A dynamic approach for workload partitioning on GPU architectures

Federico Busato, Nicola Bombieri, *Member, IEEE,*

Abstract—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 tradeoff 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 article 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 article 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 article 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.

Index Terms—Computer Society, IEEE, IEEEtran, journal, prefix-scan, load balacing, GPU.

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 [1], [2]. 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 [3], [4], [5], [6]. 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 [7].

Given a workload to be allocated over the GPU threads (see Fig. 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) [8]. 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 [8], [9], [10], [11], [12], [13], [14]. 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 [9], [10] 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], [11], [12], [13], [14] 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.

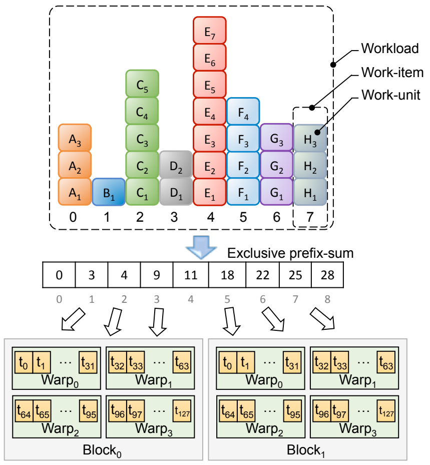


FIG. 1: *Overview of the load balancing problem in the workload partitioning and mapping to threads of scan-based applications*

This article 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.

• *Federico Busato and Nicola Bombieri are with the Department of Computer Science, University of Verona, Italy, email: {name.surname@univr.it}.*

Then the article 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 article presents the experimental results obtained by applying all the analysed 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 article is organized as follows. Section 2 presents the analysis of the related work. Section 4 presents the proposed multi-phase mapping technique. Section 5 presents the experimental results and their analysis, while Section 6 is devoted to the conclusions.

2 Background and Related Work

2.1 The workload partitioning problem in GPUs

Consider a workload to be partitioned and mapped to GPU threads (see Fig. 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, and the work-items are the node neighbours 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. 1 associated to the BFS, the neighbour analysis of eight nodes is partitioned and mapped to eight threads. t_0 elaborates the neighbours of node *0* (work-units *A*), t_1 elaborates the neighbours 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 [15], at run-time if the workload changes at every iteration [8], [11], or it could be already part of the problem [16]).

2.2 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.

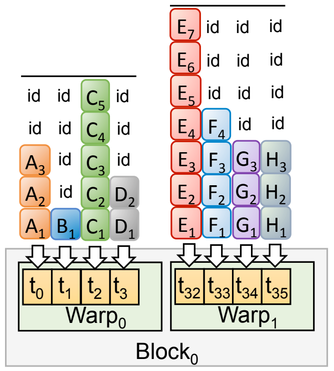


FIG. 2: *Example of work-items to threads mapping*

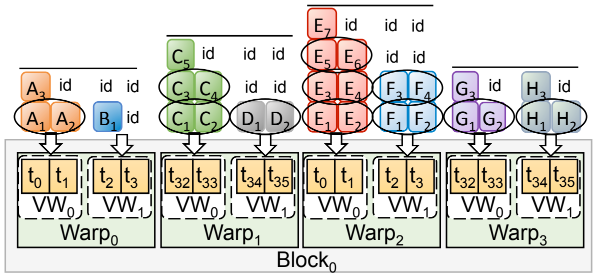


FIG. 3: *Example of Virtual warps work-units mapping (black circles represent coalesced memory accesses)*

2.2.1 *Work-items to threads*

It represents the simplest mapping approach by which each work-item is mapped to a single thread [9]. Fig. 2 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 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. 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 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.

2.2.2 *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 [10]. Fig. 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. 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:

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

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:

```
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
```

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. 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).

2.3 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.

2.3.1 Dynamic Virtual Warps + Dynamic Parallelism

This technique has been introduced in the work [11] 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 2.2.2), 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).

