamic graph formulation. The proposed algorithm has two main benefits. 1) Unlike many algorithms that rerun the static graph triangle counting after the removal of non-conforming edges, we use a new dynamic graph formulation that only requires updating the edges affected by the removal. As the updates are local, we only do a fraction of the work compared to the other algorithms. 2) The algorithm is extremely scalable and is able to concurrently detect deleted triangles in contrast to past sequential approaches. While the algorithm is architecture independent, we show a CUDA based implementation for NVIDIA GPUs. In numerous instances, the new algorithm is anywhere from 100X-10000X faster than the Graph Challenge benchmark. Furthermore, the algorithm shows significant speedups, in some cases over 70X, over a recently developed sequential and highly optimized algorithm.

19.1 Introduction

The subgraph isomorphism problem tries to answer the following question, given two graphs H and G (where H is the smaller of these graphs): is there a $1-1$ mapping of vertices in H to vertices in G such that each edge in H is also in G?. For example, H might be a clique of size k, in which case the question is, "Is there a clique of size k in G?". The answer to this question is an $NP-$ *Complete* problem. Yet, there are simplifying assumptions on the structure of H that can help make the problem computationally feasible and tractable - so long as a simpler subgraph H is defined. The need for subgraph isomorphism presents itself in numerous applications, including community detection and social network analysis, were there is a need to find a subgraph with a given set of properties. Another way of looking at the subgraph isomorphism problem is pattern finding, where H represents the pattern. Therefore, it is not overly surprising that in many cases the pattern will be relatively small in comparison with the initial input - this is almost like looking for a *"needle in a haystack"*. For example, a triangle in a graph can be thought of as a pattern and enumerating all the triangles in the graph meets the requirements of the subgraph isomorphism problem. While maximal clique finding is computationally intractable, finding and enumerating all the triangles in a graph can be done in polynomial time. Thus, the problem of subgraph isomorphism can be tractable for specific patterns and for a known H.

The HPEC Graph Challenge [244] seeks to find a high performance solution for a specific subgraph isomorphism problem where the structure of H is a k-truss within G. A k-truss is subgraph where each edge is part of at least $k-2$ triangles. The maximal k-truss in a graph, denoted by $k = k_{max}$ is the largest k-truss in the graph where the set of satisfying edges is not empty. The exact k or structure of the final maximal k-truss is not known apriori and is dependent on the graph. Finding the maximal k-truss can be done in polynomial time [72, 104, 130, 272].

Contribution

In this Section we show a new algorithm for finding a k-truss subgraph as well as the maximal k-truss in a graph. While the algorithm focuses on finding trusses in a static graph, we introduce concepts and principles used in dynamic graph algorithms. First of all we use a dynamic graph data structure designed for sparse networks where the edges can be removed efficiently from the graph without needing to rebuild the graph after each change [110]. Second, we show a highly efficient and scalable dynamic graph triangle counting algorithm for updating the number of triangles in the graphs without needing to recompute all the triangles in every iteration. While the algorithm is architecture independent, we show an implementation of it for the NVIDIA GPU.

Altogether, the new algorithm is significantly faster than the HPEC Graph Challenge [244] benchmarks. While the algorithm always completed in a reasonable amount of time, there are numerous instances in which one or more of the benchmarks did not complete. In most cases we saw that the new algorithm is easily $100X$ faster than the best performing Graph Challenge benchmark and upto $10000X$ faster than the other remaining benchmarks. The new algorithm also outperformed the recent work of Wang & Cheng [272]. While the algorithm in [272] is highly efficient, it is also inherently sequential as it is unable to update the triangle count concurrently when removing multiple edges. The algorithm is concurrent and extremely scalable.

19.2 KTruss Algorithm Using Dynamic Graphs

In this section we present a new algorithm for finding the maximal k-truss (or a specific k-truss) in a graph. The algorithm in [72] suggests recomputing the triangles in every iteration - this is computationally expensive. The algorithm in [272] avoids recomputing triangles for effect edges, yet is sequential. The new algorithm is both scalable and avoids unnecessary computations. The new algorithm extends the algorithm from [175] and updates the number of triangles per edge rather than per vertex.

Both the algorithms in [72] and [272] require removing edges from the graph once the edges no longer support the necessary number of triangles. This edge deletion process is exactly where the algorithm in [175] excels by avoiding unnecessary computations. Part of the edge deletion process also includes removing the edge from the graph. For sparse graphs, this has proven to be challenging, yet several recent data structures have been created that take care of the graph update at high rates, these include STINGER [95] and cuSTINGER [110] for the GPUs. We use cuSTINGER as it supports sorted updates [175] and its data layout is great for both static graph and dynamic graph triangle counting.

19.2.1 Problem Definition

Given a graph, $G = (V, E)$, the vertices are denoted as V and the edges are denoted by E. The maximal k-truss of the graph, $H = (\hat{V}, \hat{E})$ meets the following criteria: 1) $\hat{V} \subseteq V$, 2) $\hat{E} \subseteq E$, 3) $H \subseteq G$, and 4) $\forall e \in \hat{E}, tri(e) >= k - 2$. For the maximal k-truss problem, k needs to be the maximal value before $E = \varnothing$ and $V = \varnothing$. In many papers, the term *support of edge* can be used to replace the term $tri(e)$. We use both throughout this Section.

19.3 Proposed Algorithm

Algorithm 16 presents the pseudo code for the new algorithm. While the various functions in the algorithm do not highlight the parallelism in the algorithm, the function calls are all inherently parallel. For example, finding all the vertices with a support smaller than $k - 2$ can be done by accessing all the edges in the graph concurrently. Deleting the edges that lack support can also be done in parallel. Lastly, updating the triangle counting of the edges can also be done in parallel.

19.3.1 Triangle Subtraction

Consider a triangle in a graph consisting of three vertices u, v, w. The different and **ordered** triangles consisting of vertices are : $\langle u, v, w\rangle$, $\langle u, w, v\rangle$, $\langle v, w, u\rangle$, $\langle v, u, w\rangle$, $\langle w, u, v\rangle$, and $\langle w, v, u\rangle$. As the graph is undirected, there is a certain amount of symmetry: $sup(u, v) = sup(v, u)$. This also means that if (u, v) is deleted, then (v, u) is also deleted. We denote a set of two edges (u, v) and (v, u) as an **edge-pair**.

Thus, given a triangle in the graph prior to the removal of a subset of edge-pairs, the following scenarios can arise from the removal: 1) a single edge-pair is removed, 2) two edge-pairs are removed, and 3) all three edge pairs are removed. If a single edge-pair is removed, then the remaining two edge-pairs need their support to be modified. If two edge-pairs are removed, then the remaining edge-pair needs to be updated. When all three edge-pairs are deleted, then no modifications are

Algorithm 16 New algorithm for finding k-truss

```
    Input: $G = (V, E), K$

    UpdateTriangle($\hat{G}$, $E_{rem}$)
 1: Construct $G_{rem} = (V_{rem}, E_{rem})$
 2:         → Update triangles where 1 edge-pair is deleted
 3: parallel for $\langle u, v\rangle \in E_{rem}$ do
 4:     $Intersect(\hat{G}_u, \hat{G}_v)$
 5: end parallel for
 6:         → Update triangles where 2 edge-pairs is deleted
 7: parallel for $\langle u, v\rangle \in E_{rem}$ do
 8:     $Intersect(\hat{G}_u, G_{rem,v})$
 9: end parallel for

    OneK($G, K$)
10:         → Par-for on all edges in graph looking for $sup(e) < k - 2$
11: $E_{rem} \leftarrow FindUnderKm2(G, K)$
12: while ($|E_{rem}| \neq 0$) do
13:     $Remove(G, E_{rem})$
14:     if ($G = \varnothing$) then
15:         return
16:
17:     $UpdateTriangle(G, E_{rem})$
18:             → Par-for on all edges in graph looking for $sup(e) < k - 2$
19:     $E_{rem} \leftarrow FindUnderKm2(G, K)$
20: end
21: return

    NewKTruss($G, K$)
22: while $True$ do
23:     $OneK(G, K)$
24:     if ($G = \varnothing$) then
25:         return $k - 1$;
26:
27:     $k \leftarrow k + 1$
28: end
```

Table 19.1: GPU and CPU system used in experiments.

Architecture	Processor	Micro-architecture	SM	SP (per SM)	Total SPs	DRAM Size	DRAM Type
GPU-CUDA	P100	Pascall	56	64	3854	16GB	HBM2

Architecture	Micro-architecture	Processor	Frequency	Cores	LL-Cache	DRAM Size	DRAM Type
CPU x86-64	Broadwell	2× Intel Xeon E5-2695 v4	2.1 GHz	2× 16	2× 45 MB	1024GB	DDR4-2400

required as all the edges are no longer in the graph. Note, that 1) a single deleted edge-pair can affect multiple triangles and 2) these three scenarios capture all the possible changes caused by a deletion of a given edge-pair.

19.3.2 Triangle Detection For Single Edge-Pair Deletions

Assuming that the deleted edge-pair consists of vertices u and v, we are required to find all the affected triangles. This requires intersecting the adjacency arrays of u and v in $\hat{G}$ (where $\hat{G} = (\hat{V}, \hat{E})$) is the graph after the removal of the edges). By intersecting (u, v) and (v, u), the common neighbors are found. For each of these common neighbors a triangle is decremented from its edge count. This is the simpler of the two cases.

TABLE 19.2: Networks used in the experiments. $|E|$ refers to directed edges. Networks are sorted based on the number of EDGES. Execution time is for cuSTINGER-Delta.

Name	$\|V\|$	$\|E\|$	k_{max}	$Time(s)$
p2p-Gnutella08	6.3K	21K	5	0.007
ca-HepTh	9.8K	26K	32	0.005
ca-HepPh	12K	119K	239	0.009
email-Enron	37K	184K	22	0.026
soc-Epinions1	76K	406K	33	0.09
cit-HepPh	35K	421K	25	0.24
soc-Slashdot0902	82K	504K	36	0.085
roadNet-PA	1M	1.5M	4	0.078
flickrEdges	106K	2.3M	574	0.26
amazon0601	400K	2.4M	11	0.12
graph500-scale18	262K	4.2M	159	0.74
graph500-scale19	524K	8.4M	213	6.8
graph500-scale20	640K	16M	284	17.3
cit-Patents	3.8M	16.5M	36	45.3
graph500-scale21	2.1M	34M	373	117
graph500-scale22	4.2M	67M	485	291
graph500-scale23	8.4M	134M	625	780
Name (Wang & Chang [272])	$\|V\|$	$\|E\|$	k_{max}	$Time(s)$
wiki-Talk	2.4M	4.7M	53	9.07
as-skitter	1.7M	11M	68	57.1
soc-LiveJournal1	4.8M	43M	362	258

19.3.3 Triangle Detection For Dual Edge-Pair Deletions

The process for detecting and updating the number of triangles when deleting two edge pairs is a bit more complex (the reader is referred to [175] for additional details) and we provide only a sketch of the process. For simplicity, assume that these edges are $(u, v), (v, u), (u, w), (w, u)$. Thus, we are required to update the edge pair (v, w) and (w, v). Given the set of deleted edges E_{rem}, a graph of the deleted edges is created, we call this graph $G_{rem} = (V_{rem}, E_{rem})$. Given G_{rem} and $\hat{G}$, to find the common neighbors (v, w) and (w, v), we do the following intersections between the vertex pairs: $\langle u_{rem}, \hat{v}\rangle$, $\langle v_{rem}, \hat{u}\rangle$, $\langle u_{rem}, \hat{w}\rangle$, $\langle w_{rem}, \hat{u}\rangle$ where the adjacencies arrays of $\hat{v}, \hat{u}, \hat{v}$ are in $\hat{G}$ and the adjacencies arrays of $u_{rem}, v_{rem}, w_{rem}$ are in G_{rem}. Note, all four of these intersections are required due to the asymmetry of the adjacency arrays in the two different graphs.

19.4 Experimental Results

The experiments are conducted on an NVIDIA P100 GPU connected to an Intel Xeon E5-2695 with 32 cores (details in Table 19.1. The P100 is a Pascal based GPU with 56 SMs and 64 SPs per SM, for a total of 3584 SPs. The P100 has a total of 16GB of HBM2 memory. The Intel Xeon E5-2695 is a Broadwell based processor running at 2.1 GHz with 45MB L3 cache. The server consists of two such processors with a total of 1TB of memory. While the new algorithm is architecture independent, the final implementation targets the GPU. Thus, while the GPU is connected to a high-end Intel processor, in practice we only utilize a single CPU thread.

19.4.1 Dynamic Graphs

The *cuSTINGER* data structure is the first fully dynamic graph data structure for the GPU [110]. *cuSTINGER* uses dynamically growing arrays. This allows for improved locality and increased parallel scalability for the GPU's warp based execution model. Specifically, the use of arrays *cuSTINGER* allows for inserting and deleting edges from the graph while ensuring that the edge lists are sorted after the update with relatively low computational effort [175].

19.4.2 Benchmarks

We compare the performance of the proposed algorithms with several different implementations, including the baseline benchmarks defined by the HPEC Graph Challenge [244]. The baseline benchmarks are formulated in a linear algebra based formulation in several different programming languages: Matlab/Octave, Julia, and Python. While these programming languages are high-level, they utilize several optimized libraries for the sparse matrix representation as well as for the SpMV operations. We also compare the new algorithm to Graphulo [104] and Wang and Chen [272]. Of these, only [272] and the proposed algorithm use a non linear algebra formulation.

We evaluate the benchmarks on two related but distinct challenges: 1) finding k-truss for a specific value of k and 2) finding all k-trusses up to and including the maximal. We treat these as distinct because there are algorithmic optimizations that are available to the former that aren't to the latter, and vice versa. The results of Wang and Cheng [272] are only for the latter problem. On the other hand, the Graph Challenge benchmarks find trusses of a single k by default. We extended these implementations to iteratively find the maximal k-truss, as suggested in [104].

Proposed implementations - in the performance analysis we compare two different implementations: 1) *cuSTINGER-Iterative* - a näive algorithm that enumerates the triangles for all the edges in the graph for each iteration of the algorithm using a static graph formulation and 2) *cuSTINGER-Delta* - an implementation of the new algorithm discussed in 19.2. While both these algorithms utilized the cuSTINGER data structure for deleting edges not meeting the support requirements of the k-truss, only cuSTINGER-Delta utilizes the smart update process.

Python - we found the Python implementation to typically be the most stable of the Graph Challenge benchmarks. Whereas the other benchmarks did not always complete, the Python benchmark always did. The Python implementation utilizes SciPy [136] library for its sparse operations. This benchmark is sequential.

Matlab/Octave - this sequential benchmark supports both Matlab and Octave syntax. We used the Octave framework for the execution. We ran into memory-related errors (exceeding memory, seg faults) on some inputs.

TABLE 19.3: Execution time comparison for finding the maximal k-truss with those found in [272] and $k = 3$ with those found in [131].

	P2P	HEP	Amazon	Wiki	Skitter	LJ
Time (Wang & Cheng [272])	< 1	< 1	31	121	281	664
Time (cuSTINGER-Delta)	0.014	0.038	0.43	9.07	57.1	258
Speedup	< 70	< 26	72	13	5	2.57

	S10	S11	S12	S13	S14	S15	S16
Time (Graphulo [131]	1.63	3.93	12.1	37.2	110	3290	8770
Time (cuSTINGER-Delta)	0.003	0.007	0.016	0.042	0.106	0.352	1.18
Speedup	518	595	741	883	1041	9330	7847

Julia - we found that the sequential Julia implementation had several problems, including memory leakage and bad parsing of the input files. Further, there were several cases that the execution was so slow that the benchmarks were stopped.

