A performance, power, and energy efficiency analysis of load balancing techniques for GPUs

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*—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 paper 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).**

I. 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 [1]. This makes them well suited to a variety of architectures, ranging from supercomputers to low-power and mobile devices [2], [3].

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 [4]. 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 [5].

Different techniques for GPU applications have been presented in literature to decompose and map the workload to threads [6], [7], [8], [9], [10], [11], [12]. 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 [6], [7] 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 [8], [9], [10], [11], [12] 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*[13] 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, 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 paper 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 paper is organized as follows. Section II presents some background on the load balancing problem in GPUs. Section III presents a concise survey of the load balancing techniques for GPUs. Section IV presents the load balancing analysis, while SectionV is devoted to the conclusions.

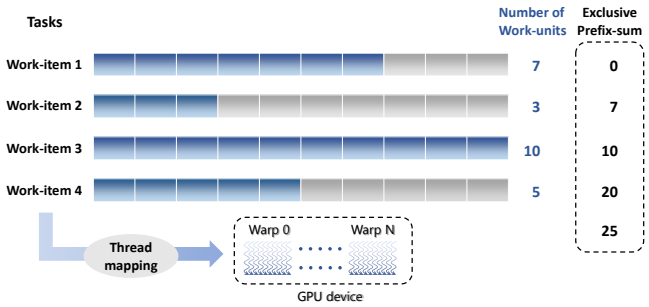


FIG. 1: *Example of work-items to threads mapping*

II. Background on load balancing in GPUs

Consider a workload to be partitioned and mapped to GPU threads. The workload consists of *work-units*, which are grouped into *work-items* (see Fig. 1). 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, and the work-items are the node neighbours of each node. The native mapping is implemented over work-items through the *prefix-sum* procedure [14]. 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 simplified example of Fig. 1 associated to the BFS, the neighbour analysis of four nodes is partitioned and mapped to four threads. 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 each work-item may have a variable number of work-units. In the example, the first thread would analyse seven neighbour nodes, the second three neighbour nodes, and so on.

III. A survey of load balancing techniques for GPUs

The problem of workload partitioning and mapping to threads (*mapping* in the following) in GPU applications has been deeply investigated in the last decade. The different mapping techniques proposed to deal with such an issue can be organized into three classes: *Static mapping* [6], [7], *semi-dynamic mapping* [9], [8], and *dynamic mapping* [11], [10], [12], [13]. They are all based on the prefix-sum array that, in this work, is assumed to be already generated. In general, the prefix-sum array is generated, depending on the mapping technique, in a preprocessing phase [15], at run-time if the workload changes at every iteration [9], [8], or it could be already part of the problem [16].

The simplest and more representative static technique statically assigns each work-item (or blocks of work-units) to a corresponding GPU thread [6]. It is suited to fairly well balanced and regular workload, since it does not involve computational overhead. A step towards more irregular datasets has been done by Hong et al. [7] with the concept of *virtual warp*. The approach consists of assigning chunks of work-units to groups of threads (i.e., virtual warps), where the virtual warps are equally sized and the threads of a virtual warp belong to the same warp. 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. It also improves the coalescing of memory accesses since more threads of a virtual warp access to adjacent addresses 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. Then, a wrong sizing of the virtual warps (which has to be set statically) can sensibly impact on the application performance. Finally, the technique provides good balancing among threads of the same warp, while it does not guarantee good balancing among different warps nor among different blocks.

Differently from the static ones, the semi-dynamic techniques include all the approaches by which different mapping configurations are calculated statically and, at run time, the application switches among them. Busato et al. [8] introduced the *Dynamic Virtual Warps* combined to *Dynamic Parallelism*. 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. The strategy switches at run time to dynamic parallelism when the workload consists of work-items having size greater than a threshold. It best applies to datasets 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.

Merrill et al. [9] proposed an alternative approach based on three steps (CTA+Warp+Scan). It provides a perfect balancing among threads and warps. Nevertheless, the first step (also called strip-mined gathering) 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.

