On the Load Balancing Techniques for GPU Applications Based on Prefix-scan

Federico Busato
Dept. of Computer Science
University of Verona
Italy
Email: federico.busato@univr.it

Nicola Bombieri
Dept. of Computer Science
University of Verona
Italy
Email: nicola.bombieri@univr.it

***Abstract*—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 paper presents a classification of the mapping strategies at the state of the art and their comparison to understand in which problem they best apply. Then, it presents *Multi-Phase Search*, an advanced dynamic technique that addresses the workload unbalancing problem by fully exploiting the GPU device characteristics. In particular, the proposed technique implements a dynamic mapping of work-units to threads through an algorithm whose complexity is sensibly reduced with respect to the other dynamic approaches in the literature. The paper shows, compares, and analyses the experimental results obtained by applying all the mapping techniques to different datasets, each one having very different characteristics and structure.**

I. Introduction

Prefix-scan is a data-parallel operation whose broad importance is well known. Sequence compaction, radix sort, quick-sort, sparse-matrix vector multiplication, and minimum spanning tree construction are some of the many algorithms that can be efficiently implemented in terms of scan operations [1], [2]. These operations are the analogs of parallel prefix circuits [3], which have a long history, and have been widely used in collection-oriented languages dating since APL [4]. They also form the basis for efficiently mapping nested data-parallel languages such as NESL [5] on to flat data-parallel machines.

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 solutions have been presented for both array processor architectures [6], [2], [7] and GPUs [8], [9], [10], [11].

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 [12]. Given a workload to be allocated over the GPU threads (see Fig. 1), prefix-sum calculates the offsets to be used by the threads to access to the corresponding work-items (coarse-grained mapping) or work-units (fine-grained mapping) [13].

Even though prefix-scan operations allows the threads to efficiently access in parallel to the corresponding data, they do not address the load balancing problem. Indeed, the workload decomposition and mapping strategies are let to the application designer. How the application implements such a mappings can have a significant impact on the overall application performance.

Several different techniques have been presented in the literatures to decompose and map the workload to threads through the use of prefix-sum data structures [14], [15], [16], [13], [17], [18], [19]. 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 [14], [15] best apply to very regular workloads while they cause strong unbalancing and, as a consequence, lost of performance in case of irregular workloads. More complex solutions [16], [13], [17], [18], [19] 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 paper firstly presents an accurate analysis of all such load balancing techniques based on prefix-scan of the literature, by underlining advantages and drawbacks over different workload characteristics. Then, the paper presents an advanced dynamic technique, called *Multi-Phase Search*, which addresses the workload unbalancing problem by fully exploiting the GPU device characteristics. In particular, *Multi-Phase Search* implements a dynamic mapping of work-units to threads through an algorithm whose complexity is sensibly reduced with respect to the other dynamic approaches in the literature. This allows the proposed approach to provide good performance also when applied to very regular and balanced workloads.

The paper also provides a detailed analysis of the coalescing issue during the memory accesses both to the prefix-sum structure and to the global memory, which is strictly related to the mapping strategy implementation.

Finally, the paper presents the experimental results obtained by applying all the mapping techniques to different datasets,

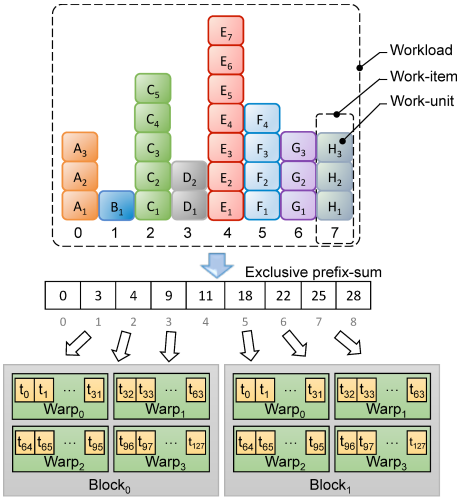


FIG. 1: *Overview of the load balancing problem in the workload decomposition and mapping to threads of scan-based applications*

each one having different sizes and characteristics.