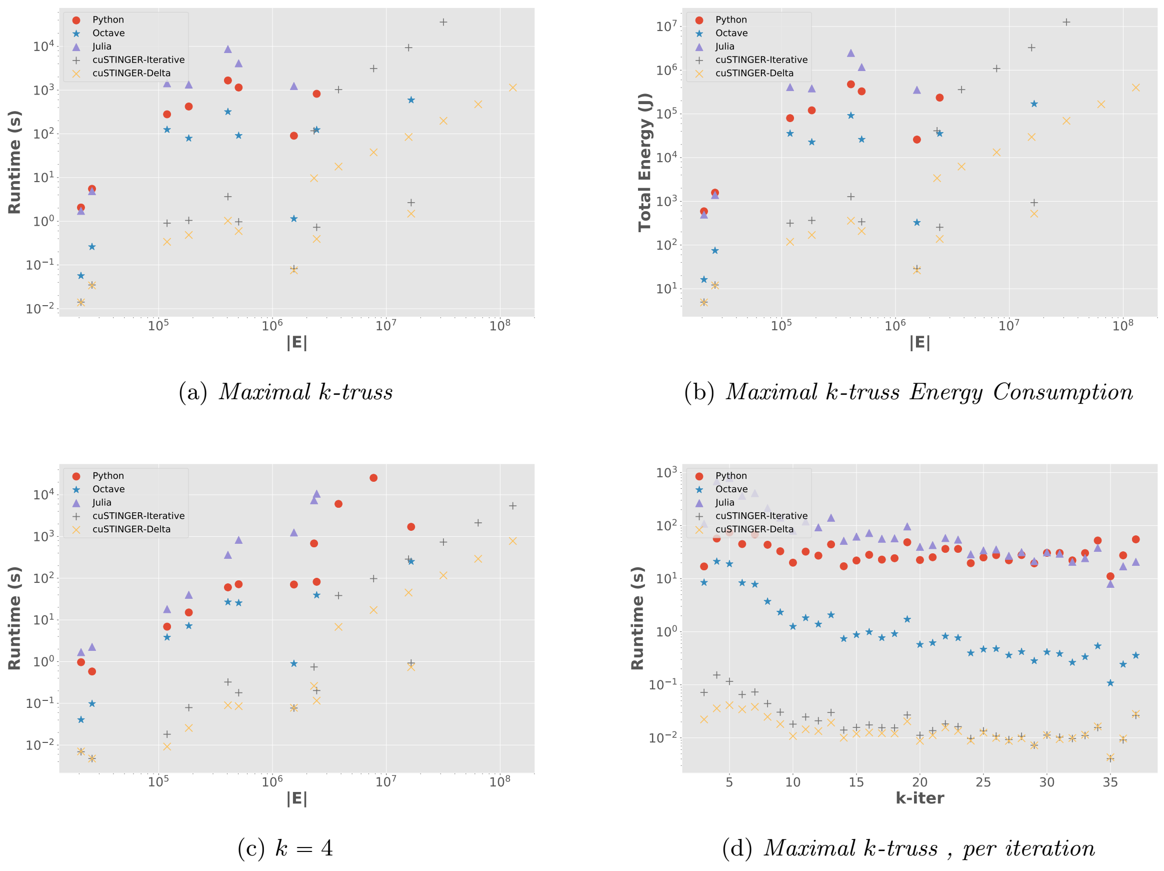
Wang and Cheng [272] - this benchmark is highly optimized, yet sequential algorithm for finding the k-truss. The code for this algorithm is not open-source, as such we compare the performance of the algorithm directly to the numbers reported in their paper. Details of the system used for these experiments can be found in [272].

Graphulo [104] - while the Graphulo framework has an open-source k-truss implementation, we were unable to collect execution times due to errors. As such, we use the execution times reported in [131], and ran the same inputs on cuSTINGER for comparison. This benchmark is parallel - details of the system used for these experiments can be found in [131].

19.4.3 Dataset

The HPEC Graph Challenge [244] has a pre-determined set of networks that are to be used for evaluating the performance of the new algorithm. This consists of graphs from the SNAP dataset [161] and synthetic graphs which are also used for the Graph500 benchmark. Details of the Graph Challenge Graphs can be found in the upper part of Table 19.2. For the sake of brevity we do not present all the graphs in the Graph Challenge list, rather we highlight only a subset of them. We used adjacency files as provided by the Graph Challenge dataset for cuSTINGER, and convert them into their incidence forms for linear-algebra-based benchmarks. The graphs used in [272] also consist of SNAP graphs and can be found in the bottom part Table 19.2, though they are not in the original Graph Challenge List. As such, we preprocessed these graph from their original SNAP [161] format to the one required by the benchmarks. For comparison with the results of Graphulo [131], we generated scale-free graphs (scales 10-16) using their generator script and seed. We then converted these graphs to be run on cuSTINGER.

Fig. 19.1 (a) and (c) depict the execution of the various algorithms for finding the maximal k-truss and for finding the k-truss for $k = 4$, respectively. The abscissa denotes the number of edges in the input graph and the ordinate depicts the



(a) *Maximal k-truss* (b) *Maximal k-truss Energy Consumption* (c) $k = 4$ (d) *Maximal k-truss , per iteration*

FIG. 19.1: *(a),(b), (c): execution times and energy consumption for finding the trusses as a function of the graph size (number of edges in the input graph): (a) maximal k-truss of the graph, (b) the energy consumption for find the maximal k-truss , and (c) for trusses of $k = 4$. These are log-log plots. (d) Execution time of the various algorithms for finding the maximal k-truss ,for the soc-Slashdot, as a function of the iteration (k). Note, that while $k_{max} = 36$, the algorithms terminate for $k = 37$ when the graph becomes empty. The Y-axis is log-scale.*

execution time. Note, both the abscissa and the ordinate are log based. Missing data points imply either the benchmark did not finish in a reasonable amount of time (the upper-bound on execution time was approximately 8 hours) or the benchmark did not complete for some reason (exceed memory, crashed in the graph loading phase). The motivation for separately getting execution times for the maximal k-truss versus finding trusses of a predefined k stems from the way that edges are removed from the graph. Consider the case where k is selected to be k_{max}. Even though the output of both these test cases will be the same, the execution time for $k = k_{max}$ will be faster than the execution time when needing to look for k_{max} due to more aggressive edge filtering.

For both these approaches, it can be distinguished from Fig. 19.1 that both the new implementations, cuSTINGER-Iterative and cuSTINGER-Delta, significantly outperform the remaining benchmarks by orders of magnitude (from $100X$ and upto $10000X$). While it is not fair to compare a GPU implementation with a sequential implementation, we note that the speedup of the proposed algorithms is not just from the use of a NVIDIA GPU, but is due to several additional factors: 1) problem formulation, 2) algorithmic optimizations, and 3) data structure support. It is also worth noting the work of Lee *et al.* [155] compares the performance of CPUs to GPUs and narrowed down the relative speedup of a GPU over a CPU to a significantly smaller value than the achieved speedups of the proposed implementation. **Problem formulation** - based on the findings of Schank & Wagner [258], it has been established that the time complexity for the linear algebra formulation is higher than the vertex-centric formulation (which we use). **Algorithmic optimizations** - the new algorithm has several important algorithmic optimizations that reduce the total amount of work required for re-enumerating the number of triangles per edge. **Data structure support** - given the usage of cuSTINGER [110] and its support of dynamic graphs, we don't need to recreate the sparse graph after each iteration of the graph. This saves a lot of time on memory allocations.

While we are unable to measure the magnitude of each of the aforementioned optimizations and their contribution to the overall speedup, we do know that the new algorithm, cuSTINGER-Delta, is several times faster than cuSTINGER-Iterative (we saw an up to 30X difference between the two algorithms). This is directly attributed to the algorithmic optimizations. Also, it is worth noting that due to these optimizations, the implementations scale to significantly larger graphs (well over 100M edges) compared to the remaining benchmarks.

Fig. 19.1 (b) depicts the projected energy (J) consumption for maximal k-truss run-times and power measurements while running benchmarks on the CPU and GPU respectively. We used the *ipmitool* tool for measuring power. The power measured for the CPU is ($286\frac{J}{s}$). The power measured for the GPU also includes the CPU power and is only slightly higher at ($350\frac{J}{s}$). Based on measurement, the GPU is not using its peak power. This probably has to do with the fact that the GPU is executing relatively small kernels and for a short period of time. However, the power-performance plot of Fig. 19.1 (b) does not look significantly different than Fig. 19.1 (a) - this is due to the new algorithm being extremely power efficient due to its short execution time.

Fig. 19.1 (d). depicts the runtime of the various implementations as a function of the k, in the process of searching for the maximal k-truss . The *soc-Slashdot0902* graph was selected as it successfully completed on all benchmarks. As expected, generally the time per iteration decays over time as the number of active edges (edges under consideration) tends to monotonically decrease with the increase of k [1]. The only exception is the Python benchmark, where the time per iteration remains in a constant range for this input. The new algorithms are orders of magnitude faster than the Graph Challenge Benchmark [244].

[1] Note, the execution time for a given k is dependent on the number of sub-iterations for that given k - this can also explain the increase in execution time from $k = 3$ to $k = 4$

Comparison with Wang and Cheng [272] Table 19.3 (upper) compares the new algorithms with the algorithm by Wang & Cheng [272]. We picked six inputs of varying sizes from their list of tested graphs (all from the SNAP graph repository). While [272] had numerous implementations, we compare against their bottom-up approach as it is the most similar to the proposed algorithm. We show speedups of the proposed algorithm over theirs. In all cases the cuSTINGER-Delta implementation outperforms [272].

Comparison with Graphulo [131] Lastly, we compare the new algorithm, Table 19.3 (lower), with the Graphulo framework which has a linear algebra based implementation for finding a k-truss of a given size, using inputs from their paper. The Graphulo framework is intended to process larger graphs. Similar to the other linear algebra based formulations, the new algorithm is orders of magnitude faster.

19.5 Conclusions

In this Section we showed a new algorithm for finding k-truss subgraphs. The new algorithm uses a dynamic graph formulation and exploits two important features: 1) it utilizes a dynamic graph data structure that can insert and remove edges without creating a new data structure after each update and 2) it avoids recomputing the number of triangles per edge in each iteration of the algorithm after the edge removals. The latter of these properties means that the new algorithm does a fraction of the work that static graph algorithms do - this leads to significant speedups. In addition to this, the new algorithm is also extremely scalable and can concurrently detect when a deleted edge is part of multiple triangles and it can update all the affected edges (in parallel).

While the algorithm is architecture independent, the CUDA based implementation showed massive speedups over the Graph Challenge benchmarks. There were numerous instances where the Graph Challenge benchmarks did not complete in reasonable amount of time (8 hours) whereas the proposed algorithm finished in a few minutes. The new algorithm was often over a hundred times faster than the best performing Graph Challenge Benchmarks and thousands of times faster than the remaining benchmarks. The proposed algorithm also scaled to much larger graphs than in the benchmarks. While part of speedup can be attributed to the usage of an NVIDIA GPU, the bigger part of the speedup is due to the new algorithmic optimizations we showed. Further, the new algorithm is in some case over $70X$ faster than a recently developed and optimized algorithm that is inherently sequential.

Part V

Applications

Introduction

In this part of the thesis, many concepts and methodologies deriving from previous works concerning GPU applications have been applied to an important problem arising from an external context as well as several graph analytic results have been combined to accelerate environments not strictly related to high-performance computing. This part first presents an efficient dynamic invariant miner to accelerate extraction of logic formulas from an execution trace of a system under verification (Section 20). Such formulas express stable conditions in the behavior of the system and they are selected among a very large set of runtime states. Secondly, the part introduces *cuRnet*, an R package that provides a parallel implementation of the most important graph algorithms in the literature (Section 21). The package allows offloading computing intensive applications to GPU device and allows to perform such analysis which would be highly time-consuming on a standard sequential computation.

20

Invariant Mining

Dynamic mining of invariants is a class of approaches to extract logic formulas from the execution traces of a system under verification (SUV), with the purpose of expressing stable conditions in the behaviour of the SUV. The mined formulas represent likely invariants for the SUV, which certainly hold on the considered traces, but there is no guarantee that they are true in general. A large set of representative execution traces must be analysed to increase the probability that mined invariants are generally true. However, this becomes extremely time-consuming for current sequential approaches when long execution traces and large set of SUV variables are considered. To overcome this limitation, the Section presents a parallel approach for invariant mining that exploits GPU architectures for processing an execution trace composed of millions of clock cycles in few seconds.

20.1 Introduction

Invariant mining is a technique to extract logic formulas that hold between a couple (or several couples) of points in an implementation. Such formulas express stable conditions in the behaviour of the system under verification (SUV) for all its executions, which can be used to analyse several aspects in verification of SW programs and HW designs, at different abstraction levels. For example, invariant mining has been applied for analysis of dynamic memory consumption [53], static checking [200], detection of race conditions [234], identification of memory access violations [123], test generation [77], mining of temporal assertions [79] and bug catching in general [253].

Both static and dynamic approaches exist for mining invariants. The first exhaustively and formally explore the state space of the SUV [100, 265], but they work well for relatively small/medium size implementations. Moreover, they require the source code of the SUV is available. When larger designs are considered, dynamic techniques represent a not exhaustive but more scalable solution, since they rely on simulation rather than formal methods [97, 116, 178, 253]. Moreover, these approaches are the unique alternative when the source code of the SUV is non available. In fact, they generally work by analysing a set of execution traces of the SUV and searching for counterexamples of the logic formulas that represent the desired invariant candidates. However, at the end of the analysis, survived candidates are *likely invariants*, i.e., formulas that are only statistically true on the SUV, because they have been proved to hold only on the analysed traces. For this reason, to increase the degree of confidence on likely invariants, a large and representative set of execution traces must be analysed by dynamic approaches. Unfortunately, for complex HW designs this could require to elaborate thousands of execution traces, including millions of clock cycles, and predicating over hundreds of variables, which becomes an unmanageable time-consuming activity for existing approaches.

The solution we propose to speed-up the mining process is to move from a sequential to a parallel implementation of likely invariant miners, such that general-purpose computing on graphics processing units (GPGPU) can be exploited to significantly reduce the time required for processing a large number of execution traces composed of millions of clock cycles. A first parallel approach for invariant mining has been presented in [80] showing sensible improvements with respect to Daikon [97], one of the most popular sequential miners. In this Section, we propose an alternative parallel algorithm that greatly benefits from advanced graphics processing unit (GPU) programming techniques, such that the memory throughput of the GPU is significantly improved. In this way, as reported in the experimental results, the overall performance of the mining algorithm are increased up to three orders of magnitude with respect to [80].

The rest of the work is organized as follows. Section 20.2 defines some preliminary concepts. Section 20.3 describes the proposed parallel approach for dynamic invariant mining. Finally, Section 20.4 and Section 20.5 are devoted, respectively, to experimental results and concluding remarks.

20.2 Preliminary definitions

The following definitions concerning *execution traces* and *likely invariants* are necessary to describe how the mining approach presented in Section 20.3 works.

Definition 1. Given a finite sequence of simulation instants $\langle t_1, ...t_n \rangle$ and a set of variables $\mathcal{V}$ of a model $\mathcal{M}$, an *execution trace* of $\mathcal{M}$ is a finite sequence of pairs $\tau = \langle (\mathcal{V}_1, t_1), ...(\mathcal{V}_n, t_n) \rangle$, where $\mathcal{V}_i = eval(\mathcal{V}, t_i)$ is the value of variables in $\mathcal{V}$ at simulation instant t_i.

Definition 2. Given a model $\mathcal{M}$ and the corresponding sets of variables $\mathcal{V}$ and execution traces $\mathcal{T}$, a *likely invariant* for $\mathcal{M}$ is a logic formula over $\mathcal{V}$ that holds throughout each $\tau \in \mathcal{T}$.

20.3 Invariant mining

The main mining function, in its sequential form, is reported in Algorithm 17. The inputs of the function are represented by an execution trace τ of the SUV, an invariant template set $\mathcal{I}$, and a variable dictionary $\mathcal{D}$. The dictionary contains tuples of different arity composed by all the possible combinations of the variables $\mathcal{V}$ of the SUV. Such tuples represent the actual parameters to be substituted inside the formal parameters of the invariant templates during the mining phase.

The algorithm extracts all likely invariants for τ that correspond to logic formulas included in $\mathcal{I}$, by substituting in the elements of $\mathcal{I}$ all the possible tuples of $\mathcal{V}$ belonging to $\mathcal{D}$, according to the respective arity. More precisely, the *check_invariant* function (line 5) checks if a specific template INV, instantiated with the current tuple of variables TUPLE, holds at simulation time INSTANT. When a counterexample is found for INV, it is removed from the template set (line 6) for the current tuple of variables. If all elements of the template set are falsified (line 8), the algorithm restarts by considering the next tuple in the dictionary, by skipping the remaining simulation instants of τ. At the end, the algorithm collects all the pairs composed by the the survived templates and the corresponding tuples of the variable dictionary (line 11). The instantiation of the tuples in the survived templates represent the final set of likely invariants for τ. The current implementation supports the invariant template sets reported in Table 20.1.

The proposed algorithm has a worst-case time complexity equal to $\mathcal{O}(|\mathcal{V}|^K \cdot |\tau| \cdot |\mathcal{I}|)$, where $\mathcal{V}$ is the number of considered variables, K is the arity of the invariant template belonging to $\mathcal{I}$ with the highest arity, $|\tau|$ is the number of simulation instants in the execution trace τ, and $|\mathcal{I}|$ is the number of invariant templates included in $\mathcal{I}$.

20.3.1 The parallel implementation for GPUs

The mining approach reported in Algorithm 17 is well suited for parallel computation. In fact, the problem can be easily decomposed in many independent tasks, each one having regular structure and fairly balanced workload. For this reason we implemented a parallel version of the mining algorithm, called *Mangrove*. It implements the mining algorithm with the aim of exploiting the massive parallelism of GPUs and, at the same time, an inference strategy to reduce redundant checking of invariants, as explained in Section 20.3.2.