Finally, the class of the dynamic techniques allow achieving perfect workload partition and balancing among threads even with particularly irregular workloads at the cost of additional computation at run time. The core of such a computation is the binary search over the prefix-sum array, which aims at mapping work-units to the corresponding threads. To reduce the binary search computation and the scattered accesses to the global memory, the technique in [8] first loads chunks of the prefix-sum array from the global to the GPU registers. Each chunk consists of 32 elements and is loaded by 32 warp threads through a coalesced memory access. Then, each thread of the warp performs a lightweight binary search over the corresponding chunk by using *Kepler* warp-shuffle instructions [17]. A similar dynamic approach has been proposed by Davidson et al. [10], who 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. Nevertheless, these dynamic strategies do not guarantee balancing among different blocks nor memory coalescing among threads when they access the assigned work-units.

Green et al [11] and Baxter [12] proposed equivalent methods to deal with the inter-block load unbalancing. 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. In the expansion phase, all the threads load the corresponding chunks into the shared memory. 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 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. 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.

Differently from the approaches in the literature, the more recent *Multi-phase* approach 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 literature. This allows *Multi-phase* to provide the best performance when handling irregular as well as regular and balanced workloads.

A. 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*.

1) 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* [18] 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** $S \geq Array[\text{left}]$ and $S \leq Array[\text{right}]$ **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:

$$Std\_Dev\_WI_{size} \leq Threshold_{SD}$$

and

$$Average\_WI_{size} \leq Threshold_{AVG}$$

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.

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 I: Benchmark Characteristics

2) 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 III-A1) 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.

IV. LOAD BALANCING ANALYSIS

A. 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 III) over a dataset of different benchmarks (see Table I). The dataset, consists of five representative benchmarks selected from *The University of Florida Sparse Matrix Collection* [19], 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.

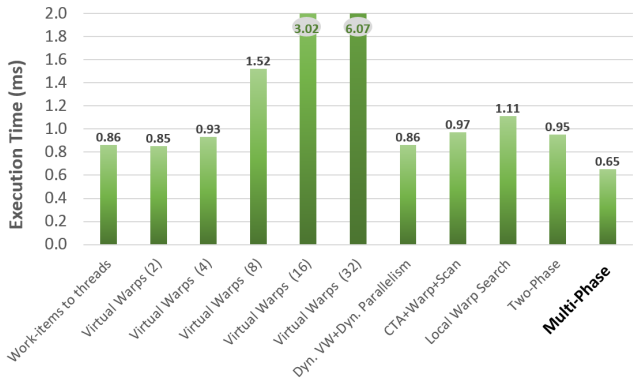
Performance information has been collected through the CUDA runtime API to measure the execution time and through the `clock64()` device instruction for throughput values to ensure clock-cycle accuracy of time measurements.

Power and energy consumption information have been collected through the *Powermon2* power monitoring device [20]. 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., *GPUBoost*). 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.

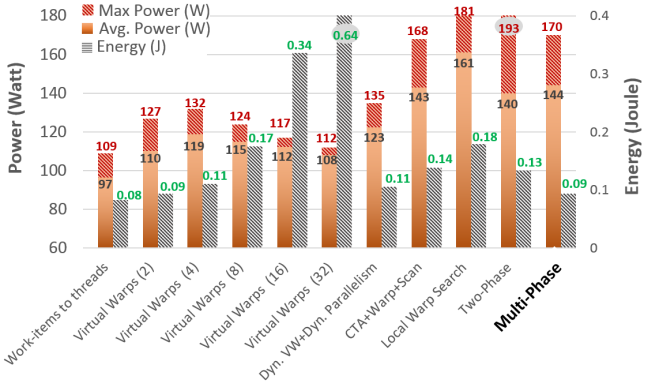
B. Performance, power, energy efficiency analysis and comparison

Figures 2, 3, 4, 5, and 6 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.