The paper is organized as follows. Section II present the analysis of the related work. Section III presents the proposed dynamic technique. Section IV presents the experimental results and their analysis, while SectionV is devoted to the conclusions.

II. Related Work

In the literature, the techniques for decomposing and mapping a workload to threads based on prefix-scan 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[1].

A. Static mapping techniques

This class includes all the techniques that statically assign each work-item (or blocks 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 sensibly reduced 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.

1) Work-items to threads: It represents the simplest and fastest mapping approach by which each work-item is mapped to a single thread [14]. Fig. 2(a) shows an example, in which the eight items of Fig. 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 two levels 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. 2(a) are an

[1]The prefix-sum array is generated, depending on the mapping technique, in a preprocessing phase [20], at run-time if the workload changes at every iteration [13], [16], or it could be already part of the problem [21].

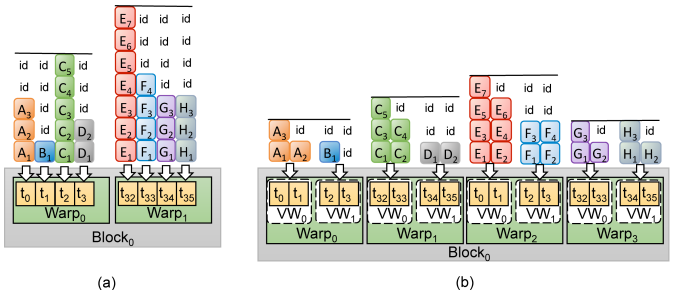


FIG. 2: *Example of static mapping techniques: (a) Work-items to threads, and (b) Virtual warps*

example. Then, irregular work-items lead to whole warps to be in idle state (e.g., $warp_0$ w.r.t. $warp_1$ in 2(a)). As a third level of unbalancing, this technique can lead to whole blocks of threads to be in idle state.

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.

2) Virtual Warps: 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 [15]. Fig. 2(b) 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. In addition, 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. 2(b)). 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. First, the maximum size of virtual warps is limited by the number of available threads in the device. Given the number of work-items and a virtual warp size, the required number of threads is expressed as follows:

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

If such a 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 sensibly 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.

Finally, 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:

```
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
```

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. 2(b)), 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).

B. 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.

1) Dynamic Virtual Warps + Dynamic Parallelism: This technique has been introduced in [16] and relies on two main strategies. First, 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 II-A2), 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 datasets representing scale free networks or graphs with power-law distribution in general).

On the other hand, dynamic parallelism, which exploits the most advanced features of the GPU architectures (e.g., from NVIDIA Kepler on) [22] 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 [16].

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, while it applies dynamic virtual warps to the others. 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 load balancing among threads of the same warps and among warps. It does not guarantee balancing among blocks.

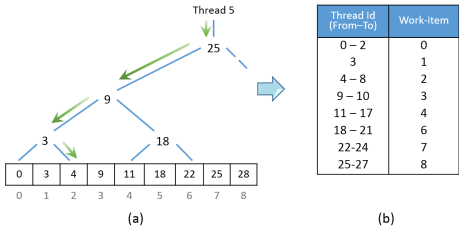
2) CTA+Warp+Scan: In the context of graph traversal, Merrill et al. [13] proposed an alternative approach to the load balancing problem. Their algorithm consists of three steps:

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:

```
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 then
6:          SharedStart = prefixsum[Th_ID]
7:          SharedEnd = prefixsum[Th_ID + 1]
8:      end
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
```

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 sensible 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



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. 3: *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).*

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 sensible overhead, which slows down the application performance in case of quite regular workloads. The strategy well applies only in case of very irregular workloads.

C. Dynamic mapping techniques

Contrary to *static mapping*, the *dynamic mapping* approaches achieve perfect workload partition and balancing among threads at the cost of additional computation 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.