Algorithm 17 The invariant mining algorithm.

```
    sequential_mining($\mathcal{D}$, $\mathcal{I}$, $\tau$) return result
 1: for all TUPLE $\in \mathcal{D}$ do
 2:     template_set = $\mathcal{I}$
 3:     for all INSTANT $\in \tau$ do
 4:         for all INV $\in$ template_set do
 5:             if $\neg$check_invariant(INV,TUPLE,INSTANT) then
 6:                 template_set = template_set $\setminus$ INV
 7:         end
 8:         if template_set = $\emptyset$ then
 9:             break
10:     end
11:     result = result $\cup$ $\langle$TUPLE, template_set$\rangle$
12: end
```

	BOOLEAN			NUMERIC		
	UNARY	BINARY	TERNARY	UNARY	BINARY	TERNARY
TEMPLATE SET I	true, false				$=, \neq, <, >, \leqslant, \geqslant$	
TEMPLATE SET II	true, false	$=, \neq$	$\mathrm{Var}_1 = \mathrm{Var}_2 \,\mathrm{AND}\, \mathrm{Var}_3$ $\mathrm{Var}_1 = \mathrm{Var}_2 \,\mathrm{OR}\, \mathrm{Var}_3$ $\mathrm{Var}_1 = \mathrm{Var}_2 \,\mathrm{XOR}\, \mathrm{Var}_3$	$\mathrm{Var} = 7$ $\mathrm{Var} \neq 0$ $\mathrm{Var} < 10$ $\mathrm{Var} \leqslant 10$	$\mathrm{Var}_1 = \mathrm{Var}_2$ $\mathrm{Var}_1 \leqslant \mathrm{Var}_2$ $\mathrm{Var}_1 < \sqrt{\mathrm{Var}_2}$ $\mathrm{Var}_1 = \log \mathrm{Var}_2$ $\mathrm{Var}_1 < \mathrm{Var}_2 + 1$ $\mathrm{Var}_1 = \mathrm{Var}_2 * 2$	$\mathrm{Var}_1 = \mathrm{Var}_2^{\mathrm{Var}_3}$ $\mathrm{Var}_1 = \min(\mathrm{Var}_2, \mathrm{Var}_3)$ $\mathrm{Var}_1 = \max(\mathrm{Var}_2, \mathrm{Var}_3)$ $\mathrm{Var}_1 < \mathrm{Var}_2 * \mathrm{Var}_3$ $\mathrm{Var}_1 \leqslant \mathrm{Var}_2 + \mathrm{Var}_3$

TABLE 20.1: *Template sets considered by the miner.*

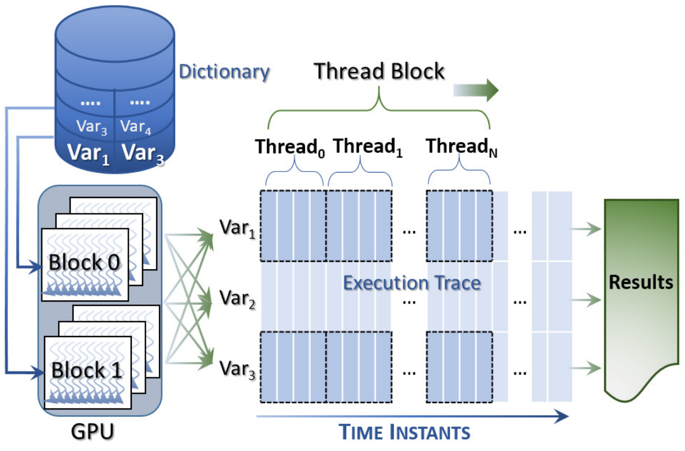


FIG. 20.1: *Overview of block mapping and vectorized accesses for the parallel algorithm on GPU.*

In an initialization phase, the Boolean and numeric variables included in the variable dictionary are organized over bit and float arrays in *row-major order*. This allows the full coalescing of memory accesses by the GPU threads in the mining phase. Furthermore, all accesses are *vectorized* [169], namely, each thread loads four consecutive 32-bit words instead of a single word. This technique allows improving the memory bandwidth between DRAM and thread registers.

Mangrove computes the mining process by elaborating, in sequence, the unary templates, the binary templates, and, finally, the ternary templates reported in Table 20.1. The tool takes advantage of the massive parallelism of GPUs by mapping each thread block on a different entry of the variable dictionary (Fig. 20.1).

In each block, the threads communicate and synchronize through shared memory. As for the standard characteristics of the GPU architectures, such hardware-implemented operations are extremely fast and their overhead is negligible. Communication and synchronization among block threads allow avoiding redundant checking of already falsified invariants and stopping the computation of the whole block as soon as all invariants for a particular set of variables have been falsified.

In the GPU implementation the variable dictionary consists of a simple data structure that stores in each entry a subset of variables involved in a specific template. *Mangrove* initializes the variable dictionary through the host CPU and strongly exploits it in the mining phase through the GPU threads, as detailed in the following sections.

20.3.2 Generation of the variable dictionary

In the generation of the variable dictionary, the goal is to avoid redundant storing and elaboration of variables during the mining phase. Such a redundancy is due to the fact that the GPU threads, during the mining phase, cannot have information about any already discovered invariant among variables in the whole execution trace. Thus, to increase the efficiency of the parallel computation, *Mangrove* implements different optimizations during the generation of the variable dictionary. The idea behind such optimizations consists of avoiding wasting of time to check if an invariant template is satisfied, when the same answer can be inferred from the result of previous mining steps, as explained in the next paragraphs:

- The result of the mining over unary templates is exploited during the mining of binary templates. As a simple example, *Mangrove* searches for any Boolean variable, var_a, whose value is always equal (or always different) to any other Boolean variable, var_b. If such a condition occurs, the generation of the entry $< var_a, var_b >$ in the dictionary can be avoided since it is redundant.
- The result of the mining over unary and binary templates is used during the mining of ternary templates. For example, by considering the ternary mining phase on Boolean variables, the goal is to figure out which operator $op \in \{\text{AND}, \text{OR}, \text{XOR}\}$ can be validated over three different variables (e.g., var_a, var_b, and var_c). Through the already extracted unary and binary invariants, *Mangrove* automatically infers some ternary invariants without applying the checking procedure throughout the execution traces. For instance, the ternary invariant $(var_a = var_b \text{ AND } var_c)$ reduces to check whether the binary invariant $(var_a = var_b)$ occurs when $(var_b = var_c)$ holds. Similarly $(var_a = var_b \text{ XOR } var_c)$ reduces to check $(var_a \neq var_c)$ when var_b is constantly set to *true*.

20.3.3 Data transfer and overlapping of the mining phase

The invariant mining process on the GPU consists of three main steps showed in Fig. 20.2(a): (i) reading of the execution trace from the mass storage (disk) and data storing in the host DRAM memory; (ii) data transfer from the host to the memory of the GPU; (ii) elaboration in the GPU device. The three steps work first on the numeric variables and then they are repeated for the Boolean variables.

Mangrove implements such a process by overlapping the three steps as shown in Fig. 20.2(b). This allows totally hiding the cost of host-device data transfers and partially hiding the cost of the mining elaboration. Moreover, *Mangrove* implements the data transfer overlapping through asynchronous kernel invocations and memory copies (i.e., `cudaMemcpyAsync` in CUDA). Finally, a specific optimization has been implemented for Boolean variables: *Mangrove* stores the values of Boolean

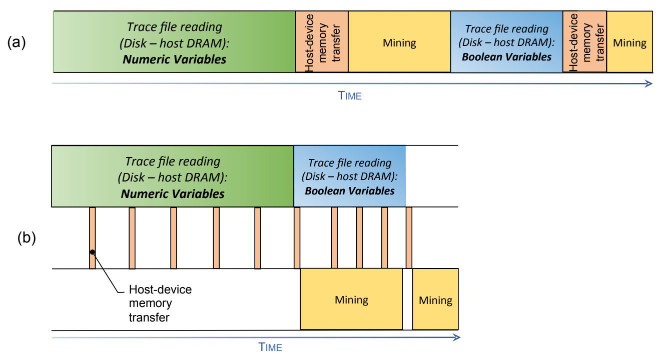

(a)
Trace file reading (Disk – host DRAM): Numeric Variables
Host-device memory transfer
Mining
Trace file reading (Disk – host DRAM): Boolean Variables
Host-device memory transfer
Mining
TIME
(b)
Trace file reading (Disk – host DRAM): Numeric Variables
Trace file reading (Disk – host DRAM): Boolean Variables
Host-device memory transfer
Mining
Mining
TIME

FIG. 20.2: *The invariant mining phases (a), and the overlapped implementation of* Mangrove *for GPUs (b).*

variables in arrays of bits to reduce the memory occupation (e.g., 5,000,000 values of a Boolean variable are stored in 600 KB). In addition, this array-based representation allows using bitwise operations to concurrently elaborate 32 Boolean values in a single chunk, thus speeding up the mining phase.

20.4 Experimental results

Experimental results have been run on a NVIDIA Kepler GeForce GTX 780 device with 5 GHz PCI Express 2.0 x16, CUDA Toolkit 7.0, AMD Phenom II X6 1055T 3GHz host processor, and the Debian 7 Operating System. To evaluate the efficiency of *Mangrove* experiments have been conducted on different kind of execution traces, whose characteristics are summarized in Table 20.2. The considered execution traces differ in terms of number of variables and number of likely invariants that it is possible to extract by considering the template sets reported in Table 20.1. These are the two parameters that most influence, together with the length of the trace, the execution time of the mining algorithm. On the opposite, information about the complexity of the SUV from which execution traces have been generated are irrelevant when the SUV model is not explored. Indeed, higher is the number of likely invariants exposed by the execution traces, higher is the time spent for their extraction, even if the SUV is very simple from the computational point of view.

The efficiency of *Mangrove* has been compared against the sequential mining approaches implemented, respectively, in [97] and in [80], and the parallel implementation proposed in [80]. Table 20.3 shows the execution time required to extract the likely invariants according the first and second template sets on the traces reported in Table 20.1. For the parallel approaches, the times include the overhead introduced for data transfer between host and device. *Mangrove* provides the best results in all datasets by executing up to four orders of magnitude faster than the sequential state-of-the-art tool Daikon[1]. Compared to the more recent

[1] For a fair comparison, Daikon has been configured to search only for the invariants specified in the first and second template sets.

	Length	Boolean Vars	Numeric Vars	Invariants (Temp. set I)	Invariants (Temp. set II)
TRACE 1	5,000,000	15	15	0	0
TRACE 2	5,000,000	15	15	142	964
TRACE 3	5,000,000	50	50	0	0
TRACE 4	5,000,000	50	50	1,788	42,371

Table 20.2: *Characteristics of execution traces.*

		Daikon [97]	Sequential [80]	Parallel [80]	Mangrove
Template Set I	TRACE 1	103 s	< 1 ms	116 ms	< 1 ms
	TRACE 2	170 s	4,629 ms	116 ms	17 ms
	TRACE 3	287 s	2 ms	369 ms	< 1 ms
	TRACE 4	1366 s	52,160 ms	457 ms	182 ms
Template Set II	TRACE 1	2 m 34 s	22 ms	352 ms	< 1 ms
	TRACE 2	5 m 47 s	11 m 0 s	1,751 ms	140 ms
	TRACE 3	8 m 23 s	119 ms	3,145 ms	< 1 ms
	TRACE 4	32 m 54 s	7 h 45 m	71,314 ms	4,577 ms

Table 20.3: *Comparison of the execution times with respect to state-of-the-art approaches.*

approach for GPUs described in [80], *Mangrove* executes up to three orders of magnitude faster[2]. The improvements achieved in *Mangrove* with respect to the parallel approach implemented in [80] are due to the implementation of a more efficient strategy for mapping thread blocks to entries of the variable dictionary, and to the vectorized accesses that best exploit the memory coalescence and the high memory throughput. These aspects are critical to improve the performance, since the memory bandwidth may limit the concurrent memory accesses. Table 20.3 shows that *Mangrove* is efficient also when no invariant can be mined (Traces 1 and 3) thanks to the capability of early terminating the search on a trace as soon as all templates have been falsified. On the contrary, the parallel implementation proposed in [80] always requires to analyse the whole trace to identify the absence of likely invariants, thus wasting time.

20.5 Concluding remarks

The Section presented *Mangrove*, a parallel approach for mining likely invariants by exploiting GPU architectures. Advanced GPU-oriented optimizations and in-

[2] The approach in [80] has been extended in order to support also the template set II.

ference techniques have been implemented in *Mangrove* such that execution traces composed of millions of clock cycles can be generally analysed in less than one second searching for thousands of likely invariants. Experimental results have been conducted on execution traces with different characteristics, and the proposed approach has been compared with sequential and parallel implementations of the most promising state-of-the-art invariant miners. Analysis of the results showed that Mangrove outperforms existing tools.

21

cuRnet: an R package for graph traversing on GPU

R has become the de-facto reference analysis environment in Bioinformatics. Plenty of tools are available as packages that extend the R functionality, and many of them target the analysis of biological networks. Several algorithms for graphs, which are the most adopted mathematical representation of networks, are well-known examples of applications that require high-performance computing, and for which classical sequential implementations are becoming inappropriate. In this context, parallel approaches targeting GPU architectures are becoming pervasive to deal with the execution time constraints. Although R packages for parallel execution on GPUs are already available, none of them provides graph algorithms.

This work presents *cuRnet*, a R package that provides a parallel implementation for GPUs of the breath-first search (BFS), the single-source shortest paths (SSSP), and the strongly connected components (SCC) algorithms. The package allows offloading computing intensive applications to GPU devices for massively parallel computation and to speed up the runtime up to one order of magnitude with respect to the standard sequential computations on CPU. We have tested *cuRnet* on a benchmark of large protein interaction networks and for the interpretation of high-throughput omics data thought network analysis.

cuRnet is a R package to speed up graph traversal and analysis through parallel computation on GPUs. We show the efficiency of *cuRnet* applied both to biological network analysis, which requires basic graph algorithms, and to complex existing procedures built upon such algorithms.

21.1 Background

Biological networks are seen as graphs, where vertices represent elements and edges are the relationships among them. Analyzing biological networks mostly means applying basic graph traversal algorithms to find, for instance, how two vertices are connected, which vertices can be reached by a source, and which part of the network is highly interconnected, i.e., every vertex is reachable from every other vertex. These tasks are commonly embedded in more crucial sophisticated analyses [224, 235] to predict, for example, protein functions [241] or to study complex diseases by relating protein interaction networks to specific conditions [16, 17, 197, 248]. Due to the constantly increasing data set complexity, such applications require high-performance algorithms, for which classical sequential implementations are become inappropriate. Alternative solutions are given by parallel approaches, and in particular by those based on GPU architectures, which allow sensibly reducing the algorithm execution time.

In the context of biological network analysis and, more in general, for statistical computing in Bioinformatics, R is becoming one of the most widely used programming environment. It provides easy-to-use packages to programmers and analysts for efficient and flexible data modeling and analysis [108]. In this context, even though some R packages based on GPU kernels have been proposed (e.g., *gpuR* for algebraic operations https://cran.r-project.org/package=gpuR), none of them provides parallel implementations of algorithms for network analysis.

This work presents *cuRnet*, an R package that provides a wrap of parallel graph algorithms to the R environment. As an initial proof of concept, *cuRnet* includes basic data structures for representing graphs, a parallel implementation of Breadth-First Search (BFS) [59], Single Source Shortest Paths (SSSP) [62], and Strongly Connected Components (SCC) [18]. The package makes available GPU solutions to R end-users in a transparent way, such that GPU modules are invoked by R functions.

cuRnet has been compared with the BFS, SSSP, and SCC implementation of the iGraph R package (http://igraph.org/r/). Tests were run over on annotated undirected protein interaction networks and on directed homology networks provided by the STRINGdb [103].

cuRnet outperformed the iGraph sequential algorithms especially on the largest networks. An average speed-up of 3x have been observed, with a maximum of 30x.

cuRnet SCC and SSSP were used to underscore their ability in helping researchers in providing clues on putative functional context of ncRNA molecules, and guide the selection of a relevant functional readout [267]. For this aim, we used available RNA sequencing dataset of 21 prostate cancer cell lines (GEO accession number GSE25183) to predict coexpression networks. We also show how enabling the GPU implementation of graph traversal algorithms in R has a potential to speed up existing complex procedures whose implementation mainly depends on such calculations. The PCSF package for R [15] is an example, which solves the Prize-collecting Steiner Forest problem by making a massive use of SSSP. It performs user-friendly analysis of high-throughput data using the interaction networks (protein-protein, protein-metabolite or any other type of correlation-based interaction networks) as a template. It interprets the biological landscape of interactome with respect to the data, i.e., to detect high-scoring neighbourhoods to identify functional modules. A real case application of intensive PCSF computation is reported on the analysis of Diffuse large B-cell lymphoma gene expression data.

cuRnet and the PCSF application accelerated with *cuRnet* are freely available on https://bitbucket.org/curnet/curnet.