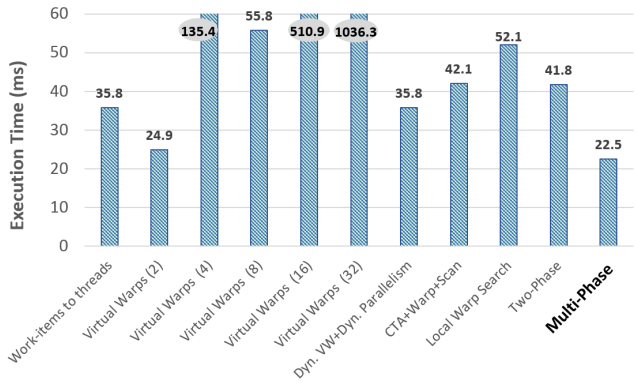
Work-item to threads [6] and *Virtual warps* [7] represent the static techniques (*Virtual warp* has been evaluated



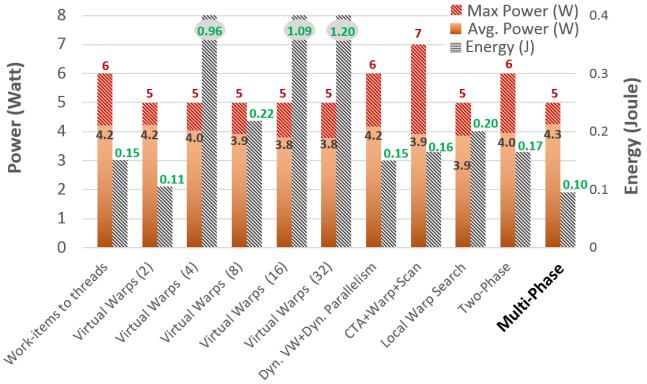
(a) *GTX980 - Execution Time*



(b) *GTX980 - Power and energy consumption*

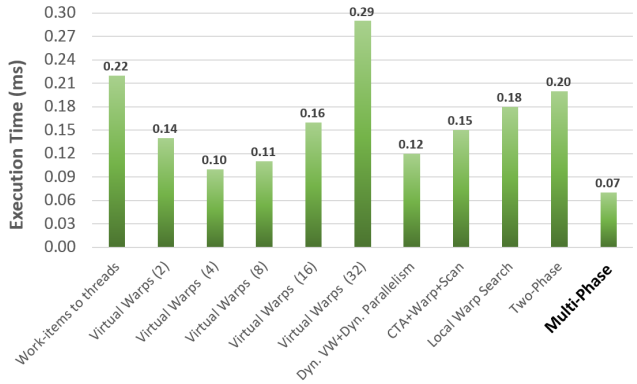


(c) *Jetson TK1 - Execution Time*

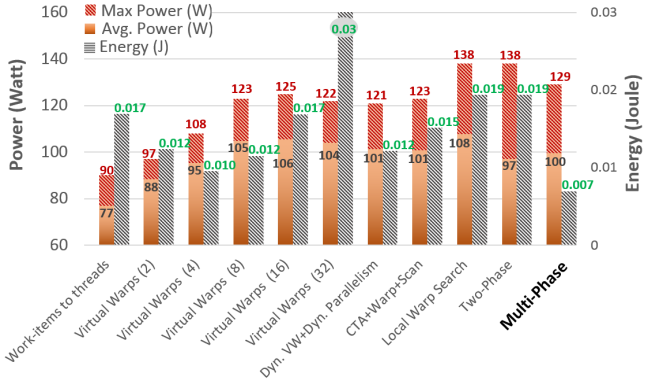


(d) *Jetson TK1 - Power and energy consumption*

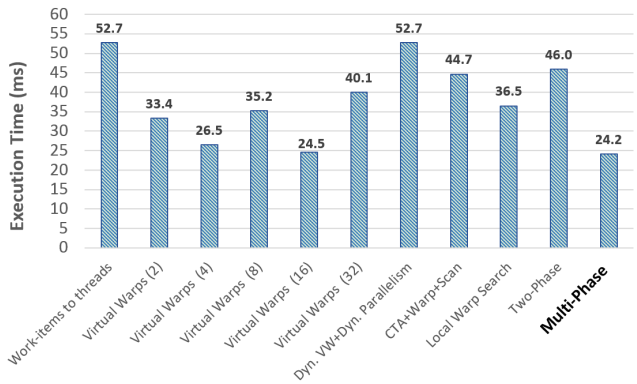
FIG. 2: *Comparison of execution time, power and energy consumption on great-britain_osm.*