1) 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. 3 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 prefix-sum array involves an arithmetic intensive computation (i.e., $\#threads \times binarysearch()$) and all the accesses performed by the threads to solve the mapping very scattered. This often eludes the benefit of the provided perfect balancing.

2) 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 $log_2 32$ 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 [16]. 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 [22].

In general, the local warp search strategy provides a very 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. Indeed, although this technique guarantees a fair balancing among threads of the same warp, it suffers from work unbalance 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.

3) Block Search: To deal with the local warp search limitations, Davidson et al. [17] 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 the 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. 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 in 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. This leads to several advantages. First, all the threads of a block are perfectly balanced. The binary searches are performed in shared memory and the overall amount of searches is sensibly 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 the assigned work-units. Finally, this strategy cannot exploit advanced features for intra-warp communication and synchronization among threads, such as, warp shuffle instructions etc.

4) Two-phase Search: Davidson et al.[17], Green et al [18] and Baxter [19] 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 the shared memory. As an example, considering blocks of 128 threads, two prefix-sum chunks pointing to $128 \times K$ units, and 1300 slots in shared memory, K is set to 10. The chunk size may differ among blocks (see for example Fig. 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

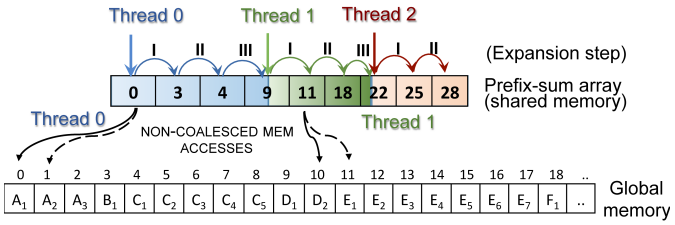


FIG. 4: *Example of expansion phase in the two-phase strategy (10 work-units per thread)*

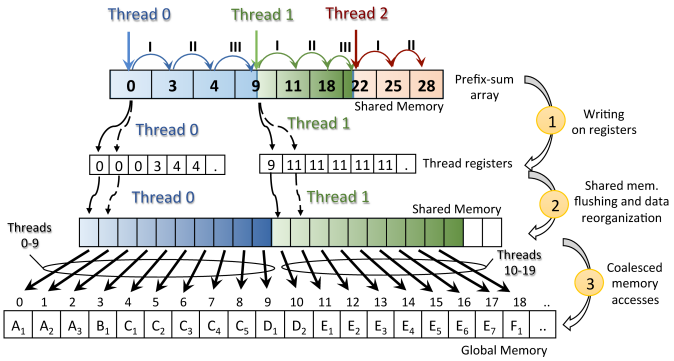


FIG. 5: *Overview of the coalesced expansion optimization (10 work-units per thread)*

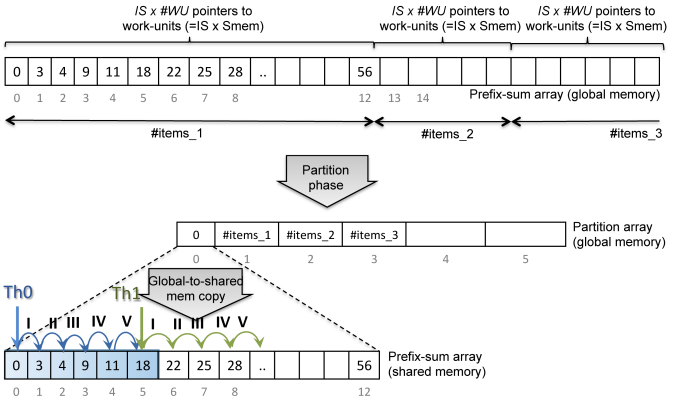


FIG. 6: *Overview of the iterated search optimization (10 work-units per thread and IS=2)*

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 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.