21.2 Methods

Figure 21.2 shows an overview of the full *cuRnet* stack, by which R data is passed, as input data, to the GPU environment for parallel computation. The input net-

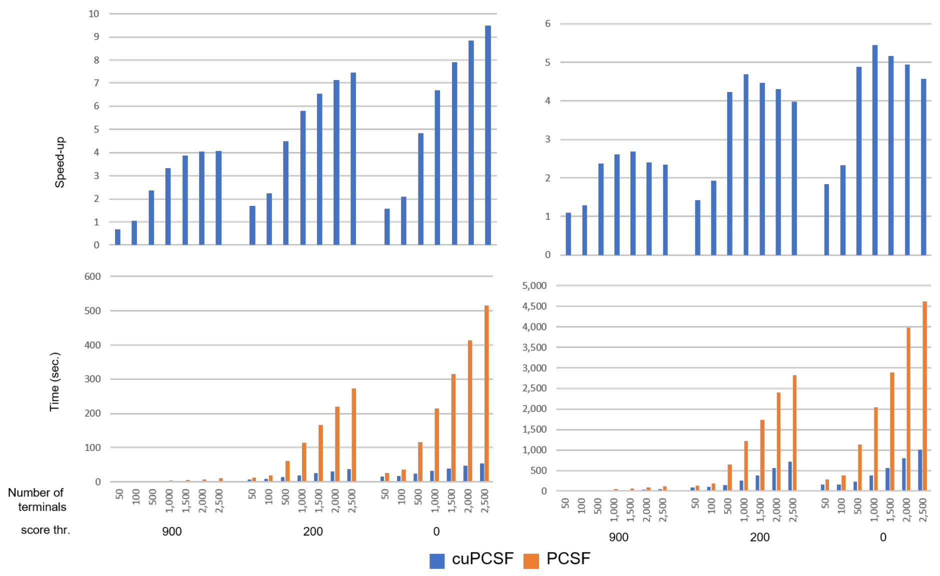
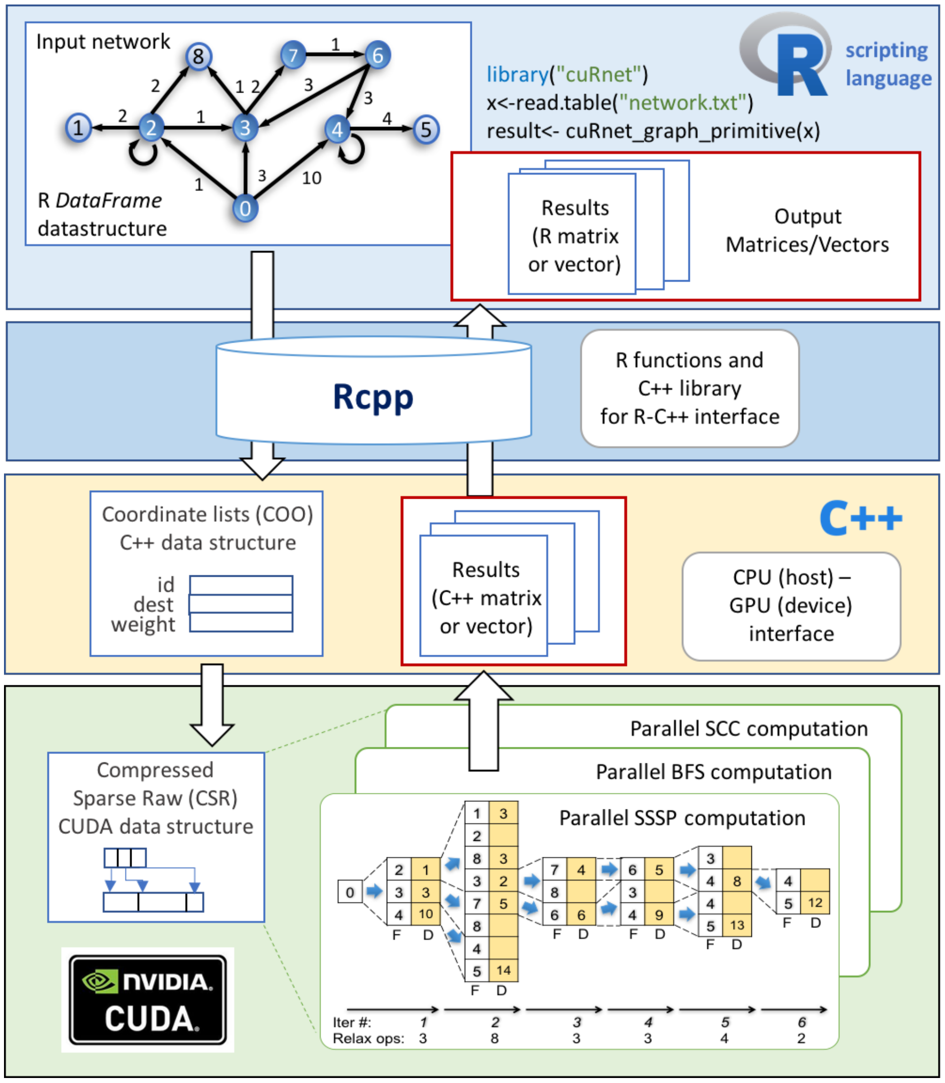


FIG. 21.1: ***Performance of the GP-GPU aided PCSF package versus its serial counterpart.*** *Charts show running time and related speed-ups of the original PCSF R package and the modified version where the SSSP primitive of the Boost library has been replaced with the GP-GPU based approach, named cuPCSF. Tests were performed on the human direct label PPI network by applying three score thresholds. Right-side charts show performances for a single PCSF run, while charts on the right side show executions of randomized selections. The GP-PGU based PCSF reaches speed-ups up to 9x. The parallelized version outperforms better on increasing the network size as well as the amount of terminal vertices. Randomization procedures introduce additional non-parallelized steps performed by the methodology, thus speed-ups reach a maximum of 5x.*

work is represented, in R, through a standard R data frame, where every edge between two vertices is stored with the corresponding weight. By exploiting the *Rcpp* library of R, an R-C++ wrapper has been developed to automatically translate the network from the standard R representation to a C++ data structure, and to link the algorithm invocation from the R to the C++ environment.

The network representation in the C++ environment relies on the coordinate list (COO) data structure, which is a mandatory step to generate the compressed sparse raw (CSR) data structure for the GPU computation. CSR is a well-known storage format to efficiently represent graphs, and it allows reaching high performance during the graph traversal on the GPU.

The C++ interface allows handling the interaction with the GPU device. It generates the host (CPU) representation of the graph starting from the rows in the data frame, it initializes the GPU kernel, it handles the host (CPU)-device (GPU) data exchanging, and, finally, it runs the kernel for the parallel computation. The computation result is retrieved from the device and passed back to R through the Rcpp/C++ layers.



Input network
R DataFrame datastructure
scripting language
library("cuRnet")
x<-read.table("network.txt")
result<- cuRnet_graph_primitive(x)
Results (R matrix or vector)
Output Matrices/Vectors
Rcpp
R functions and C++ library for R-C++ interface
Coordinate lists (COO) C++ data structure
id
dest
weight
Results (C++ matrix or vector)
C++
CPU (host) – GPU (device) interface
Compressed Sparse Raw (CSR) CUDA data structure
NVIDIA CUDA
Parallel SCC computation
Parallel BFS computation
Parallel SSSP computation
Iter #:
Relax ops:

FIG. 21.2: *cuRnet stack overview.*

In what follows we briefly describe the parallel graph traversal algorithms implemented in *cuRnet*. Given a graph $G(V, E)$, with a set V of vertices, a set E of edges, and a weight function $w : E \rightarrow R$, *cuRnet* takes G in a dataframe x having three columns listing the network edges and their weights. The dataframe can be built from an iGraph object or from a textual file (.csv). The following lines invoke the loading of the *cuRnet* package and the construction of the graph data structure:

```
library(cuRnet)
cuRnet_graph(x)
```

We refer the reader to (https://bitbucket.org/curnet/curnet) for a complete manual of the *cuRnet* usage.

21.2.1 Parallel implementation of Breadth-First Search for GPUs

The parallel graph traversal through BFS [59], which is listed in Section 1 - Algorithm 1 in the supplementary materials, explores the reachable vertices, level-by-level, starting from a source s. *cuRnet* implements the concept of frontier [74] to achieve work efficiency. A frontier holds all and only the vertices visited at each level. The algorithm checks every neighbour of a frontier vertex to see whether it has been already visited. If not, the neighbour is added into a new frontier. *cuRnet* implements a frontier propagation step through two data structures, F_1 and F_2. F_1 represents the actual frontier, which is read by the parallel threads to start the propagation step. F_2 is written by the threads to generate the frontier for the next BFS step. At each step, F_2 is filtered and swapped into F_1 for the next iteration. When a thread visits an already visited neighbour, that neighbour is eliminated from the frontier. When more threads visit the same neighbour in the same propagation step, they generate duplicate vertices in the frontier. *cuRnet* implements efficient duplicate detection and correction strategies based on hash tables, advanced strategies for coalesced memory accesses, and warp shuffle instructions. Moreover, it implements different strategies to deal with the potential workload imbalance and thread divergence caused by any actual biological network non-homogeneity. These include prefix-sum procedures to efficiently handle frontiers, dynamic virtual warps, dynamic parallelism, multiple CUDA kernels, and techniques for coalesced memory accesses.

The BFS result is a matrix $s \times |V|$, where s is the number of vertex sources from which the BFS is run. Each entry in the matrix is the depth of the BFS from a source to a graph vertex. The matrix is retrieved from the GPU device to R through the Rcpp/C++ layers. BFS is ran by invoking the following *cuRnet* function in the R environment:

```
depths <- cuRnet_bfs(g, c(sources))
```

Parallel implementation of Single-Source-Shortest-Path for GPU

The *cuRnet* CUDA implementation of the SSSP algorithm is based on the Bellman-Ford's approach [62]. The parallel algorithm is reported in Section 1 of the supplementary materials. *cuRnet* SSSP visits the graph and finds the shortest path d to reach every vertex of V from source s. Also in this case, *cuRnet* exploits the concept of frontier to deal with the most expensive step of the algorithm (i.e., the relax procedure). At each iteration i, the algorithm extracts, in parallel, the vertices from one frontier and inserts the active neighbours in the second frontier for the next iteration step. Each iteration concludes by swapping the contents of the second frontier (which will be the actual frontier at the next iteration) into the first one. Indeed, the frontiers allow working only on active vertices, i.e., all and only vertices whose tentative distance has been modified and, thus, that must be considered for the relax procedure at the next iteration.

The result is a double numeric matrix (i.e., *distances* and *predecessors*), which are retrieved from the GPU device to R through the Rcpp/C++ layer. They are obtained by invoking the *cuRnet* functions CURNET_SSSP and CURNET_SSSP_DISTS for the matrix of shortest paths (returned as lists of predecessor vertices) and the corresponding source-destination distances:

```
ret <- cuRnet_sssp(g, c(sources))
dists = ret[["distances"]]
preds = ret[["predecessors"]]
```

Parallel implementation of Strongly-Connected Components for GPU

cuRnet implements a multi-step approach that applies different GPU-accelerated algorithms for SCC decomposition [18]. The algorithm is reported in Section 1 of the supplemental materials. The multi-step approach consists of 3 phases. In the first phase it iterates a trimming procedure to identify and delete vertices of G that form trivial SCCs (i.e., vertices with no active successors or predecessors). In the second phase it iterates a forward-backward algorithm to identify the main components. The first step is related to the choice of the pivot for each set, where heuristics can be applied to maximize vertices coverage within a single iteration. Forward and backward closure is then computed from this vertex, and up to four subgraphs are generated. The first one is the component which the pivot belongs to, and it is calculated as the intersection of the forward and backward closure. The other three sets are SCC-closed subgraphs that can be processed in parallel at the next iteration. They correspond to the non-visited vertices in the current set, to the forward closure but not to the backward one, and to the backward-reachable vertices, respectively. In the third phase the approach runs a *coloring* algorithm to decompose the rest of the graph. A unique color is firstly assigned to each vertex. The max color is then propagated to the successor non-eliminated vertices until no more updates are possible. Pivots are chosen as the vertices which color is unchanged. Running the backward closure from these vertices on the corresponding set, *cuRnet* detects the components labelled with that color.

The *cuRnet* SCC computation results in a vector of associations between vertices and strongly component IDs. It is retrieved from the GPU device to R through the Rcpp/C++ layer and obtained by invoking the following *cuRnet* function:

```
scc_ids <- cuRnet_scc(g)
```

21.3 Results and discussion

We evaluated the *cuRnet* performance by comparing its execution time with the corresponding sequential implementations provided in the *iGraph* R package (http://igraph.org/r/). *cuRnet* has been tested on an Ubuntu 16.04 OS with CUDA v8.0. The *cuRnet* software requires a GPU device with compute capabilities at least 3.0.

21.3.1 Data

We used the STRING dataset [103], which mainly contains Protein-Protein Interaction (PPI) networks of several organisms, varying from microbes to eukaryotes. We used the R package STRINGdb to download the data. We refer the reader to Section 2 of the supplementary materials for details on the data.

We retrieved the *undirected unlabeled networks* related to *Homo sapiens*, *Danio rerio* and *Zea mais* (see Figure 1 in the supplementary materials for a description of the network characteristics). Those species were chosen among the organisms having the largest networks stored in STRING, to cover the biological diversity that can be encountered in performing analysis of biological networks. For each network, we varied the threshold on the assigned edge scores to obtain sparse as well as dense networks.

We created a banchmark of *undirected label networks* by using the pvalues of differential expression values regarding the treatment of A549 lung cancer cells by means of Resveratrol, a natural phytoestrogen found in red wine and a variety of plants shown to have protective effects against the disease [103] (see Figure 2 in the supplementary materials). We used such values to label the above networks.

We also created a set of *directed unlabelled networks* (see Figure 3 of the supplementary materials) as follows. We used the complete set of 115 archaea species to create homology networks having incremental amount of involved organisms. The homology information between proteins is measured by sequence BLAST alignments. For each protein, STRING reports the best BLAST hits [262] , w.r.t. the given species. Horizontal gene transfer is a frequent phenomenon in microbes [251], and homology networks are used to search for gene families shared by several organisms [149].

The running time to create graph data structures in *cuRnet* and iGraph from the above datasets is reported in Figures 4 and 5 of the supplemental materials. In general, *cuRnet* requires half the time of iGraph to perform such a task.

21.3.2 *cuRnet* performance

We tested *cuRnet* BFS on undirected unlabeled networks and SSSP on undirected labeled networks related to *Homo sapiens*, *Danio rerio* and *Zea mais* by varying the number of sources ranging from just to few vertices to a 20% of vertices. Figures 21.3 and 21.4 show the execution time of the BFS and SSSP, as well as the corresponding speedup w.r.t. the sequential counterpart.

Figures 6 and 7 of the supplemental materials show the total running time including the call to the function primitives, plus the time required for building the graph data structures. Highly functional networks have small sizes and the execution time of the two implementations is in terms of few seconds, obtaining however speedups up to 5x. The time of both packages highly depends on the number of source vertices, but the slope of *cuRnet* is sensibly lower than iGraph. On average, iGraph shows similar performance up to a small percentage of sources (0.5%). Above that, *cuRnet* shows up to 15x speedup w.r.t. the sequential counterpart. The time requirements and the general speedup are similar for the three species.

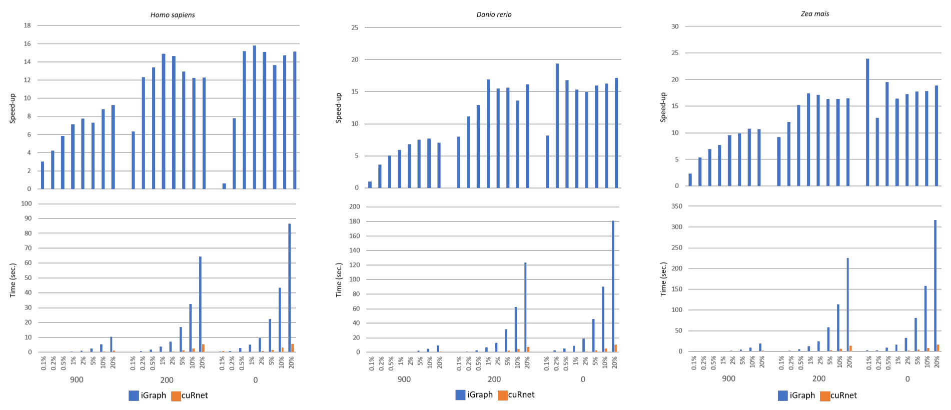


FIG. 21.3: ***cuRnet performance vs iGraph on computing breath first search.*** *Three different score thresholds,* 0, 200 *and* 900, *were applied, and different amounts of source vertices were selected.*

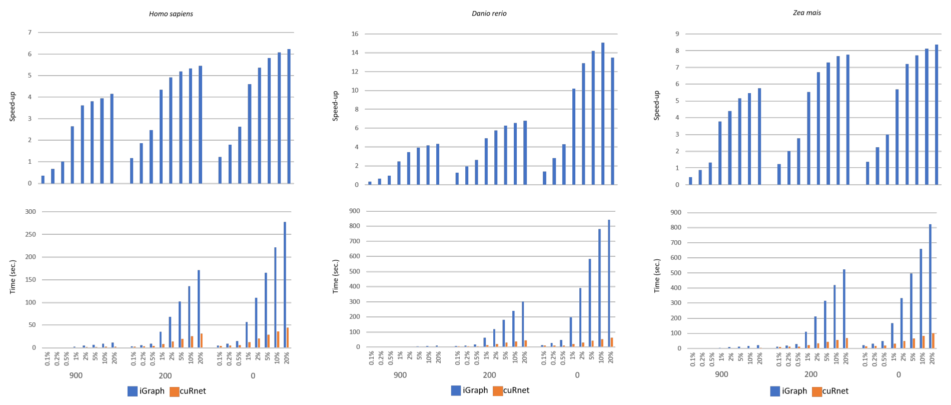


FIG. 21.4: ***cuRnet performance vs iGraph on computing shortest paths distances.*** *Three different score thresholds,* 0, 200 *and* 900, *were applied, and different amounts of source vertices were selected. The underlying charts show running times of cuRnet and iGraph in calculating distance of shortest paths within the PPI of the three selected species for every combination of score threshold and amount of selected sources.*

We tested *cuRnet* SCC performance on directed unlabelled networks representing inter-species proteins homology. Figure 21.5 shows the running time and corresponding speedups by increasing the size of the extracted homology networks, up to the final one of 114 species. Figure 8 in the supplemental materials reports the total running time including the graph data structure generation. *cuRnet* shows an extremely low slope w.r.t. iGraph, and the speedup increases by increasing the network size up to a maximum of 14x.