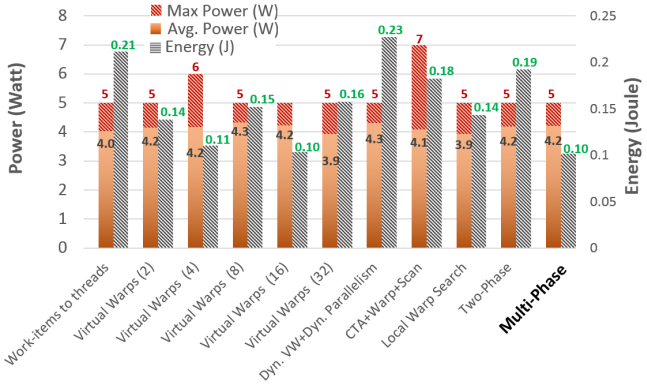
(a) *GTX980 - Execution Time*



(b) *GTX980 - Power and energy consumption*

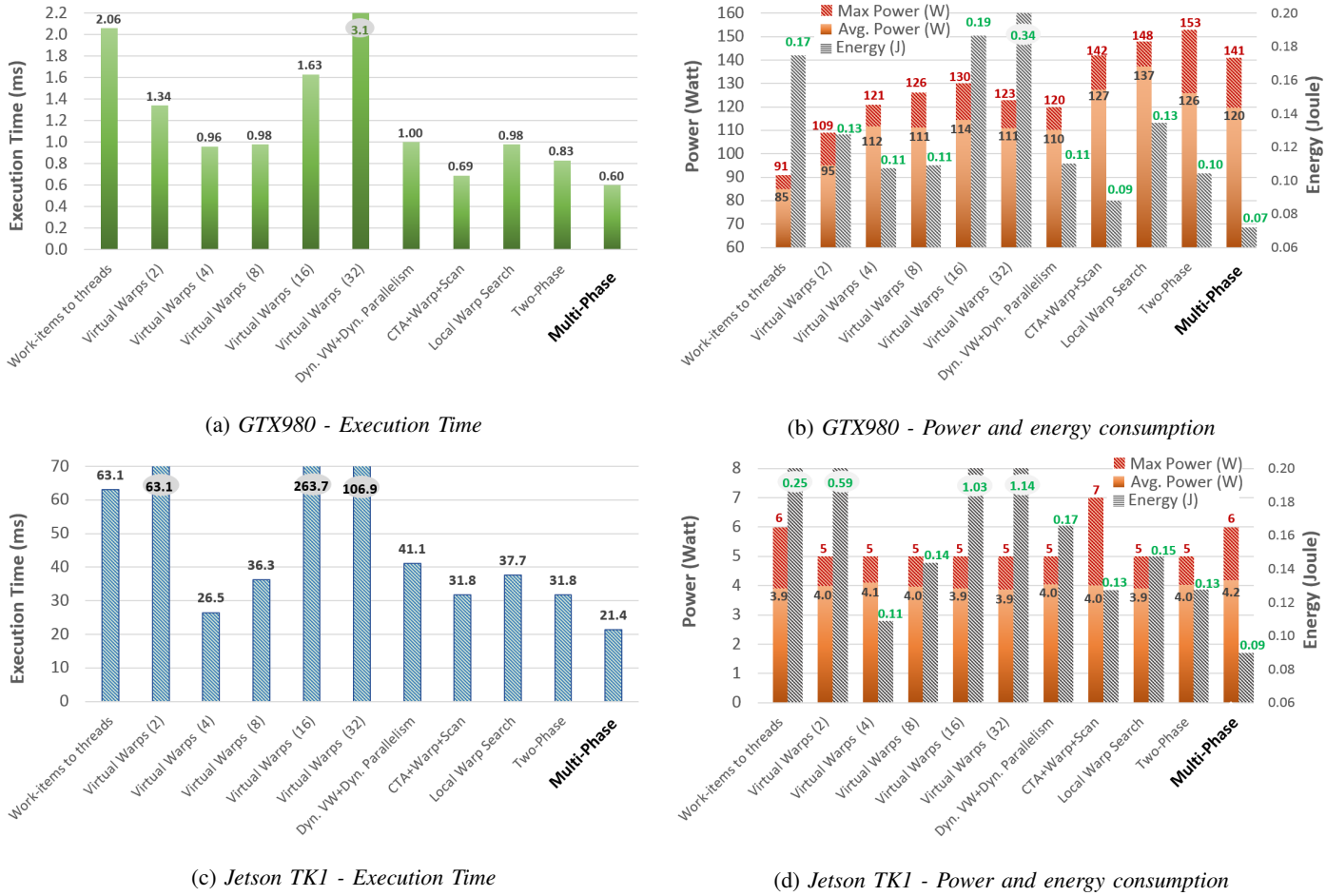


(c) *Jetson TK1 - Execution Time*



(d) *Jetson TK1 - Power and energy consumption*

FIG. 3: *Comparison of execution time, power and energy consumption on web-Notredame.*



(a) *GTX980 - Execution Time*

(b) *GTX980 - Power and energy consumption*

(c) *Jetson TK1 - Execution Time*

(d) *Jetson TK1 - Power and energy consumption*

FIG. 4: *Comparison of execution time, power and energy consumption on Cit-Patents.*

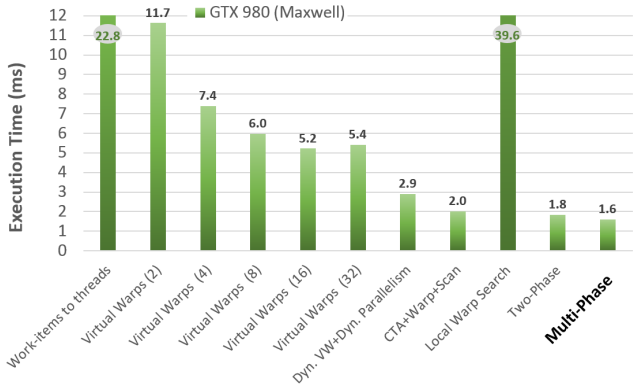
with different warp sizes). *Dyn.VW+Dyn.Parallelism* [8] and *CTA+Warp+Scan* [9] represent the semi-dynamic techniques, while *Local Warp