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 most 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 involves 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). Such a problem worsens by increasing the K value.

III. The Proposed Multi-phase Search Technique

The proposed multi-phase mapping strategy aims at exploiting the balancing advantages of the two-phase algorithms while overcoming the limitations concerning the scattered memory accesses. It consists of two main contributions: Coalesced expansion and Iterated search.

A. Coalesced Expansion

The proposed expansion phase consists of three 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. The technique applies for both reading and writing accesses to the global memory as for the two-phase approach.

For the sake of clarity, we consider *writing* accesses in the following.

1) 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 shows an example. The amount of units is limited by the available number of free registers.
2) 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) Finally, the whole warp of threads cooperates 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 are iterated until all the work-units assigned to the threads are processed. Even though these steps involve some extra computations with respect to the direct writings, the achieved coalesced accesses in global memory significantly improve the overall performance.

B. Iterated Searches

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 II-C4).

In particular, the overhead introduced to synchronize the threads after the writing on registers (see step 1 of coalesced expansion) is the bottleneck of the expansion phase (each register writing step requires two barriers of thread). The iterated search optimization aims at reducing such an overhead as follows:

1) In the partition phase, the prefix sum array is partitioned into balanced chunks (see Fig. 6). Differently from the two-phase search strategy, the size of such chunks is fixed as a multiple of the available space in shared memory:
$$Chunk_{size} = Block_{size} \times K \times IS$$
where $Block_{size} \times K$ 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. 6), i.e., it sequentially writes an amount of work-units in the local registers. Such an amount is equal 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 $Block_{size} \times K$. Then, the whole warp of threads cooperates for a coalesced transaction of the reordered data into the global memory, as in the coalesced expansion phase presented in Section III-A. This step iterates IS times, until all the data stored in the registers has been processed.

With respect to the standard partitioning and expansion strategy, the iterated search optimization reduces the number of synchronization points 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 more large 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 constraints 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.

IV. Experimental Results

A. Experimental Setup

We tested the load balancing efficiency of all the techniques presented in Section II and the proposed *Multi-Phase Search* over a dataset of different benchmarks. Table I summarizes the features of the dataset. It consists of four representative benchmarks selected from *The University of Florida Sparse Matrix Collection* [23], 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 four benchmarks have been selected among the collection to cover very different data characteristics in terms of 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. *Circuit5M* represents a circuit simulation instance, 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.

Workload Source	Max work-item size	Avg. work-item size	Std. Dev. work-item size
great-britain_osm	8	2.1	0.5
web-NotreDame	3,445	5.2	21.4
circuit5M	1,290,501	10.7	1,356.6
kron_g500-logn20	413,378	96.2	1,033.1