We also show how *cuRnet* allows users to quickly retrieve ncRNA-pathway associations and individual genes contributing to them. To evaluate the *cuRnet* performance in making highly confident ncRNA function predictions, we analysed a case

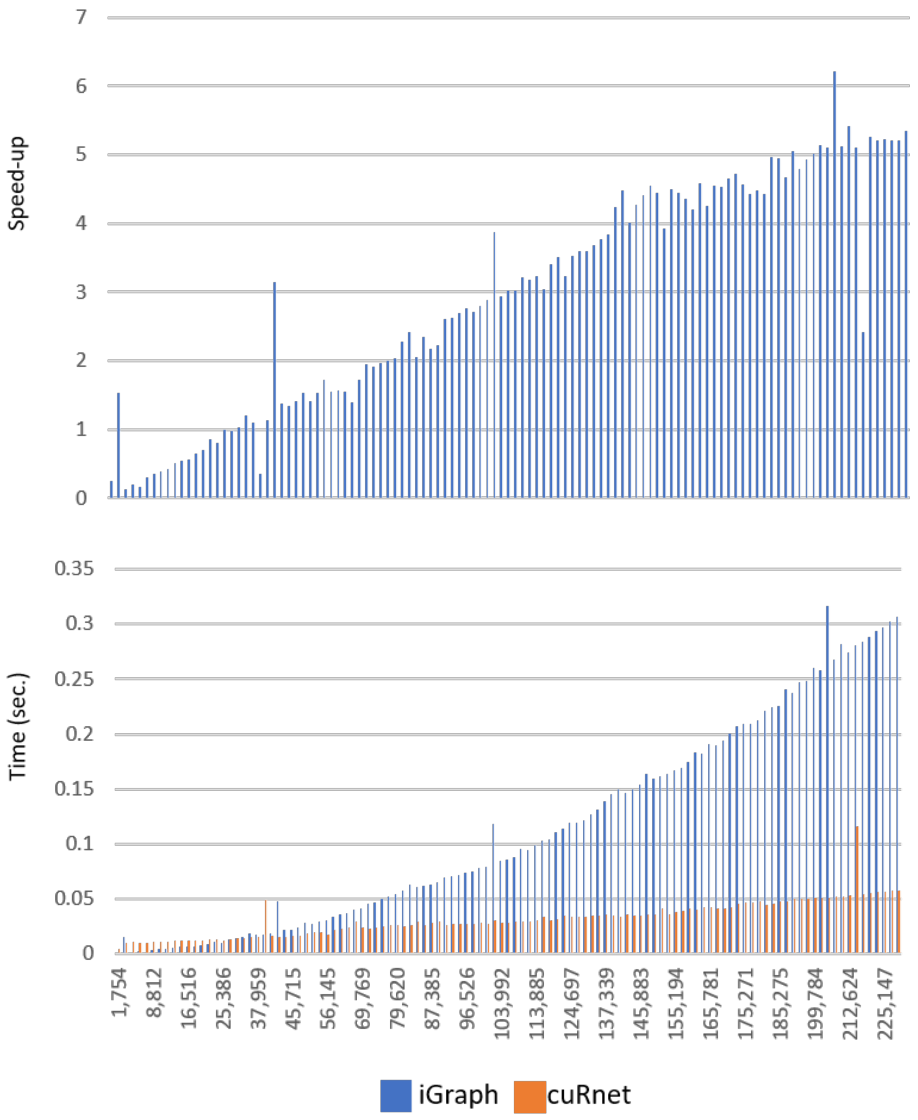


FIG. 21.5: ***cuRnet performance vs iGraph on computing strongly connected components.*** *Running times, and corresponding speed-ups, of cuRnet and iGraph on increasing the size of the extracted homology network, up to the final one of 114 species. Left-side charts show total running, includes the call to the SCC primitive, plus the time required for construction of graph data structures. Right-size charts show comparisons performed by timing only the execution of the SCC algorithm.*

study with the well-known lncRNA involved in cancer called MALAT1. Noncoding RNAs (ncRNAs) are emerging as key molecules in human cancer but only a small number of them has been functionally annotated. Using the guilt-by-association principle is possible to infer functions of lncRNAs on a genome-wide scale [191]. This approach identifies protein coding genes significantly correlated with a given lncRNA using gene-expression analysis. In combination with enrichment strategies, it projects functional protein coding gene sets onto mRNAs correlated with the lncRNA of interest, generating hypotheses for functions and potential regulators of the candidate lncRNA. We used a public RNA sequencing dataset of 21 prostate cancer cell lines sequenced on the Illumina Genome Analyzer and GAII (GEO accession number GSE25183) and built up a large-scale gene association network using *cuRnet* SCC (Pearson method as pairwise correlations). We extracted the sub-networks where MALAT1 is present and calculated single-source shortest paths, mean distance of shortest paths within this subnetwork, and mean distance of shortest paths over the whole big graph. Gene Set Enrichment Analysis (GSEA) was carried out to identify associated biological processes and signalling pathways [163]. We computed overlaps of genes in the MALAT1 sub-networks with gene sets in MSigDB C2 CP (Canonical pathways) and hallmark gene sets [163]. Several cancer related pathways such as epithelial mesenchymal transition (EMT) and DNA replication were enriched, which implies that MALAT1 sub-networks might be involved in the metastasis related pathways. In addition, we identified an over-representation of gene sets that corresponds to the validated MALAT1 functionality reported in the literature: cell cycle, e2f-targets, proliferation, B-MYB-related, and G2M checkpoint [13, 267].

Finally, we tested a modified version of PCSF R package [15] where the original sequential SSSP implementation has been replaced by the parallel SSSP implementation of *cuRnet*. PCSF, taken an input network, may give prizes to vertices according to the measurements of differential expression, copy number, or number of gene mutations. After scoring the interactome, the PCSF identifies high-confidence subnetworks, the neighborhoods in interaction networks potentially belonging to the key pathways that are altered in a disease. It also interactively visualizes the resulting subnetworks with functional enrichment analysis. The running time of the PCSF module is highly dominated by SSSP computations and the application of the *cuRnet* SSSP provided up to 9x speedup for the total execution times of the PCSF (see Figure 21.1). This allows for even more rigorous computations on larger networks.

We applied the PCSF to analyze Diffuse large B-cell lymphoma (DLBCL), which is the most common form of human lymphoma. Based on gene expression profiling studies DLBCL can be divided into two subgroups, the germinal center B-cell (GCB) and the activated B-cell like (ABC), with different clinical outcome and response to therapies [78, 263]. Therefore, it is important to understand underlying molecular mechanism of two subtypes. A public gene expression datasets GSE10846 from Gene Expression Omnibus online repository[1] has been used in the analysis. The dataset is composed of 350 patients being 167 ABC and 183 GCB. We run the PCSF separately for ABC and GCB patients providing top 100 differentially expressed genes as terminals and their absolute fold changes as prizes. The STRING database (version 13) [257] is provided as a template network by applying some filtering steps described in [16], which afterwards had 15405 nodes and 175821 genes.

[1] https://www.ncbi.nlm.nih.gov/geo

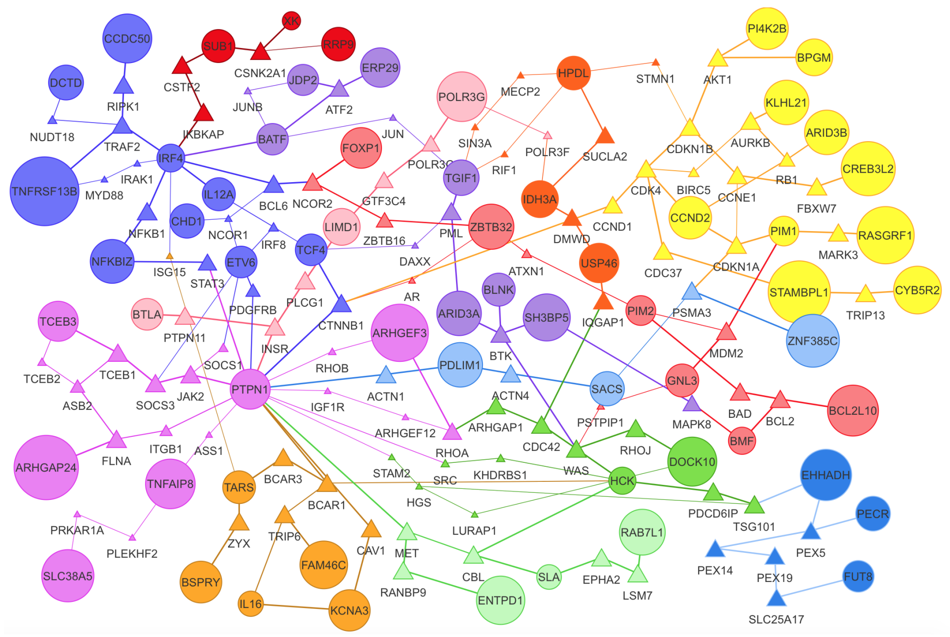


FIG. 21.6: ***The PCSF subnetworks for ABC patients.*** *The node sizes and edge widths are proportional to the number of appearance in multiple PCSF runs. Circular nodes are terminals and algorithm uses triangular nodes to connect terminals. Nodes are colored according to subnetwork membership. The resulting subnetwork for ABC patents was significantly enriched in NFKB pathway (cluster in purple located at top right of the figure) and composed of up-regulated ABC genes including IRF4, FOXP1, IL6, BATF and PIM2.*

An interactive visualization of the subnetwork for ABC patients is shown in Figure 21.6. PCSF also performs enrichment analysis on subnetworks by employing either EnrichR [66] API or topGO [19] that can be specified by the user. For the resulting subnetwork of ABC patients, the hallmark of ABC-DLBCL, as constitutive activation of nuclear factor kappa-B (NFKB) signalling, was confirmed by the enrichment of NFKB pathway (cluster in purple) and up-regulation of well defined ABC genes including IRF4, FOXP1, IL6, BATF and PIM2 among others [225]. In parallel, PCSF subnetwork for GCB patients (see Figure 9 in the supplementary materials) showed activation of the PI3K/Akt/mTOR signalling pathway (cluster in red) and over-expression of germinal center markers such as BCL6, LMO2, MME (CD10) and MYBL1 [221, 225].

21.4 Conclusion

cuRnet has been developed to be easy to use both as a stand-alone analysis application and as a core primitive to be incorporated in more complex algorithmic frameworks. *cuRnet* has been structured to modularly include, as current and future work, a wide collection of algorithms for biological network analysis.

22

Conclusions

Graph analytics and applications are pervasive in sciences and real-world problems as they provide a powerful and expressive representation to model relationships between objects. As computational data complexity and size is consistently growing over the years they will become more and more a primary tool to deal with sophisticated tasks in the future.

The thesis first presented a library of main graph algorithms in the literature that have wide applications in real-world problems. In particular, it introduced an accurate analysis of the load balancing techniques which is a fundamental aspect arising from irregular applications such as graphs. It presented an advanced dynamic technique, called *Multi-phase Mapping*, which addresses the workload unbalancing problem and whose complexity is sensibly reduced with respect to the other techniques in the literature.

Then, it described two high-performance implementations for efficient traverse a graph. The first, called *BFS-4K*, proposes different techniques focusing Kepler GPU architectures to deal with the potential workload imbalance and thread divergence. Thanks to its specialized implementation, BFS-4K is one most efficient BFS implementations for GPUs at the state of the art. BFS-4K has been further improved with *Helix*, a fully configurable BFS for GPUs which thanks to a flexible and expressive programming model, allows selecting the most efficient combination of features and their implementation strategy for a graph traversal. Helix provides high-performance and customized BFSs with speedups ranging from 1.2x to 18.5x with regard to the best parallel BFS solutions for GPUs at state of the art.

Later, the thesis described efficient implementations of three relevant graph problems, single-source shortest path, strongly connected components, and approximate sub-graph isomorphism. More in details, it presented *H-BF*, a fast implementation of the Bellman-Ford algorithm which thanks to different algorithm and architecture optimizations allows to improve the performance and, at the same time, to optimize the work inefficiency typical of the Bellman-Ford algorithm. For SCC decomposition, it introduced a novel parametric *multi-step scheme* allows defining a new set of algorithms for SCC graph decomposition as well as a type of the parallelization for individual graph operations. Finally, in this context, it is presented *APPAGATO*, a stochastic and parallel algorithm to find approximate occurrences of a query network in biological networks. Thanks to its random and parallel nature, it applies to large networks and, compared to existing tools, it provides higher performance as well as statistically significant more accurate results.

The thesis proposed, as second aims, a study of advanced profiling models to analyze performance, energy and power consumption of algorithm implementations. It first introduces *Pro++*, profiling framework for GPU primitives that allows measuring the implementation quality of a given primitive which allows distinguishing the impact of each optimization on the overall quality of the primitive implementation. Then, it presented a *fine-grained performance model* for GPU architectures which relies on microbenchmarks to characterize the GPU device, measure the implementation quality, and to accurately calculate the potential performance of a given application. The initial set of microbenchmarks have been enriched to form *MIPP*, a suite of microbenchmarks that aims at characterizing a GPU device in terms of performance, power, and energy consumption. MIPP aims at understanding how application bottlenecks involving selected functional components or underutilization of them can affect code characteristics on a given device. Finally, the microbenchmark suite has been extended to show how such specialized procedures can be combined with the standard profiler information to efficiently tune any parallel application for a given GPU device and for a given design constraint. In addition, performance, power, and energy consumption have been evaluated on a wide range of load balancing techniques and dataset to provide a comprehensive analysis of this important problem for GPU architectures.

The thesis also focused on extending sparse data structure to handle dynamic graphs and linear algebra problems. In this context, it has been proposed *Hornet*, a new fast dynamic data structure which supports both insertions, deletions, and value updates without sacrifice efficiency as it shows performance comparable, and in some cases even better, than CSR. Thanks to a unique data structure and an advanced memory manager, Hornet provides performance up to one order of magnitude compared to state-of-the-art solutions and only about 30% higher than the storage requirements of CSR. Then, the Hornet data structure has been applied to implement a new algorithm for efficiently finding k-truss subgraphs. The new algorithm, thanks to the dynamic graph data structure, allows avoiding redundant computation and providing significant speedups compared to static solutions in the literature.

Finally, many techniques previously developed in the thesis have been applied in two other contexts, mining likely invariants of a system and an R package targeting graph primitives. The thesis presented *Mangrove*, a fast parallel approach for mining likely invariants on GPUs which allows dealing with execution traces composed of millions of clock cycles in less than one second searching for thousands of likely invariants. Secondly, it introduced *cuRnet*, an R package that provides a parallel implementation for GPUs of breadth-first search, single-source shortest paths, and strongly connected components algorithms. The package allows offloading computing intensive applications to GPU devices to speed up the runtime with respect to the standard sequential computations on CPU.

Future Work

The future work and effort will be spent to provide a comprehensive graph framework, called *HornetsNest* for static and dynamic analytics which allows to easily express any graph algorithm in few lines of code similarly to pseudocode. The programming model will abstract all the complexity of common graph operations data structures such as load balancing, parallel queues, data layouts, etc. allowing the user to focus on the algorithm design rather than implementation details. Despite the simple and flexible programming model it will provide the fastest implementations of a wide range of graph algorithms for parallel architecture, not only restricted to GPUs. The performance of framework primitives will strongly rely on the techniques and methodologies developed during the thesis. The framework capability to deal with evolving graphs will be clearly based on the Hornet data structure, while special cases of static computation will be handled by relying on CSR and COO formats. *HornetsNest* will be a significant step forward towards a unified and comprehensive tool for analytical graph processing compared to the vast universe of graph frameworks currently available in the literature in terms of performance, flexibility, and programmability.