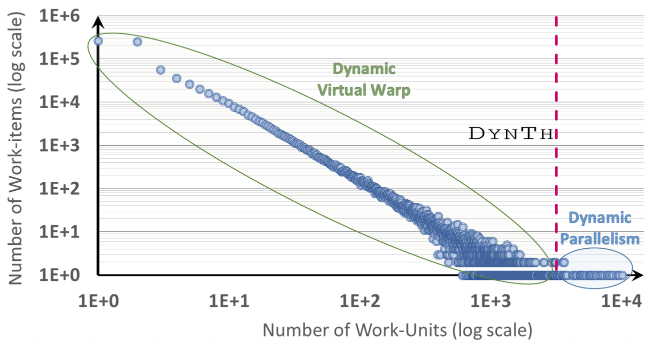


FIG. 4: *Example of Dynamic Virtual Warps + Dynamic Parallelism work-units mapping where the dynamic parallelism is applied to a subset of the workload with a power-law distribution*

On the other hand, dynamic parallelism, which exploits the most advanced features of the GPU architectures (e.g., from NVIDIA Kepler on) [17] 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 [11] [18].

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 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.

2.3.2 CTA+Warp+Scan

In the context of graph traversal, Merrill et al. [8] 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-*

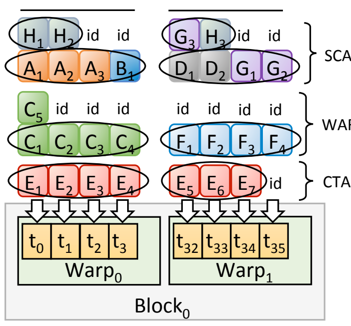
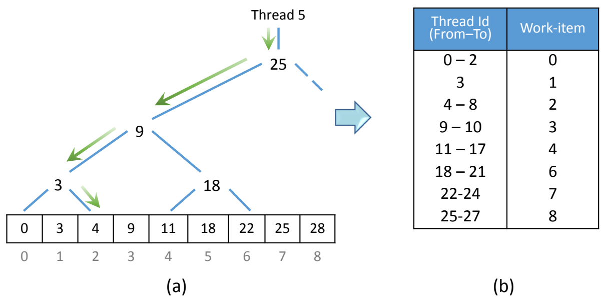


FIG. 5: *Example of CTA+Warp+Scan work-units mapping (black circles represent coalesced memory accesses)*



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. 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).*