Search* [10], *Two-Phase*, and *Multi-phase* represent the dynamic ones. For the *Two-Phase* algorithm, we used the well-know *ModernGPU* library [12], which is based on the GPU algorithm proposed by Green [11].

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 best performance in both the GPU architectures (see Figs. 2a and 2c). 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.

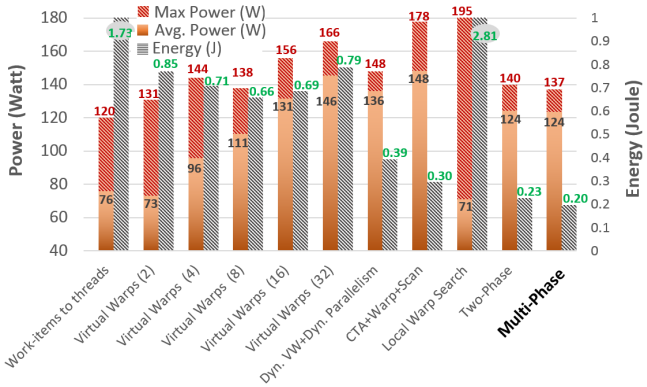
Figs. 2b and 2d report the average, maximum power and energy consumption of the load balancing applications for the same first benchmark. The static techniques show low average and maximum power on the GTX 980 device, while the semi-dynamic and dynamic techniques present the highest values, which are 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.