Part VI

Ph.D. Candidate's Bibliografy

- Green, Oded; Fox, James; Kim Euna; Busato, Federico; Bombieri, Nicola; Lakhotia, Kartik; Zhou, Shijie; Singapura, Shreyas; Zeng, Hanqing; Kannan, Rajgopal; Prasanna, Viktor; Bader, David, *"Quickly Finding a Truss in a Haystack"*, In Proc. of IEEE High Performance Extreme Computing Conference (HPEC), IEEE/Amazon/DARPA Graph Challenge, Waltham, USA, September, 12-14, 2017, *Innovation Award*
- Busato, Federico; Green, Oded; Bombieri, Nicola; Bader, David, *Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices"*, Submitted to IEEE International Conference on Supercomputing (ICS), Beijing, China, June 13-15, 2018.
- Bonnici Vincenzo; Busato, Federico; Aldegheri, Stefano; Akhmed, Murodzhon; Cascione, Luciano; Arribas Carmena, Alberto; Bertoni, Francesco; Bombieri, Nicola; Kwee, Ivo; Giugno, Rosalba,*"cuRnet: an R package for graph traversing on GPU"*, In Proc. of Bioinformatics Italian Society (BITS2017), Cagliari, Italy, July 5-7, 2017.
- Busato, Federico; Bombieri, Nicola, *"A performance, power, and energy efficiency analysis of load balancing techniques for GPUs"*, In Proc. of IEEE International Symposium on Industrial Embedded Systems (SIES), Toulouse, France, 14-16 June, 2017, pp. 1-10.
- Busato, Federico; Bombieri, Nicola, *"Helix: A Fully Configurable Breadth-first Search for GPUs"*, Submitted to IEEE Transactions on Computers (TC).
- Bombieri, Nicola; Busato, Federico; Fummi, Franco; *"Power-aware Performance Tuning of GPU Applications Through Microbenchmarking"*, In Proc. of ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, Texas, USA, June 18-22, 2016, pp. 1-6, HiPEAC Award.
- Busato, Federico; Danese, Alessandro; Piccolboni, Luca; Pravadelli, Graziano; Bombieri, Nicola, *"Mangrove: Efficient dynamic invariant mining on GPU architectures"*, Submitted to IEEE Transactions of Parallel and Distributed Systems (TPDS), 2018, pp. 1-14.
- Aldegheri, Stefano; Barnat, Jiri; Bombieri, Nicola; Busato, Federico; Ceska, Milan, *"Parametric Multi-Step Scheme for GPU-Accelerated Graph Decomposition into Strongly Connected Components"*, to International European Conference on Parallel and Distributed Computing (Europar), Workshop on Performance Engineering for Large Scale Graph Analytics (PELGA), Grenoble, France, August 24-26, 2016, pp. 1-12.
- Bombieri, Nicola; Busato, Federico; Fummi, Franco; Scala, Michele, *"MIPP: A Microbenchmark Suite for Performance, Power, and Energy Consumption Characterization of GPU architectures"*, In Proc. of IEEE International Symposium on Industrial Embedded Systems (SIES), Krakow, Poland, May 23-25, 2016, pp. 1-8.
- Busato Federico; Bombieri, Nicola, *"A dynamic approach for workload partitioning on GPU architectures"*, In IEEE Transactions of Parallel and Distributed Systems (TPDS), DOI: 10.1109/TPDS.2016.2631166, 2016, pp. 1-14

- Bombieri, Nicola; Busato, Federico; Fummi, Franco, *"A Fine-grained Performance Model for GPU Architectures"*, In Proc. of ACM/IEEE International Conference on Design, Automation and Test in Europe (DATE), Dresden, Germany, March 14-18, 2016, pp. 1-8.
- Bombieri, Nicola; Busato, Federico; Danese Alessandro; Piccolboni, Luca; Pravadelli, Graziano, *"Exploiting GPU Architectures for Dynamic Invariant Mining"*, In Proc. of IEEE International Conference on Computer Design (ICCD), New York City, NY-USA, October 18-21, 2015, pp. 192-195.
- Busato, Federico; Bombieri, Nicola, *"On the Load Balancing Techniques for GPU Applications Based on Prefix-scan"*, In Proc. of IEEE International Symposium on Embedded Multicore/Many-core System-on-Chip (MCSoC), Turin, Italy, September 23-25, 2015, pp. 88-95.
- Bombieri, Nicola; Busato, Federico; Fummi, Franco, *"An enhanced Profiling Framework for the Analysis and Development of Parallel Primitives for GPUs"*, In Proc. of IEEE International Symposium on Embedded Multicore/Many-core System-on-Chip (MCSoC), Turin, Italy, September 23-25, 2015, pp. 1-8.
- Busato, Federico; Bombieri, Nicola, *"Graph Algorithms on GPUs"*, in "Advanced in GPU Research and Practice" Elsevier, in printing 2016.
- Bonnici, Vincenzo; Busato, Federico; Micale, Giovanni; Giugno, Rosalba; Pulvirenti, Alfredo; Bombieri, Nicola, *"APPAGATO: APproximate PArallel and stochastic GrAph searching TOol for biological graphs"*, (10.1093/bioinformatics/btw223) In Bioinformatics, pp. 1-7, 2015.
- Bombieri, Nicola; Busato, Federico; Fummi, Franco, *"Pro++: A Profiling Framework for Primitive-based GPU Programming"*, In IEEE Transactions on Emerging Topics in Computing (TECT), pp 1-12, 2015.
- Busato, Federico; Bombieri, Nicola *"An efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures"*, In IEEE Transactions of Parallel and Distributed Systems (TPDS), Vol. 27, no. 8, pp. 2222-2233, 2016.
- Busato, Federico; Bombieri, Nicola, *"BFS-4K: an Efficient Implementation of BFS for Kepler GPU Architectures"*, In IEEE Transactions of Parallel and Distributed Systems (TPDS), Vol. 26, no. 7, pp. 1826-1838, 2015.

References

[1] 10th DIMACS Implementation Challenge. http://www.cc.gatech.edu/dimacs10/index.shtml.

[2] CLPP - OpenCL Parallel Primitives Library. http://gpgpu.org/2011/06/03/opencl-parallel-primitives-library.

[3] GTgraph: A suite of synthetic random graph generators. http://www.cse.psu.edu/~madduri/software/GTgraph/.

[4] Hybrid System Architecture - HSA Foundation. http://www.hsafoundation.com.

[5] Matlab: Statistics and machine learning toolbox.

[6] NVIDIA CUDA ZONE - GPU-accelerated libraries. https://developer.nvidia.com/gpu-accelerated-libraries.

[7] NVIDIA GEFORCE GTX 780. http://www.nvidia.com/gtx-700-graphics-cards/gtx-780/.

[8] NVIDIA Tegra X1. http://www.nvidia.com/object/tegra.html.

[9] NVidia Tesla V100 GPU Architecture.

[10] OpenACC - Directives for Accelerators. http://www.openacc-standard.org/.

[11] Qualcomm Snapdragon. http://www.qualcomm.com/products/snapdragon.

[12] Spiral - Software/Hardware Generation for DSP Algorithms. http://www.spiral.net/bench.html.

[13] The noncoding rna malat1 is a critical regulator of the metastasis phenotype of lung cancer cells. *Cancer Research*, 73(3):1180–1189, 2013.

[14] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

[15] M Akhmedov, A Kedaigle, RE Chong, R Montemanni, F Bertoni, Fraenkel E, and Ivo Kwee. Pcsf: An r-package for network-based interpretation of high-throughput data. *PLoS Comput Biol*, 13(7), 2017.

[16] Murodzhon Akhmedov et al. "A Fast Prize-Collecting Steiner Forest Algorithm for Functional Analyses in Biological Networks". In *International*

Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pages 263–276. Springer, 2017.

[17] Salvatore Alaimo et al. ”DT-Web: a web-based application for drug-target interaction and drug combination prediction through domain-tuned network-based inference”. *BMC systems biology*, 9(3):S4, 2015.

[18] Stefano Aldegheri, Jiri Barnat, Nicola Bombieri, Federico Busato, and Milan Ceska. Parametric multi-step scheme for gpu-accelerated graph decomposition into strongly connected components. In *Euro-Par 2016: Parallel Processing Workshops - Euro-Par 2016 International Workshops, Grenoble, France, August 24-26, 2016, Revised Selected Papers*, pages 519–531, 2016.

[19] Adrian Alexa and Jorg Rahnenfuhrer. topgo: enrichment analysis for gene ontology. *R package version*, 2(0), 2010.

[20] David A Bader, Jonathan Berry, Adam Amos-Binks, Daniel Chavarría-Miranda, Charles Hastings, Kamesh Madduri, and Steven C Poulos. Stinger: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation. *Georgia Institute of Technology, Tech. Rep*, 2009.

[21] David A Bader and Kamesh Madduri. Gtgraph: A synthetic graph generator suite. *For the 9th DIMACS Implementation Challenge*, 2006.

[22] David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. 10th dimacs implementation challenge: Graph partitioning and graph clustering, 2011.

[23] David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. Graph partitioning and graph clustering, 10th DIMACS implementation challenge workshop. *Contemporary Mathematics*, 588, 2013.

[24] J. Barnat, P. Bauch, L. Brim, and M. Češka. Computing Strongly Connected Components in Parallel on CUDA. In *IPDPS'11*, pages 541–552. IEEE Computer Society, 2011.

[25] J. Barnat and P. Moravec. Parallel Algorithms for Finding SCCs in Implicitly Given Graphs. In *PDMC'06*, volume 4346 of *LNCS*, pages 316–330. Springer, 2006.

[26] T. Barrett, S.E. Wilhite, P. Ledoux, and C. et al. Evangelista. Ncbi geo: archive for functional genomics data sets update. *Nucleic Acids Research*, 41(D1):D991–D995, 2013.

[27] R. Barshir, O. Shwartz, I.Y. Smoly, and E. Yeger-Lotem. Comparative analysis of human tissue interactomes reveals factors leading to tissue-specific manifestation of hereditary diseases. *PLoS Computational Biology*, 10(6), 2014.

[28] Omer Basha, Dvir Flom, Ruth Barshir, Ilan Smoly, Shoval Tirman, and Esti Yeger-Lotem. Myproteinnet: Build up-to-date protein interaction networks for organisms, tissues and user-defined contexts. *Nucl. Acids Res.*, 43(W!):W258–W263, 1 July 2015.

[29] Michael Bauer, Henry Cook, and Brucek Khailany. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *Proceedings of ACM international conference for high performance computing, networking, storage and analysis*, page Art. n. 12, 2011.

[30] Sean Baxter. Modern GPU, 2013.

[31] Sean Baxter. Modern gpu, 2013.

[32] Sean Baxter. Modern gpu, 2014.

[33] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proc of IEEE SC*, pages 1–10, 2012.

[34] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient Semi-streaming Algorithms for Local Triangle Counting in Massive Graphs. In *14th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, pages 16–24, 2008.

[35] Daniel Bedard, Min Yeol Lim, Robert Fowler, and Allan Porterfield. Powermon: Fine-grained and integrated power monitoring for commodity computer systems. In *Proc. of IEEE SoutheastCon*, pages 479–484, 2010.

[36] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[37] Nathan Bell and Michael Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *Proceedings of the ACM Conference on High Performance Computing Networking, Storage and Analysis*, page Art. n.18, 2009.

[38] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[39] V. Bertacco, D. Chatterjee, N. Bombieri, F. Fummi, S. Vinco, A.M. Kaushik, and H.D. Patel. On the use of gp-gpus for accelerating compute-intensive eda applications. In *Proceedings -Design, Automation and Test in Europe, DATE*, pages 1357–1366, 2013.

[40] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide simd many-core architectures. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 159–166, 2009.

[41] Timo Bingmann. STX B+ Tree C++ Template Classes.

[42] Mauro Bisson, Massimo Bernaschi, and Enrico Mastrostefano. Parallel distributed breadth first search on the Kepler architecture. *IEEE Transactions on Parallel and Distributed Systems*, 27(7):2091–2102, 2015.

[43] F. Bistaffa, A. Farinelli, and N. Bombieri. Optimising memory management for belief propagation in junction trees using gpgpus. In *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, volume 2015-April, pages 526–533, 2014.

[44] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.

[45] Guy E Blelloch. Prefix sums and their applications. 1990.

[46] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, Cambridge, MA, USA, 1990.

[47] Vincent Bloemen, Alfons Laarman, and Jaco van de Pol. Multi-core on-the-fly SCC decomposition. In *PPoPP'16*, pages 8:1–8:12. ACM, 2016.

[48] Sergey Bochkanov and Vladimir Bystritsky. Alglib-a cross-platform numerical analysis and data processing library. *ALGLIB Project. Novgorod, Russia*, 2011.

[49] N. Bombieri, F. Fummi, and S. Vinco. On the automatic generation of gpu-oriented software applications from rtl ips. In *2013 International Conference*

on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2013, 2013.

[50] RM Bonelli and JL Cummings. Cfrontal-subcortical circuitry and behavior. *ialogues Clin Neurosci.*, 9(2):141–151, 2007.

[51] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics*, 14(Suppl 7):S13, 2013.

[52] Elizabeth I. et al. Boyle. Go::termfinder–open source software for accessing gene ontology information and finding significantly enriched gene ontology terms associated with a list of genes. *Bioinformatics*, 20(10):3710–3715, 2004.

[53] Victor Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *J. of Object Technology*, 5(5):31–58, 2006.

[54] Aydın Buluç, John R Gilbert, and Ceren Budak. Solving path problems on the gpu. *Parallel Computing*, 36(5):241–253, 2010.

[55] Luciana S Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting Triangles in Data Streams. In *25th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 253–262, 2006.

[56] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on gpus. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151. IEEE, 2012.

[57] F. Busato and N. Bombieri. BFS-4K: An efficient implementation of BFS for kepler GPU architectures. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):1826–1838, July 2015.

[58] F. Busato and N. Bombieri. On the load balancing techniques for GPU applications based on prefix-scan. In *Proc. of IEEE MCSoC*, pages 88–95, 2015.

[59] Federico Busato and Nicola Bombieri. BFS-4K: an efficient implementation of BFS for kepler GPU architectures. *IEEE Transactions on Parallel Distributed Systems*, 26(7):1826 – 1838, 2015.

[60] Federico Busato and Nicola Bombieri. BFS-4K: an efficient implementation of BFS for kepler GPU architectures. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):1826–1838, 2015.

[61] Federico Busato and Nicola Bombieri. A dynamic approach for workload partitioning on GPU architectures. *IEEE Trans. on Parallel Distributed Systems*, preprint(99):1–15, 2016.

[62] Federico Busato and Nicola Bombieri. An Efficient Implementation of the Bellman-Ford Algorithm for Kepler GPU Architectures. *IEEE Transactions on Parallel Distributed Systems*, 27(8):2222–2233, 2016.

[63] Federico Busato and Nicola Bombieri. A dynamic approach for workload partitioning on gpu architectures. *IEEE Transactions on Parallel and Distributed Systems*, 28(6):1535–1549, 2017.

[64] V.T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal. Scalable single source shortest path algorithms for massively parallel systems. pages 889–901, 2014.

[65] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pages 666–675, 1990.

[66] Edward Y Chen, Christopher M Tan, Yan Kou, Qiaonan Duan, Zichen Wang, Gabriela Vaz Meirelles, Neil R Clark, and Avi Maayan. Enrichr: interactive and collaborative html5 gene list enrichment analysis tool. *BMC bioinformatics*, 14(1):128, 2013.

[67] L. Chen, O. Villa, S. Krishnamoorthy, and G.R. Gao. Dynamic load balancing on single- and multi-gpu systems. pages 1–12, 2010.

[68] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA C Programming*. John Wiley & Sons, 2014.

[69] John Cheng, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014.

[70] Boris V Cherkassky, Andrew V Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical programming*, 73(2):129–174, 1996.

[71] Jun-Dong Cho, Salil Raje, and Majid Sarrafzadeh. Fast approximation algorithms on maxcut, k-coloring, and k-color ordering for vlsi applications. *IEEE Trans. Comput.*, 47(11):1253–1266, November 1998.

[72] Jonathan Cohen. Trusses: Cohesive Subgraphs for Social Network Analysis. *National Security Agency Technical Report*, page 16, 2008.

[73] Jonathan Cohen. Graph Twiddling in a Map-Reduce World. *Computing in Science & Engineering*, 11(4):29–41, 2009.

[74] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT press, 2009.

[75] B. Coutinho, D. Sampaio, F.M.Q. Pereira, and W. Meira Jr. Profiling divergences in gpu applications. *Concurrency Computation Practice and Experience*, 25(6):775–789, 2013.

[76] Joseph R Crobak, Jonathan W Berry, Kamesh Madduri, and David A Bader. Advanced shortest paths algorithms on a massively-multithreaded architecture. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[77] Christoph Csallner and Yannis Smaragdakis. Check 'n' crash: Combining static checking and testing. In *Proc. of ACM/IEEE ICSE*, pages 422–431, 2005.

[78] Riccardo Dalla-Favera. Molecular genetics of aggressive b-cell lymphoma. *Hematological Oncology*, 35(S1):76–79, 2017.

[79] Alessandro Danese, Tara Ghasempouri, and Graziano Pravadelli. Automatic extraction of assertions from execution traces of behavioural models. In *Proc. of ACM/IEEE DATE*, pages 1–6, 2015.

[80] Alessandro Danese, Luca Piccolboni, and Graziano Pravadelli. A parallelizable approach for mining likely invariants. In *Proc. of ACM/IEEE CODES+ISSS*, 2015.

[81] Hoang-Vu Dang and Bertil Schmidt. The sliced coo format for sparse matrix-vector multiplication on cuda-enabled gpus. *Procedia Computer Science*, 9:57–66, 2012.