mined gathering) is run by each thread (TH_{ID}). It can be summarized as follows:

```
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 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 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. 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.

2.4 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.

2.4.1 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. 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.

2.4.2 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 $\log_2(WarpSize)$ 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 [11]. 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 [17].

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.

2.4.3 Block Search

To deal with the local warp search limitations, Davidson et al. [12] 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. 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.

2.4.4 Two-phase Search

Davidson et al. [12], Green et al [13] and Baxter [14] 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. 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.

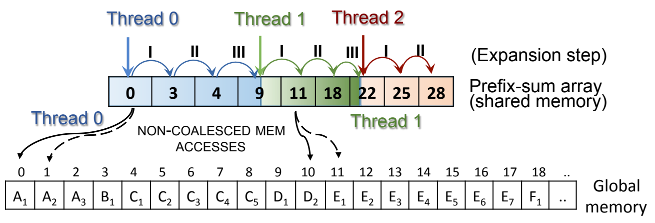


FIG. 7: *Example of expansion phase in the two-phase strategy (10 work-units per thread)*

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. 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.

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. 7). Such a problem worsens by increasing the K value.

3 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*.

3.1 Hybrid partitioning

Differently from the dynamic techniques in literature, which strongly rely on the binary search (see Section 2.4), 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.

3.1.1 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. 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.

3.1.2 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* [19] 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 $Array[\text{left}], Array[\text{right}]$. 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, we 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:

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

3.2 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 3.1.1) in such a local partition to get the assigned work-unit.

Then, the expansion phase consists of three iterative subphases, 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. 8 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.

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 2.4.4).

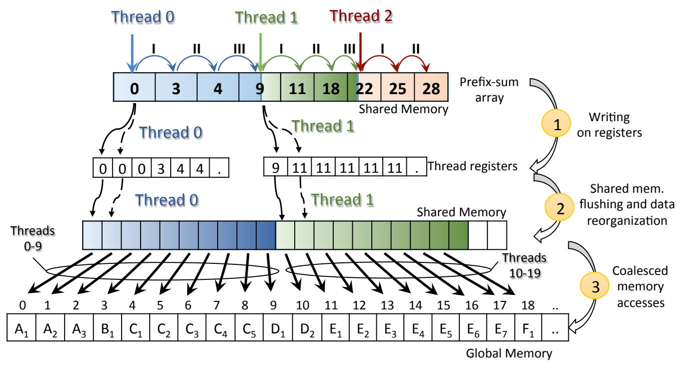


FIG. 8: *Overview of the coalesced expansion optimization (10 work-units per thread)*

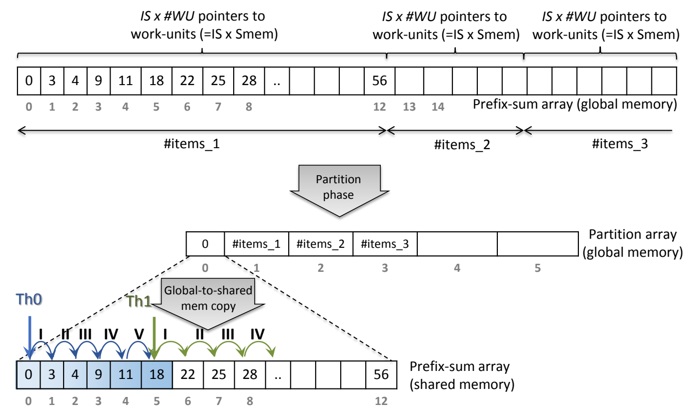

IS x #WU pointers to work-units (=IS x Smem)
Prefix-sum array (global memory)
#items_1
#items_2
#items_3
Partition phase
Partition array (global memory)
Global-to-shared mem copy
Th0
Th1
Prefix-sum array (shared memory)

FIG. 9: *Overview of the iterated search optimization (10 work-units per thread and IS=2)*

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. 9). 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. 9), 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 3.2. 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.

3.3 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.

3.3.1 *Full loop unrolling and intruction-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 [20].

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.

3.3.2 *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. 10.

3.3.3 *Global data prefetching.*

Data movement particularly affects the expansion phase of the proposed algorithm. We optimized the global-to-shared memory and shared-to-global memory data movement by introducing an additional intermediate step between the accesses to these memory spaces, as proposed in [21]. We

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. 10: *Example of data and pointer hoisting.*

exploited the thread registers as fast intermediate local memory, thus hiding the memory access latency.

3.3.4 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.

3.3.5 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.

3.3.6 Scheduler overhead minimization.

Implementing the proposed partitioning and mapping approach requires a kernel structure similar to that shown in Fig. 11. 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 3.2). 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 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

KERNEL IMPLEMENTATION (DEVICE SIDE):