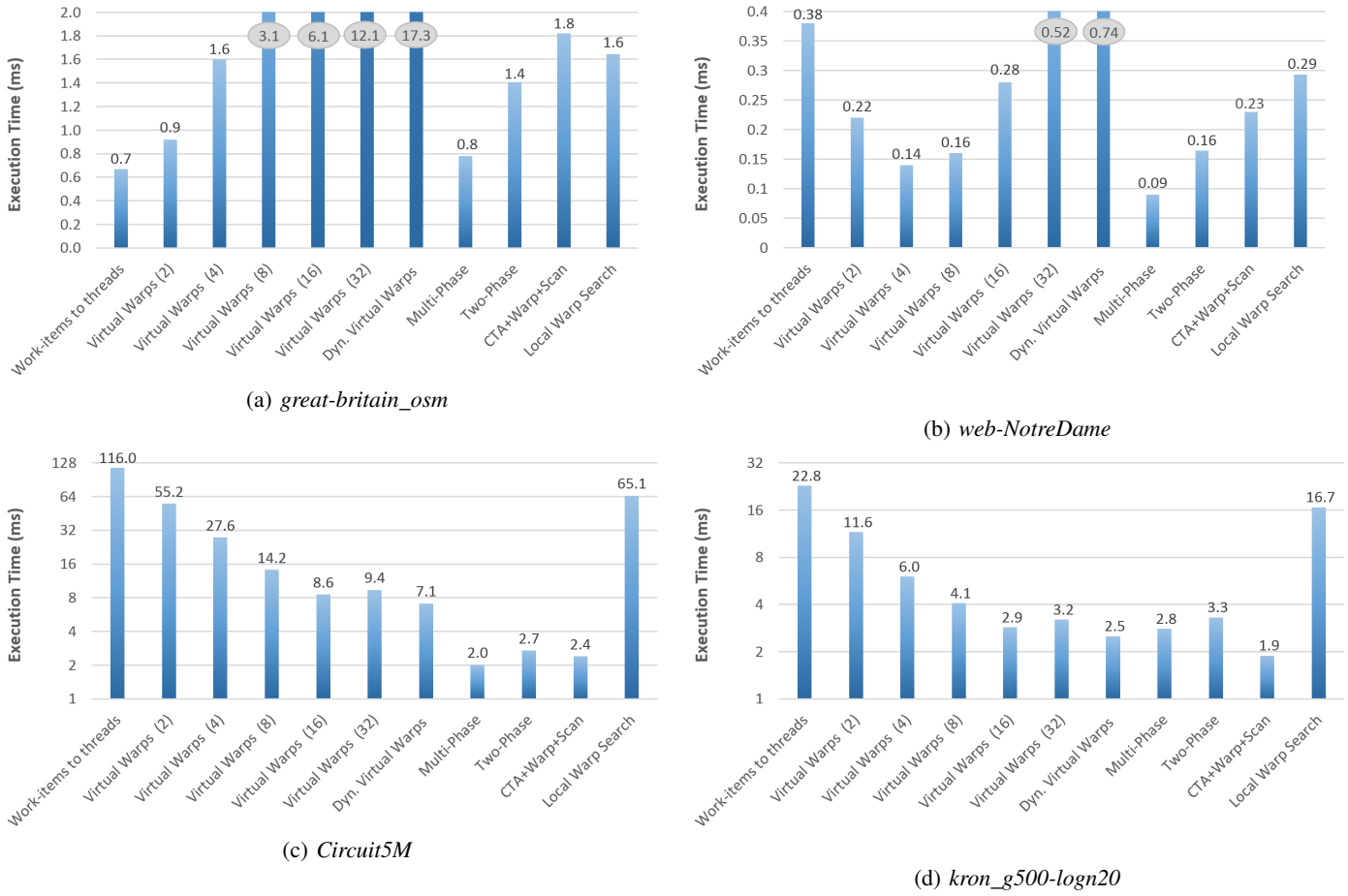
TABLE I: Dataset Characteristics

All the analysed 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 ran the experiments on a NVIDIA Kepler GeForce GTX 780 device with CUDA Toolkit 6.0, AMD Phenom II X6 1055T 3GHz host processor, and Debian 3.2.60 O.S.

B. Execution time analysis and Comparison

Figure 7 reports the obtained results in terms of execution time. In particular, the reported values are the best performance of each technique obtained by tuning the kernel configuration in terms of number of threads per block. For the GPU device used in this analysis, the best results have been reached with 128-256 threads per block for all the techniques **that allows the maximum occupancy of the device and low synchronization overhead.** The results obtained with the *Direct Search* and *Block Search* techniques are much 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 [19], which is based on the GPU algorithm proposed by Green et al [18].

In the first benchmark (Fig. 7a), as expected, *Work-items to threads* is the most efficient balancing technique. This is due to the very regular workload and the small average work-item size. In this benchmark, any overhead for the dynamic item-to-thread mapping may compromise the overall algorithm performance. However, the proposed *Multi-Phase Search* is the



(a) *great-britain_osm*

(b) *web-NotreDame*

(c) *Circuit5M*

(d) *kron_g500-logn20*

FIG. 7: *Comparison of execution time on the datasets.*

second most efficient technique. This underlines the reduced amount of overhead introduced by such a dynamic technique, which well applies also in case of very regular workloads.