[82] A. Davidson, S. Baxter, M. Garland, and J.D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. pages 349–359, 2014.

[83] Andrew Davidson, Sean Baxter, Michael Garland, and John D Owens. Work-efficient parallel GPU methods for single-source shortest paths. In *Proc. of IEEE IPDPS*, pages 349–359, 2014.

[84] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[85] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[86] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[87] Camil Demetrescu, Andrew Goldberg, and David Johnson. 9th dimacs implementation challenge–shortest paths. *American Mathematical Society*, 2006.

[88] Camil Demetrescu, Andrew V Goldberg, and David S Johnson. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74. American Mathematical Soc., 2009.

[89] Shrinivas Devshatwar, Madhur Amilkanthwar, and Rupesh Nasre. GPU centric extensions for parallel strongly connected components computation. In *GPGPU'16*, pages 2–11. ACM, 2016.

[90] Robert Dietrich, Felix Schmitt, Rene Widera, and Michael Bussmann. Phase-based profiling in gpgpu kernels. In *Proc. IEEE ICPPW*, pages 414–423, 2012.

[91] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[92] Hristo Djidjev, Sunil Thulasidasan, Guillaume Chapuis, Rumen Andonov, and Dominique Lavenier. Efficient multi-gpu computation of all-pairs shortest paths. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 360–369. IEEE, 2014.

[93] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. Fast scan algorithms on graphics processors. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 205–213, 2008.

[94] D. Ediger., K. Jiang, J. Riedy, and D. Bader. Massive Streaming Data Analytics: A Case Study with Clustering Coefficients. In *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, 2010.

[95] D. Ediger, R. McColl, J. Riedy, and D.A. Bader. STINGER: High Performance Data Structure for Streaming Graphs. In *IEEE High Performance Embedded Computing Workshop (HPEC 2012)*, pages 1–5, Waltham, MA, 2012.

[96] Nick Edmonds, Alex Breuer, Douglas Gregor, and Andrew Lumsdaine. Single-source shortest paths with the parallel boost graph library. *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem, Piscataway, NJ*, pages 219–248, 2006.

[97] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[98] Z. Fang, S. Mehta, P.-C. Yew, A. Zhai, J. Greensky, G. Beeraka, and B. Zang. Measuring microarchitectural details of multi- and many-core memory systems through microbenchmarking. *ACM Transactions on Architecture and Code Optimization*, 11(4):art. n.5, 2015.

[99] Wu-chun Feng and Kirk Cameron. The green500 list: Encouraging sustainable supercomputing. *Computer*, 40(12):50–55, 2007.

[100] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation inference for modular checkers. *Inf. Process. Lett.*, 77(2-4):97–108, 2001.

[101] L. K. Fleischer, B. Hendrickson, and A. Pinar. On Identifying Strongly Connected Components in Parallel. In *IPDPS'00*, volume 1800 of *LNCS*, pages 505–511. Springer, 2000.

[102] L. R. Ford. *Network flow theory.* Santa Monica, Calif. : Rand Corp., 1956.

[103] Andrea Franceschini et al. "string v9. 1: protein-protein interaction networks, with increased coverage and integration". *Nucleic acids research*, 41(D1):D808–D815, 2012.

[104] Vijay Gadepally, Jake Bolewski, Dan Hook, Dylan Hutchison, Ben Miller, and Jeremy Kepner. Graphulo: Linear Algebra Graph Kernels for NoSQL Databases. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 822–830. IEEE, 2015.

[105] Harold N Garbow. Scaling algorithms for network problems. *Journal of Computer and System Sciences*, 31(2):148–168, 1985.

[106] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with cuda. *IEEE Micro*, 28(4):13–27, 2008.

[107] Michael Garland. Sparse Matrix Computations on Manycore GPU's. In *Proceedings of the 45th annual conference on Design automation (DAC'08)*, pages 2–6. ACM, 2008.

[108] Robert C Gentleman et al. "Bioconductor: open software development for computational biology and bioinformatics". *Genome biology*, 5(10):R80, 2004.

[109] O. Green and D. Bader. Faster Clustering Coefficients Using Vertex Covers. In *5th ASE/IEEE International Conference on Social Computing*, SocialCom, 2013.

[110] O. Green and D.A. Bader. cuSTINGER: Supporting Dynamic Graph Algorithms for GPUS. In *IEEE Proc. High Performance Embedded Computing Workshop (HPEC)*, Waltham, MA, 2016.

[111] O. Green, L.M. Munguia, and D. Bader. Load Balanced Clustering Coefficients. In *ACM Workshop on Parallel Programming for Analytics Applications (PPAA)*, Feb. 2014.

[112] O. Green, P. Yalamanchili, and L.M. Munguía. Fast Triangle Counting on the GPU. In *IEEE Fourth Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–8, 2014.

[113] Oded Green, Robert McColl, and David A Bader. GPU merge path: a GPU merging algorithm. In *Proc. of ACM SC*, pages 331–340, 2012.

[114] CS Greene, A Krishnan, AK Wong, E Ricciotti, RA Zelaya, DS Himmelstein, R Zhang, BM Hartmann, E Zaslavsky, SC Sealfon, DI Chasman, GA FitzGerald, K Dolinski, T Grosser, and OG Troyanskaya. Understanding multicellular function and disease with human tissue-specific networks. *Nature Genetics*, 47:569–576, 27 April 2015.

[115] P. Guo and L. Wang. Accurate cross-architecture performance modeling for sparse matrix-vector multiplication (spmv) on gpus. *Concurrency Computation*, 27(13):3281–3294, 2015.

[116] Sudheendra Hangal, Sridhar Narayanan, Naveen Chandra, and Sandeep Chakravorty. IODINE: a tool to automatically infer dynamic invariants for hardware designs. In *Proc. of ACM/IEEE DAC*, pages 775–778, 2005.

[117] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, 2007.

[118] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *High performance computing–HiPC 2007*, pages 197–208. Springer, 2007.

[119] Mark Harris et al. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*, 2(4), 2007.

[120] Mark Harris and Michael Garland. *GPU Computing Gems Emerald Edition: Optimizing Parallel Prefix Operations for the Fermi Architecture*, chapter 3. Addison Wesley Professional, 2008.

[121] Mark Harris and Michael Garland. Optimizing parallel prefix operations for the Fermi architecture. *GPU Computing Gems Jade Edition*, pages 29–38, 2011.

[122] Mark Harris, John Owens, Shubho Sengupta, Yao Zhang, and Andrew Davidson. Cudpp: Cuda data parallel primitives library, 2014.

[123] R. Hastings and B. Joyce. Joyce. purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter USENIX Conference*, 1991.

[124] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2014.

[125] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 267–276, 2011.

[126] Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. On fast parallel detection of strongly connected components (scc) in small-world graphs. In *SC'13*, pages 92:1–92:11. ACM, 2013.

[127] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, June 2009.

[128] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 280–289, 2010.

[129] DW Huang, R Lempicki, and BT Sherman. Systematic and integrative analysis of large gene lists using david bioinformatics resources. *Nat Protoc*, 4(1):44–57, 2009.

[130] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. Querying K-Truss Community in Large and Dynamic Graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1311–1322. ACM, 2014.

[131] Dylan Hutchison, Jeremy Kepner, Vijay Gadepally, and Bill Howe. From nosql accumulo to newsql graphulo: Design and utility of graph algorithms inside a bigtable database. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–9. IEEE, 2016.

[132] Ankit Jain. *pOSKI: An Extensible Autotuning Framework to Perform Optimized SpMVs on Multicore Architectures*. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley.

[133] Jan Jannink. Implementing deletion in B+-trees. *ACM Sigmod Record*, 24(1):33–38, 1995.

[134] Yuntao Jia, Victor Lu, Jared Hoberock, Michael Garland, and John C.Hart. *GPU Computing Gems Jade Edition: Chapter 2. Edge v. Node Parallelism for Graph Centrality Metrics*, chapter 11. Morgan Kaufmann Publishers, 2011.

[135] Yuntao Jia, Victor Lu, Jared Hoberock, Michael Garland, and John C Hart. Edge v. node parallelism for graph centrality metrics. *GPU Computing Gems*, 2:15–30, 2011.

[136] Eric Jones, Travis Oliphant, and Pearu Peterson. SciPy: Open Source Scientific Tools for Python. 2014.

[137] Humayun Kabir and Kamesh Madduri. Shared-memory graph truss decomposition. *arXiv preprint arXiv:1707.02000*, 2017.

[138] Y Kanai and MA Hediger. The glutamate/neutral amino acid transporter family slc1: Molecular, physiological and pharmacological aspects. *Pflugers Arch Eur J Physiol.*, 447(5):469–479, 2004.

[139] Gary J Katz and Joseph T Kider Jr. All-pairs shortest-paths for large graphs on the gpu. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 47–55. Eurographics Association, 2008.

[140] Kevin Kelley and Tao B Schardl. Parallel single-source shortest paths. 2010.

[141] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. Modeling gpu-cpu workloads and systems. In *Proc. of GPGPU*, pages 31–42, 2010.

[142] Arijit Khan, Yinghui Wu, Charu C. Aggarwal, and Xifeng Yan. Nema: fast graph search with label similarity. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 181–192. VLDB Endowment, 2013.

[143] D Khananshvili. The slc8 gene family of sodium-calcium exchangers (ncx) - structure, function, and regulation in health and disease. *Mol Aspects Med.*, 34(2-3):220–235, 2013.

[144] James King, Thomas Gilray, Robert M Kirby, and Matthew Might. Dynamic Sparse-Matrix Allocation on GPUs. In *International Conference on High Performance Computing*, pages 61–80. Springer, 2016.

[145] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach.* Newnes, 2012.

[146] Steffen Klamt and Axel von Kamp. Computing paths and cycles in biological interaction graphs. *BMC Bioinformatics*, 10(6):1–11, 2014.

[147] T. Komoda, S. Hayashi, S. Miwa, and H. Nakamura. Power capping of CPU-GPU heterogeneous systems through coordinating DVFS and task mapping. In *Proc. of IEEE ICCD*, pages 349–356, 2013.

[148] K. Kothapalli, R. Mukherjee, M. Suhail Rehman, S. Patidar, P.J. Narayanan, and K. Srinathan. A performance prediction model for the cuda gpgpu platform. In *Proc. of IEEE HiPC*, pages 463–472, 2009.

[149] David M Kristensen, Lavanya Kannan, Michael K Coleman, Yuri I Wolf, Alexander Sorokin, Eugene V Koonin, and Arcady Mushegian. A low-polynomial algorithm for assembling clusters of orthologous groups from intergenomic symmetric best matches. *Bioinformatics*, 26(12):1481–1487, 2010.

[150] Jérôme Kunegis. Konect: the koblenz network collection. In *WWW'13*, pages 1343–1350. ACM, 2013.

[151] Jérôme Kunegis and Julia Preusse. Fairness on the web: Alternatives to the power law. In *Proc. of ACM WebSci*, pages 175–184, 2012.

[152] Konstantin Kutzkov and Rasmus Pagh. Triangle counting in dynamic graph streams. In *Scandinavian Workshop on Algorithm Theory*, pages 306–318. Springer, 2014.

[153] Daniel Langr and Pavel Tvrdik. Evaluation Criteria for Sparse Matrix Storage Formats. *IEEE Transactions on Parallel Distributed Systems*, 27(2):428–440, 2016.

[154] CE Lawrence, SF Altschul, MS Boguski, JS Liu, AF Neuwald, and JC Wootton. Detecting subtle sequence signals: a gibbs sampling strategy for multiple alignment. *Science*, 262(5131):208–214, 1993.

[155] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *ACM SIGARCH computer architecture news*, 38(3):451–460, 2010.

[156] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the 22Nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 303–314, 2010.

[157] A Leist, KA Hawick, DP Playne, and North Shore Albany. GPGPU and Multi-Core Architectures for Computing Clustering Coefficients of Irregular Graphs. In *Int'l Conf. on Scientific Computing (CSC'11)*, 2011.

[158] Jan Lemeire, Jan G Cornelis, and Laurent Segers. Microbenchmarks for gpu characteristics: The occupancy roofline and the pipeline model. In *Euromicro PDP*, pages 456–463, 2016.

[159] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, January 1979.

[160] Jure Leskovec et al. Stanford network analysis project. 2010.

[161] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data`, June 2014.

[162] Guohui Li, Zhe Zhu, Zhang Cong, and Fumin Yang. Efficient decomposition of strongly connected components on GPUs. *Journal of Systems Architecture*, 60(1):1 – 10, 2014.

[163] Arthur Liberzon, Aravind Subramanian, Reid Pinchback, Helga Thorvaldsdóttir, Pablo Tamayo, and Jill P. Mesirov. Molecular signatures database (msigdb) 3.0. *Bioinformatics*, 27(12):1739–1740, 2011.

[164] Hang Liu and H Howie Huang. Enterprise: Breadth-first graph traversal on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 68. ACM, 2015.

[165] Xiaodong Liu, Mo Li, Shanshan Li, Shaoliang Peng, Xiangke Liao, and Xiaopei Lu. Imgpu: Gpu accelerated influence maximization in large-scale social networks. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):136–145, 2014.

[166] U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso. A survey of performance modeling and simulation techniques for accelerator-based computing. *IEEE Transactions on Parallel and Distributed Systems*, 26(1):272–281, 2015.

[167] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proc. of the VLDB Endowment*, 5(8):716–727.

[168] Shaofeng Lu, P. Weston, S. Hillmansen, H.B. Gooi, and C. Roberts. Increasing the regenerative braking energy for railway vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 15(181):2506–2515, 2009.

[169] Justin Luitjens. CUDA pro tip: Increase performance with vectorized memory access. `http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-increase-performance-with-vectorized-memory-access/`, December 2013.

[170] Justin Luitjens. Faster Parallel Reductions on Kepler, 2014. `https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/`.

[171] Ben Lund and Justin W Smith. A multi-stage cuda kernel for floyd-warshall. *arXiv preprint arXiv:1001.4108*, 2010.

[172] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, 2010.

[173] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang. GreenGPU: A holistic approach to energy efficiency in GPU-CPU heterogeneous architectures. In *Proc. of IEEE ICPP*, pages 48–57, 2012.

[174] O. Magger, Y.Y. Waldman, E. Ruppin, and R. Sharan. Enhancing the prioritization of disease-causing genes through tissue specific protein interaction networks. *PLoS Computational Biology*, 8(9), 2012.

[175] D. Makkar, D.A. Bader, and O. Green. Exact and Parallel Triangle Counting in Streaming Graphs. Submitted.

[176] James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos. Arrayfire: a gpu acceleration platform, 2014.

[177] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[178] Roongko Doong Marat Boshernitsan and Alberto Savoia. From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In *Proc. of ISSTA*, pages 169–180, 2006.

[179] John D. Owens Mark Harris, Shubhabrata Sengupta. *GPU Gems 3: Parallel Prefix Sum (Scan) with CUDA*, chapter 3. Addison Wesley Professional, 2008.

[180] Pedro J. Martin, Roberto Torres, and Antonio Gavilanes. Cuda solutions for the sssp problem. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 904–913, 2009.

[181] Kazuya Matsumoto, Naohito Nakasato, and Stanislav G Sedukhin. Blocked all-pairs shortest paths algorithm for hybrid cpu-gpu system. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 145–152. IEEE, 2011.

[182] R. McColl, D. Ediger, J. Poovey, D. Campbell, and D. Bader. A Performance Evaluation of Open Source Graph Databases. In *ACM Workshop on Parallel Programming for Analytics Applications (PPAA)*, pages 11–18, 2014.

[183] A. McLaughlin and D. Bader. Revisiting Edge and Node Parallelism for Dynamic GPU Graph Analytics. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1396–1406, 2014.

[184] W. McLendon III, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger. Finding Strongly Connected Components in Distributed Graphs. *Journal of Parallel and Distributed Computing*, 65(8):901–910, 2005.

[185] X. Mei, K. Zhao, C. Liu, and X. Chu. Benchmarking the memory hierarchy of modern GPUs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8707 LNCS:144–156, 2014.

[186] D. Merril and A. Grimshaw. Parallel scan for stream architectures. Technical Report CS-200914, Department of Computer Science, University of Virginia, 2009.

[187] D Merrill. Cub v1.6.4: Cuda unbound, a library of warp-wide, block-wide, and device-wide gpu parallel primitives, 2015.

[188] Duane Merrill. Cub, 2015.

[189] Duane Merrill and Michael Garland. Merge-based parallel sparse matrix-vector multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 58. IEEE Press, 2016.

[190] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, 2012.

[191] P Mestdagh, E Fredlund, F Pattyn, A Rihani, T Van Maerken, J Vermeulen, C Kumps, B Menten, K De Preter, A Schramm, J Schulte, R Noguera, G Schleiermacher, I Janoueix-Lerosey, G Laureys, R Powel, D Nittner, J-C Marine, M Ringnér, F Speleman, and J Vandesompele. An integrative genomics screen uncovers ncrna t-ucr functions in neuroblastoma tumours. *Oncogene*, 29, 2010.