```
1:     $var = 0$                              \\ Initialization
2:     thread_offset = ...                    \\ Initialization
3:     __shared__ SMem                        \\ Initialization
4:     ...                                    \\ Initialization
5:     for (i = blockIdx.x; i < $\lceil \frac{WorkLoadSize}{ChunkSize} \rceil$; i += gridDim.x) do
6:         /* Computational phase
7:         through SMem accesses
8:         */
9:         __synchthreads()
10:    end
```

KERNEL CONFIGURATIONS (HOST SIDE):

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

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

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

FIG. 11: *Kernel structure and configurations.*

32, provided the best scheduler overhead minimization in all the analysed benchmarks.

3.3.7 Read-only cache and pointer aliasing.

Recent architectures introduce the read-only data cache [22], [23]. 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.

4 COMPREHENSIVE COMPARISON OF COMPLEXITY AND LIMITING FACTORS OF THE APPROACHES

To accurately compare *Multi-phase Mapping* and the existing counterparts, we present an overall overview of the performance limiting factors and the complexity analysis of each approach core algorithm. Table 1 summarizes the main performance limiting factors ordered by relevance for each technique and 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 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

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 Dyn. Parallelism	Non-coalesced memory accesses, warp divergence, block scheduling overhead Dynamic kernels overhead
CTA+WARP+SCAN	CTA Warp Scan	Synchronization overhead \ 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 Expansion	Non-coalesced memory accesses, warp divergence Available shared memory
MULTI-PHASE MAPPING	Partition - Binary Search Partition - Interpolation Search Expansion	Non-coalesced memory accesses, warp divergence Compute intensive Available shared memory

TABLE 1: Summary of the performance limiting-factors.