In the *web-NotreDame* benchmark (Fig. 7b), *Multi-Phase Search* is the most efficient technique and provides almost twice the performance with respect to the second best techniques (*Virtual Warps* and *Two-Phase*). On the other hand, *Virtual Warps* provides good performance if the virtual warp size is properly set, while it may sensibly worsen 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.

Multi-Phase Search provides the best results also in the *circuit5M* benchmark (Fig. 7c). In such a benchmark, we observed that the *CTA+Warp+Scan*, *Two-Phase Search*, and *Multi-Phase Search* dynamic techniques are one order of magnitude faster than the static-mapping techniques. **In *web-Notredame* and in *circuit5M* *Multi-Phase Search* shows the best results due to the low average (less than warp size) and high std. deviation.** In the last benchmark, *kron_g500-logn20* (Fig. 7d), *CTA+Warp+Scan* provides the best results, since the CTA and Warp phases are frequently activated and exploited. However the performance of *Multi-Phase* are comparable. *Dynamic Virtual Warps* and *Virtual Warps* provide similar performance. Indeed, these two techniques are very efficient on high-average datasets, since, with a thread group size of 32, they completely avoid the warp divergence. Finally, we observed that the *Dynamic Parallelism* feature provided by Kepler, 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 the *Multi-Phase Search*, perform better without such a feature in all the analysed datasets.

C. Multi-Phase Search Analisys

Figure 8 shows the impact of the thread block size on the performance of the main phases of *Multi-Phase*. The *partition* phase takes advantage of large block sizes. 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,

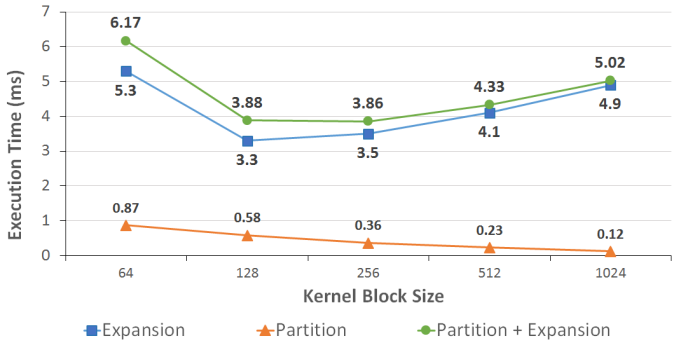


FIG. 8: *Execution time of Partition and Expansion phases varying the block size.* ***Executed on*** 2^{26} ***work-items with uniformly distributed random work-sizes.***

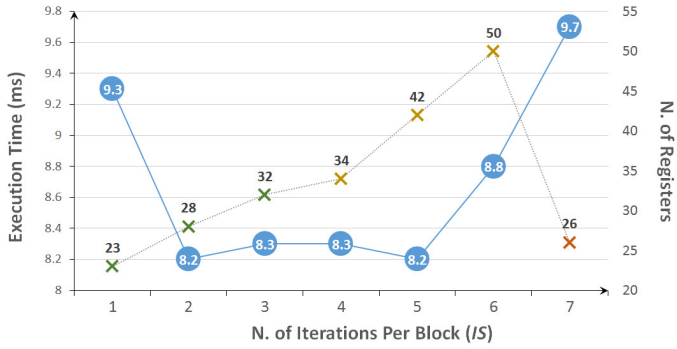


FIG. 9: *Execution time varying the number of iterations.* ***Executed on*** 2^{26} ***work-items with uniformly distributed random work-sizes.***

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 coordinate the shared memory accesses. We observed the best trade-off size of blocks equal to 256 (see Fig. 8).