[192] Ulrich Meyer and Peter Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[193] Giovanni Micale, Alfredo Ferro, Alfredo Pulvirenti, and Rosalba Giugno. Spectra: an integrated knowledge base for comparing tissue and tumor specific ppi networks in human. *Frontiers in Bioengineering and Biotechnology*, 3(58), 8 May 2015.

[194] Giovanni Micale, Alfredo Pulvirenti, Rosalba Giugno, and Alfredo Ferro. Gasoline: a greedy and stochastic algorithm for optimal local multiple alignment of interaction networks. *PLoS ONE*, 9(6):e98750, 06 2014.

[195] P Mironowicz, A Dziekonski, and M Mrozowski. A Task-Scheduling Approach for Efficient Sparse Symmetric Matrix-Vector Multiplication on a GPU. *SIAM Journal on Scientific Computing*, 37(6):C643–C666, 2015.

[196] Sparsh Mittal and Jeffrey S. Vetter. A survey of methods for analyzing and improving GPU energy efficiency. *ACM Comput. Surv.*, 47(2):19:1–19:23, August 2014.

[197] Ji Moon et al. "PINTnet: construction of condition-specific pathway interaction network by computing shortest paths on weighted PPI". *BMC systems biology*, 11(2):15, 2017.

[198] YM Mudunuri U, Che A and RM Stephens. biodbnet: the biological database network. *Bioinformatics*, 25(4):555–556, 2009.

[199] John Nickolls and William J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, March 2010.

[200] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proc. of ACM FSE*, pages 11–20, 2002.

[201] N. Nishikawa, K. Iwai, H. Tanaka, and T. Kurokawa. Throughput and power efficiency evaluation of block ciphers on kepler and GCN GPUs using micro-benchmark analysis. *IEICE Transactions on Information and Systems*, E97-D(6):1506–1515, 2014.

[202] NVIDIA. Kepler gk110. `www.nvidia.com/content/PDF/kepler/NV_DS_Tesla_KCompute_Arch_May_2012_LR.pdf`.

[203] NVIDIA. Parallel thread execution isa version 4.1. *http://docs.nvidia.com/cuda/parallel-thread-execution/*, 1, 2014.

[204] NVIDIA. Maxwell architecture. `http://international.download.nvidia.com/geforce.com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF`, 2015.

[205] NVIDIA. PTX: Parallel Thread Execution ISA, 2015. `http://docs.nvidia.com/cuda/parallel-thread-execution/`.

[206] CUDA NVidia. C best practices guide. *NVIDIA, Santa Clara, CA*, 2012.

[207] CUDA NVIDIA. Cuda api reference manual, 2015.

[208] CUDA Nvidia. Nvidia cuda c programming guide. 2015.

[209] NVIDIA Corporation. Kepler Tuning Guide.

[210] NVidia Corporation. NVIDIA Kepler GK110 Architecture Whitepaper, 2012. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-780.

[211] NVidia Corporation. NVIDIA Maxwell GeForce GTX 980 Whitepaper, 2014. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980.

[212] NVidia Corporation. NVIDIA Pascal GeForce GTX 1080 Whitepaper, 2016. http://www.geforce.com/hardware/10series/geforce-gtx-1080.

[213] Nvidia CUDA. Programming guide, 2015. `http://docs.nvidia.com/cuda/cuda-c-programming-guide`.

[214] Saher Odeh, Oded Green, Zahi Mwassi, Oz Shmueli, and Yitzhak Birk. Merge path-parallel merging made simple. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1611–1618. IEEE, 2012.

[215] H. Ortega-Arranz, Y. Torres, D.R. Llanos, and A. Gonzalez-Escribano. A new gpu-based approach to the shortest path problem. pages 505–511, 2013.

[216] S. Orzan. *On Distributed Verification and Verified Distribution.* PhD thesis, Free University of Amsterdam, 2004.

[217] U Pape. Implementation and efficiency of moore-algorithms for the shortest route problem. *Mathematical Programming*, 7(1):212–222, 1974.

[218] DL Pauls, A Abramovitch, SL Rauch, and DA Geller. Obsessive-compulsive disorder: an integrative genetic and neurobiological perspective. *Nat Rev Neurosci.*, 15(6):410–424, 2014.

[219] Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation Search—a log log N Search. *Communications of the ACM*, 21(7):550–553, 1978.

[220] Adam Polak. Counting triangles in large graphs on GPU. *arXiv preprint arXiv:1503.00576*, 2015.

[221] Julia R Pon and Marco A Marra. Clinical impact of molecular features in diffuse large b-cell lymphoma and follicular lymphoma. *Blood*, 127(2):181–186, 2016.

[222] AP Privitera, R Distefano, HA Wefer, A Ferro, A Pulvirenti, and R Giugno. Ocdb: a database collecting genes, mirnas and drugs for obsessive-compulsive disorder. *DATABASE (The OXFORD Journal of Biological Databases and Curation)*, Jul 30;2015:bav069, 2015.

[223] Dolbeau R., Bihan S., and Bodin F. Hmpp: A hybrid multicore parallel programming environment. 2007.

[224] Fabio Rinnone et al. Netmatchstar: an enhanced cytoscape network querying app. *F1000Research*, 4, 2015.

[225] Mark Roschewski, Louis M Staudt, and Wyndham H Wilson. Diffuse large b-cell lymphoma [mdash] treatment approaches in the molecular era. *Nature reviews Clinical oncology*, 11(1):12–23, 2014.

[226] Andreas Ruepp, Brigitte Waegele, Martin Lechner, Barbara Brauner, Irmtraud Dunger-Kaltenbach, Gisela Fobo, Goar Frishman, Corinna Montrone, and H.-Werner Mewes. Corum: the comprehensive resource of mammalian protein complexes. *Nucleic Acids Research*, 38(suppl 1):D497–D501, 2010.

[227] Greg Ruetsch and Paulius Micikevicius. Optimizing matrix transpose in cuda. *Nvidia CUDA SDK Application Note*, 28, 2009.

[228] G. Rustici, N. Kolesnikov, M. Brandizi, and T. et al. Burdett. Arrayexpress updatetrends in database growth and links to data analysis tools. *Nucleic Acids Research*, 41(D1):D987–D990, 2013.

[229] M. Saad. Joint optimal routing and power allocation for spectral efficiency in multihop wireless networks. *IEEE Transactions on Wireless Communications*, 13(5):2530– 2539, 2014.

[230] Yousef Saad. *Iterative methods for sparse linear systems.* Siam, 2003.

[231] Sayed Mohammad Ebrahim Sahraeian and Byung-Jun Yoon. Resque: Network reduction using semi-markov random walk scores for efficient querying of biological networks. *Bioinformatics*, 28(16):2129–2136, 2012.

[232] Ahmet Erdem Sariyuce, C. Seshadhri, Ali Pinar, and Umit V. Catalyurek. Finding the hierarchy of dense subgraphs using nucleus decompositions. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 927–937, 2015.

[233] K. Sato, K. Komatsu, H. Takizawa, and H. Kobayashi. A history-based performance prediction model with profile data classification for automatic task allocation in heterogeneous computing systems. In *Proc. of IEEE ISPA*, pages 135–142, 2011.

[234] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. In *ACM Trans. on Computer Systems*, pages 391–411, 1997.

[235] Giovanni Scardoni et al. Biological network analysis with centiscape: centralities and experimental dataset integration. *F1000Research*, 3, 2014.

[236] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. Efficient breadth-first search on the cell/be processor. *IEEE Transactions on Parallel Distributed Systems*, 19(10):1381–1395, 2008.

[237] R. Sedgewick and K. Wayne. *Algorithms.* Pearson Education, 2011.

[238] Dipanjan Sengupta and Shuaiwen Leon Song. Evograph: On-the-fly efficient mining of evolving graphs on gpu. In *International Supercomputing Conference*, pages 97–119. Springer, 2017.

[239] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. GraphIn: An Online High Performance Incremental Graph Processing Framework. In *European Conference on Parallel Processing*, pages 319–333. Springer, 2016.

[240] Shubhabrata Sengupta, Mark Harris, and Michael Garland. Efficient parallel scan algorithms for gpus. Technical report, NVIDIA, 2009.

[241] Roded Sharan, Igor Ulitsky, and Ron Shamir. Network-based prediction of protein function. *Molecular systems biology*, 3(1):88, 2007.

[242] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices*, volume 48, pages 135–146. ACM, 2013.

[243] Julian Shun and Kanat Tangwongsan. Multicore Triangle Computations Without Tuning. In *IEEE Int'l Conf. on Data Engineering (ICDE)*, 2015.

[244] S. Siddharth, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner. Static graph challenge: Subgraph isomorphism, 2017.

[245] Jacob Siegel, Juergen Ributzka, and Xiaoming Li. Cuda memory optimizations for large data-structures in the gravit simulator. *Journal of Algorithms & Computational Technology*, 5(2):341–362, 2011.

[246] Jeremy G Siek, Lie-Quan Lee, and Andrew Lumsdaine. *Boost Graph Library: User Guide and Reference Manual, The.* Pearson Education, 2001.

[247] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. In *Proc. of ACM SIGPLAN PPoPP*, pages 11–22, 2012.

[248] Sérgio Nery Simões et al. "Shortest paths ranking methodology to identify alterations in PPI networks of complex diseases". In *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, pages 561–563. ACM, 2012.

[249] Gunjan Singla, Amrita Tiwari, and Dhirendra Pratap Singh. New approach for graph algorithms on gpu using cuda. *International Journal of Computer Applications*, 2013.

[250] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. Bfs and coloring-based parallel algorithms for strongly connected components and related problems. In *IPDPS '14*, pages 550–559. IEEE Computer Society, 2014.

[251] Shannon M Soucy, Jinling Huang, and Johann Peter Gogarten. Horizontal gene transfer: building the web of life. *Nature Reviews Genetics*, 16(8):472–482, 2015.

[252] Stanford Network Analysis Platform. Stanford university, 2013.

[253] Monica S. Lam Sudheendra Hangal. Tracking down software bugs using automatic anomaly detection. In *Proc. of ACM/IEEE ICSE*, pages 291–301, 2002.

[254] Makoto Sugawara, Shoichi Hirasawa, Kazuhiko Komatsu, Hiroyuki Takizawa, and Hiroaki Kobayashi. A comparison of performance tunabilities between opencl and openacc. In *Proc. of the 2013 IEEE 7th International Symposium on Embedded Multicore/Manycore System-on-Chip (MCSOC'13)*, pages 147–152, 2013.

[255] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, 2014.

[256] D Szklarczyk, A Franceschini, M Kuhn, M Simonovic, A Roth, P Minguez, T Doerks, M Stark, J Muller, P Bork, LJ Jensen, and C von Mering. The string database in 2011: functional interaction networks of proteins, globally integrated and scored. *Nucleic Acids Res.*, 39(Database issue):D561–8, 2011.

[257] Damian Szklarczyk, Andrea Franceschini, Michael Kuhn, Milan Simonovic, Alexander Roth, Pablo Minguez, Tobias Doerks, Manuel Stark, Jean Muller, Peer Bork, et al. The string database in 2011: functional interaction networks of proteins, globally integrated and scored. *Nucleic acids research*, 39(suppl_1):D561–D568, 2010.

[258] T. Schank and D. Wagner. Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study. In *Experimental & Efficient Algorithms*, pages 606–609. Springer, 2005.

[259] W. Tan, W. Tang, R. Goh, S. Turner, and W. Wong. A code generation framework for targeting optimized library calls for multiple platforms. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–12, 2014.

[260] David Tarjan, Kevin Skadron, and Paulius Micikevicius. The art of performance tuning for cuda and manycore architectures. *Birds-of-a-feather session at SC*, 9, 2009.

[261] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[262] Roman L Tatusov, Eugene V Koonin, and David J Lipman. A genomic perspective on protein families. *Science*, 278(5338):631–637, 1997.

[263] M Testoni, E Zucca, KH Young, and F Bertoni. Genetic lesions in diffuse large b-cell lymphomas. *Annals of Oncology*, 26(6):1069–1080, 2015.

[264] Peter Thoman, Klaus Kofler, Heiko Studt, John Thomson, and Thomas Fahringer. Automatic opencl device characterization: guiding optimized kernel design. In *European Conf. on Parallel Processing*, pages 438–452. Springer, 2011.

[265] Nikolai Tillmann, Feng Chen, and Wolfram Schulte. Discovering likely method specifications. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering*, volume 4260 of *LNCS*, pages 717–736. Springer, 2006.

[266] Quoc-Nam Tran. Designing efficient many-core parallel algorithms for all-pairs shortest-paths using cuda. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 7–12. IEEE, 2010.

[267] Vidisha Tripathi, Zhen Shen, Arindam Chakraborty, Sumanprava Giri, Susan M. Freier, Xiaolin Wu, Yongqing Zhang, Myriam Gorospe, Supriya G. Prasanth, Ashish Lal, and Kannanganattu V. Prasanth. Long noncoding rna malat1 controls cell cycle progression by regulating the expression of oncogenic transcription factor b-myb. *PLOS Genetics*, 9(3):1–18, 03 2013.

[268] M. Uhlen, P. Oksvold, L. Fagerberg, and E. et al. Lundberg. Towards a knowledge-based human protein atlas. *Nature Biotechnology*, 28:1248–1250, 2010.

[269] Gayathri Venkataraman, Sartaj Sahni, and Srabani Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. *Journal of Experimental Algorithmics (JEA)*, 8:2–2, 2003.

[270] S. Vinco, D. Chatterjee, V. Bertacco, and F. Fummi. Saga: Systemc acceleration on gpu architectures. In *Proceedings - Design Automation Conference*, pages 115–120, 2012.

[271] Vasily Volkov. Better Performance at Lower Occupancy. In *Proceedings of the GPU Technology Conference, GTC*, volume 10, page 16, 2010.

[272] Jia Wang and James Cheng. Truss Decomposition in Massive Networks. *Proceedings of the VLDB Endowment*, 5(9):812–823, 2012.

[273] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D Owens. A comparative study on exact triangle counting algorithms on the gpu. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, pages 1–8. ACM, 2016.

[274] Y. Wang, S. Roy, and N. Ranganathan. Run-time power-gating in caches of gpus for leakage energy savings. In *Proc. of ACM/IEEE DATE*, pages 300–303, 2012.

[275] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proc. ACM PPoPP*, pages 265–266, 2016.

[276] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of Sparse Matrix–Vector Multiplication on Emerging Multicore Platforms. *Parallel Computing*, 35(3):178–194, 2009.

[277] Nicholas Wilt. *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education, 2013.

[278] Martin Winter, Rhaleb Zayer, and Markus Steinberger. Autonomous, independent management of dynamic graphs on gpus. In *International Supercomputing Conference*, pages 97–119. Springer, 2017.

[279] Wolfram Mathematica 9. Pseudo diameter, 2012.

[280] Yinglong Xia and Viktor K. Prasanna. Topologically adaptive parallel breadth-first search on multicore processors. In *Proceedings of the 21st International Conference on Parallel and Distributed Computing and Systems*, PDCS09, 2009.

[281] Shucai Xiao and Wu chun Feng. Inter-block gpu communication via fast barrier synchronization. Technical report, Dept. of Computer Science Virginia Tech, 2009.

[282] X. Xiao, A. Moreno-Moral, M. Rotival, L. Bottolo, and E. Petretto. Multi-tissue analysis of co-expression networks by higher-order generalized singular value decomposition identifies functionally coherent transcriptional modules. *PLoS Genetics*, 10(1), 2014.

[283] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[284] Kai Xu, Yiwen Wang, Fang Wang, Yuxi Liao, Qiaosheng Zhang, Hongbao Li, and Xiaoxiang Zheng. Neural decoding using a parallel sequential Monte Carlo method on point processes with ensemble effect. *BioMed research international*, 2014.

[285] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. yaSpMV: yet another SpMV framework on GPUs. In *Acm Sigplan Notices*, volume 49, pages 107–118. ACM, 2014.

[286] X. Yan, X. Shi, L. Wang, and H. Yang. An OpenCL micro-benchmark suite for GPUs and CPUs. *Journal of Supercomputing*, 69(2):693–713, 2014.

[287] Carl Yang, Yangzihao Wang, and John D Owens. Fast Sparse Matrix and Sparse Vector Multiplication Algorithm on the GPU. *Proceedings of IEEE Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, pages 841–847, 2015.

[288] Y. Yang, P. Xiang, M. Mantor, and H. Zhou. Fixing performance bugs: An empirical study of open-source GPGPU programs. In *Proc. of IEEE ICPP*, pages 329–339, 2012.

[289] Shuai Mu Yangdong Deng, Bo D. Wang. Taming irregular eda applications on gpus. In *Proc. of the IEEE International Conference on Computer-Aided Design (ICCAD'09)*, pages 539–546, 2009.

[290] F Benjamin Zhan and Charles E Noon. Shortest path algorithms: an evaluation using real road networks. *Transportation Science*, 32(1):65–73, 1998.

[291] X. Zhang, F. Yan, L. Tao, and D.K. Sung. Optimal candidate set for opportunistic routing in asynchronous wireless sensor networks. pages 1– 8, 2014.

[292] M. Zheng, V.T. Ravi, W. Ma, F. Qin, and G. Agrawal. Gmprof: A low-overhead, fine-grained profiling approach for gpu programs. In *Proc. of IEEE HiPC*, 2012.