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	$O(W)$	$O\left(\frac{\text{Dyn}_{\text{Th}}}{\lvert\text{VirtualWarp}\rvert}\right)$	$N + \sum_{W_i \geq \text{Dyn}_{\text{TH}}} W_i$	Yes	Partial/Yes
CTA+WARP+SCAN	$O(W)$	$\max_i \begin{cases} W_i \geq \text{CTA}_{\text{TH}} & \frac{W_i}{\lvert\text{CTA}\rvert} \\ W_i \geq \text{Warp}_{\text{TH}} & \frac{W_i}{\lvert\text{Warp}\rvert} \\ \text{otherwise} & W_i \end{cases}$	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
TWO-PHASE SEARCH	$O\left(\frac{N}{\text{SMem}} \cdot \log \text{N}\right) +$ $O\left(\frac{N \cdot \lvert Block\rvert}{\text{SMem}} \cdot \log \text{SMem} + \text{W}\right)$	$O\left(\log \text{N}\right) +$ $O\left(\log \text{SMem} + \frac{\text{W}}{\text{N}}\right)$	$\frac{W}{\text{SMem}} + \frac{N \cdot \lvert Block\rvert}{\text{SMem}}$	Yes	No
MULTI-PHASE MAPPING	$O\left(\frac{N}{\text{SMem} \cdot \text{IS}} \cdot \log \text{N}\right) +$ $O\left(\frac{N \cdot \lvert Block\rvert}{\text{SMem}} \cdot \log \text{SMem} + \text{W}\right)$	$O\left(\log \text{N}\right) +$ $O\left(\log \text{SMem} + \frac{\text{W}}{\text{N}}\right)$	$\frac{W}{\text{SMem} \cdot \text{IS}} + \frac{N \cdot \lvert Block\rvert}{\text{SMem} \cdot \text{IS}}$	Yes	Yes

TABLE 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.

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 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 2 reports the results. All the static and semi-dynamic techniques are *work-efficient*, as they achieve the same work complexity of the sequential algorithm. 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 3.2), *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 EXPERIMENTAL RESULTS

We tested the load balancing efficiency of all the techniques presented in Section 2 and *Multi-phase Mapping* over different benchmarks, whose characteristics are reported in Table 3. The benchmarks have been selected from *The University of Florida Sparse Matrix Collection* [24], 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 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.

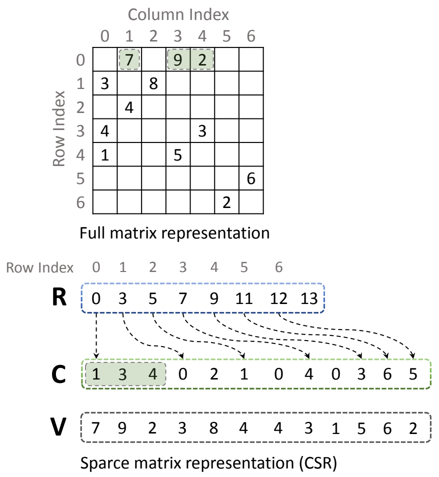

Column Index
0 1 2 3 4 5 6
Row Index
Full matrix representation
R 0 3 5 7 9 11 12 13
C 1 3 4 0 2 1 0 4 0 3 6 5
V 7 9 2 3 8 4 4 3 1 5 6 2