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. 3a, 3c). 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.

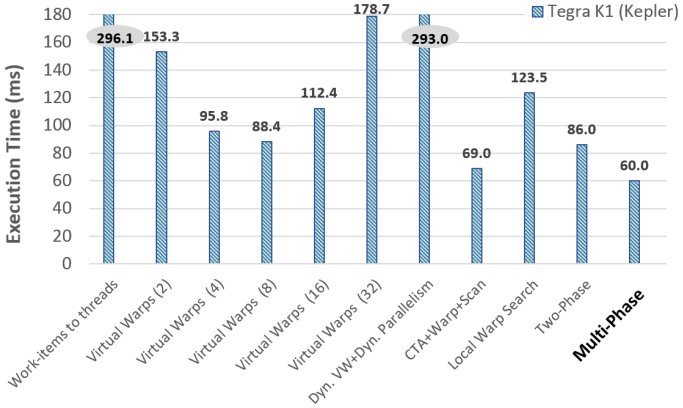
The power and energy consumption (Fig. 3b, 3d) follows



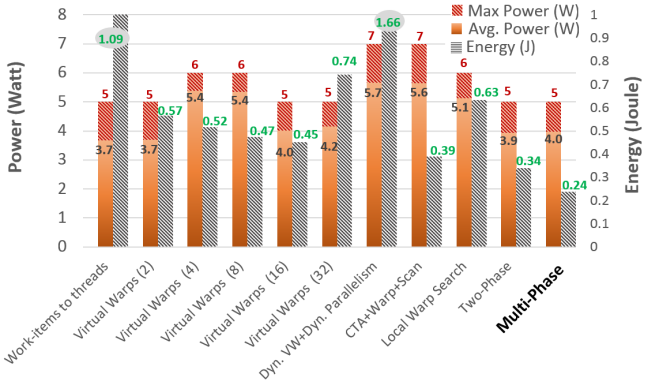
(a) *GTX980 - Execution Time*



(b) *GTX980 - Power and energy consumption*

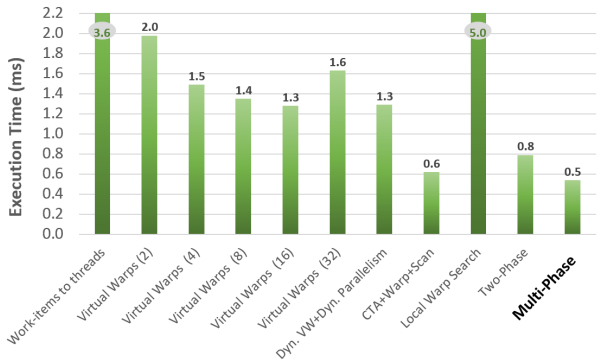


(c) *Jetson TK1 - Execution Time*

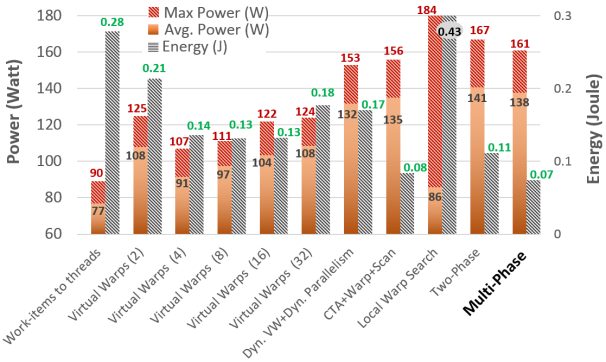


(d) *Jetson TK1 - Power and energy consumption*

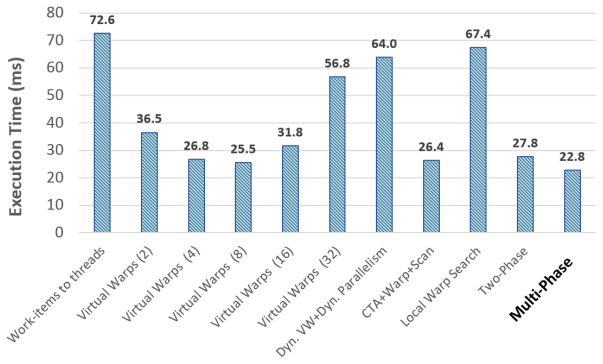
FIG. 5: *Comparison of execution time, power and energy consumption on Circuit5M.*



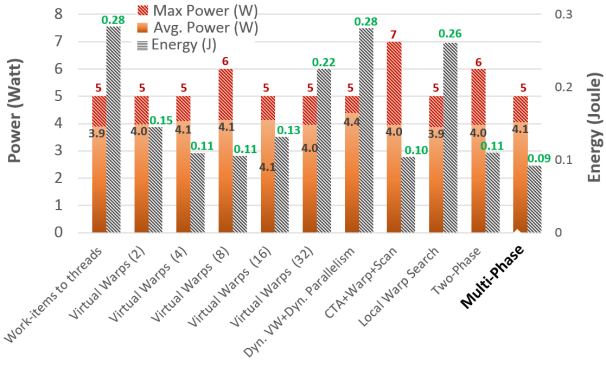
(a) *GTX980 - Execution Time*



(b) *GTX980 - Power and energy consumption*



(c) *Jetson TK1 - Execution Time*



(d) *Jetson TK1 - Power and energy consumption*

FIG. 6: *Comparison of execution time, power and energy consumption on Skitter.*

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. 4 and 5). In these benchmarks, we observed that the dynamic techniques (*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. 4b, 4d) and *circuit5M* benchmarks (Fig. 5b, 5d), *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. 6a, 6c), *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. 6b, 6d) 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).

V. Conclusions

This paper 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 paper 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.

References

[1] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving GPU energy efficiency," *ACM Comput. Surv.*, vol. 47, no. 2, pp. 19:1–19:23, Aug. 2014.

[2] "NVIDIA Tegra X1," http://www.nvidia.com/object/tegra.html.

[3] "Qualcomm Snapdragon," http://www.qualcomm.com/products/snapdragon.