Figure 9 reports the *Multi-Phase Search* 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 device 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, since all the data elaborated by the threads manly fits in registers and, in small part, in the L1 cache. With seven iterations and beyond, the performance drastically decreases since the compiler places the data variables outside the on-chip memory.

V. Conclusions

This paper presented an accurate analysis of the load balancing techniques based on prefix-scan in the literature, by underlining advantages and drawbacks over different workload characteristics. The paper then presented an advanced dynamic technique, called *Multi-Phase Search*, which addresses the workload unbalancing problem by fully exploiting the GPU device characteristics. In particular, the paper showed how *Multi-Phase Search* implements a dynamic mapping of work units to threads through an algorithm whose complexity is sensibly reduced with respect to the other dynamic approaches in the literature. This allows the proposed approach to provide good performance also when applied to very regular and balanced workload. The paper presented a set of experimental results to underline where and why the different static, semi-dynamic, and dynamic techniques find the best application.

References

[1] G. E. Blelloch, *Vector Models for Data-parallel Computing*. Cambridge, MA, USA: MIT Press, 1990.

[2] ——, "Scans as primitive parallel operations," *IEEE Trans. Comput.*, vol. 38, no. 11, pp. 1526–1538, 1989.

[3] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, pp. 831–838, 1980.

[4] K. E. Iverson, *A Programming Language*. New York, NY, USA: John Wiley & Sons, Inc., 1962.

[5] G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein, and M. Zagha, "Implementation of a portable nested data-parallel language," *SIGPLAN Not.*, vol. 28, no. 7, pp. 102–111, Jul. 1993.

[6] G. E. Blelloch, "Prefix sums and their applications," School of Computer Science, Carnegie Mellon Univ., Tech. Rep. CMU-CS-90-190, 1990.

[7] S. Chatterjee, G. E. Blelloch, and M. Zagha, "Scan primitives for vector computers," in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, 1990, pp. 666–675.

[8] M. Billeter, O. Olsson, and U. Assarsson, "Efficient stream compaction on wide simd many-core architectures," in *Proceedings of the Conference on High Performance Graphics 2009*, 2009, pp. 159–166.

[9] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli, "Fast scan algorithms on graphics processors," in *Proceedings of the 22nd Annual International Conference on Supercomputing*, ser. ICS '08, 2008, pp. 205–213.

[10] D. Merril and A. Grimshaw, "Parallel scan for stream architectures," Department of Computer Science, University of Virginia, Tech. Rep. CS-200914, 2009.

[11] S. Sengupta, M. Harris, and M. Garland, "Efficient parallel scan algorithm for GPUs," NVIDIA, Tech. Rep., 2009.

[12] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. MIT press, 2009.

[13] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, 2012, pp. 117–128.

[14] P. 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*, ser. HiPC'07, 2007, pp. 197–208.

[15] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11, 2011, pp. 267–276.

[16] F. Busato and N. Bombieri, "BFS-4K: an efficient implementation of BFS for kepler GPU architectures," *IEEE Transactions on Parallel Distributed Systems*, vol. preprint, no. 99, pp. 1–14, 2015.

[17] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel gpu methods for single-source shortest paths," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 349–359.

[18] O. Green, R. McColl, and D. A. Bader, "Gpu merge path: a gpu merging algorithm," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 331–340.

[19] "Modern gpu library." [Online]. Available: http://nvlabs.github.io/moderngpu/

[20] K. Xu, Y. Wang, F. Wang, Y. Liao, Q. Zhang, H. Li, and X. Zheng, "Neural decoding using a parallel sequential monte carlo method on point processes with ensemble effect," *BioMed research international*, vol. 2014, 2014.

[21] C. Yang, Y. Wang, and J. D. Owens, "Fast sparse matrix and sparse vector multiplication algorithm on the gpu," *IPDPSW*, 2015.
[22] NVIDIA, "Kepler GK110," www.nvidia.com/content/PDF/kepler/NV_DS_Tesla_KCompute_Arch_May_2012_LR.pdf.
[23] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.