[4] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao, "Dynamic load balancing on single- and multi-gpu systems," 2010, pp. 1–12.

[5] S. Hong and H. Kim, "An integrated gpu power and performance model," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10, 2010, pp. 280–289.

[6] 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.

[7] 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.

[8] F. Busato and N. Bombieri, "BFS-4K: an efficient implementation of BFS for kepler GPU architectures," *IEEE Transactions on Parallel Distributed Systems*, vol. 26, no. 7, pp. 1826–1838, 2015.

[9] 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.

[10] 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.

[11] 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.

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

[13] F. Busato and N. Bombieri, "A dynamic approach for workload partitioning on gpu architectures," *IEEE Transactions on Parallel Distributed Systems*, vol. preprint, no. 99, pp. 1–15, 2016.

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

[15] 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.

[16] C. Yang, Y. Wang, and J. D. Owens, "Fast sparse matrix and sparse vector multiplication algorithm on the gpu," *IPDPSW*, 2015.

[17] NVIDIA, "Kepler GK110," www.nvidia.com/content/PDF/kepler/NV_DS_Tesla_KCompute_Arch_May_2012_LR.pdf.

[18] Y. Perl, A. Itai, and H. Avni, "Interpolation search—a log log n search," *Communications of the ACM*, vol. 21, no. 7, pp. 550–553, 1978.

[19] 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.

[20] D. Bedard, M. Y. Lim, R. Fowler, and A. Porterfield, "Powermon: Fine-grained and integrated power monitoring for commodity computer systems," in *Proc. of IEEE SoutheastCon*, 2010, pp. 479–484.