Sparce matrix representation (CSR)

FIG. 12: *The CSR data structure.*

In our 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 [25], [26], [27], [28]. 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. 12 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

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 3: Benchmark Characteristics

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 4: GPU Characteristics

not-uniform distribution, while they have low and high average, respectively.

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 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 13, 14, and 15 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 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 [14] developed by NVIDIA Research, which is based on the merge path algorithm proposed by Green et al. [29]. 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).

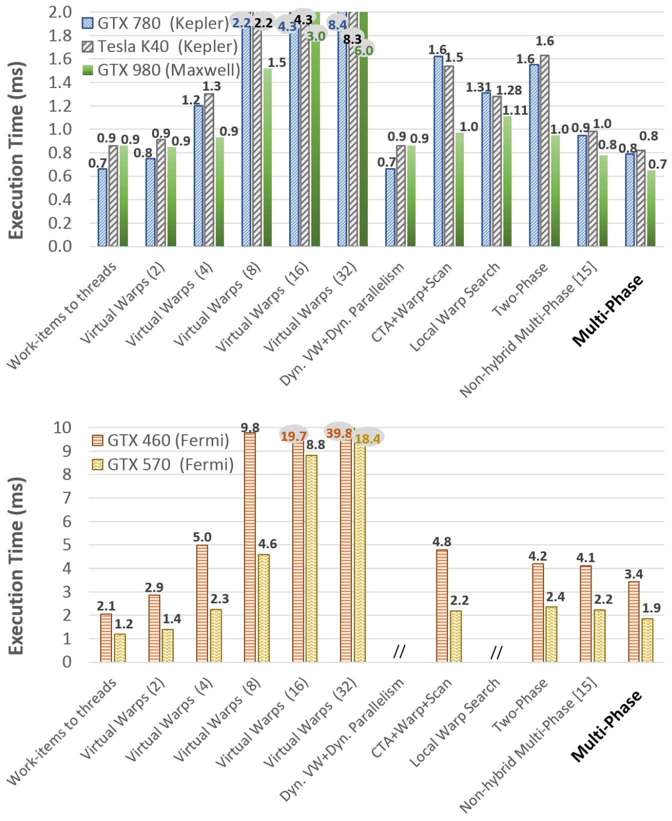
In the first benchmark (Fig. 13a), as expected, the static techniques are the most efficient. This is due to the very regular workload and to the low average work-item size. The semi-dynamic *Dyn. VW + Dyn. Parallelism* performs well since the dynamic parallelism feature is always switched off in such 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 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. 13b) 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.

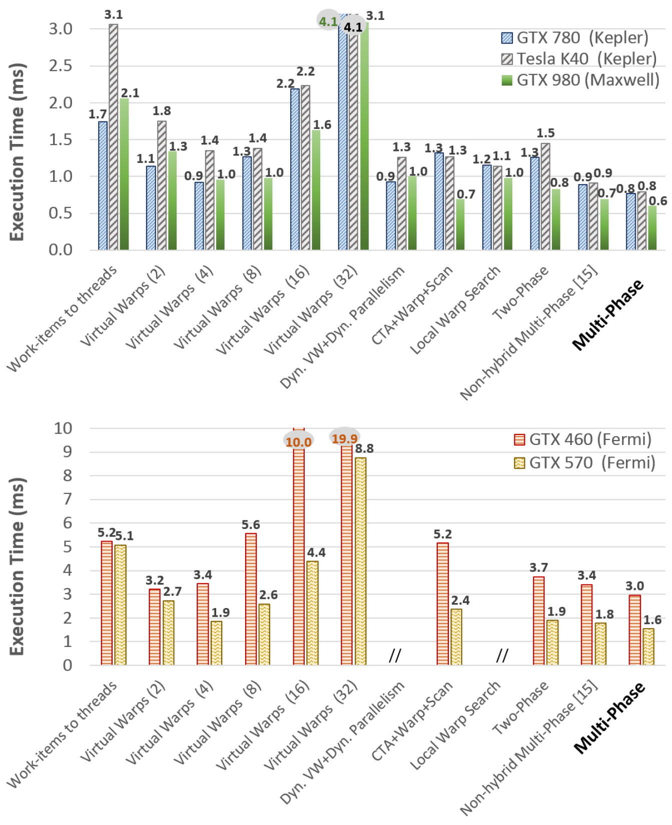
In *web-NotreDame* (Fig. 14a), *Multi-phase Mapping* is the most efficient technique and provides almost twice the performance with respect to the second best technique (*Virtual Warps*). It is three times faster than the other dynamic mapping techniques (*Local Warp Search* and *Two-Phase*) on all the 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.

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. 14b). In such a benchmark, we observed that the *CTA+Warp+Scan*, *Two-Phase Search*, and

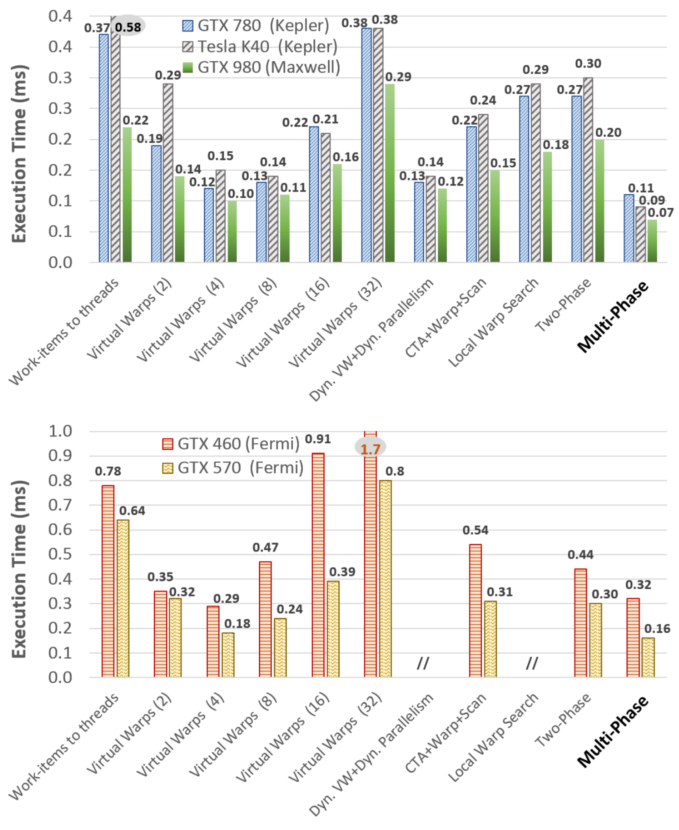


(a) *great-britain_osm*

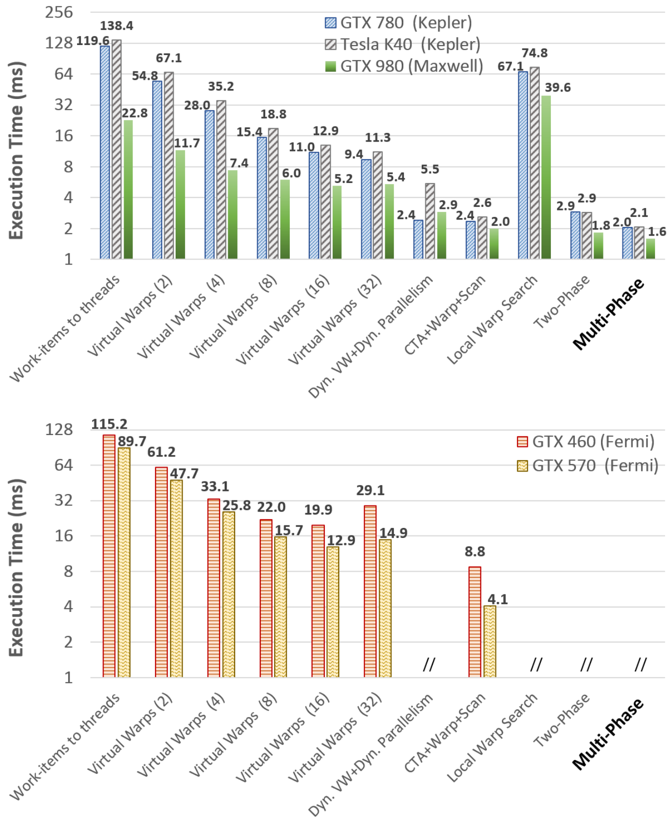


(b) *cit-Patent*

FIG. 13: *Comparison of execution time on the benchmarks.*

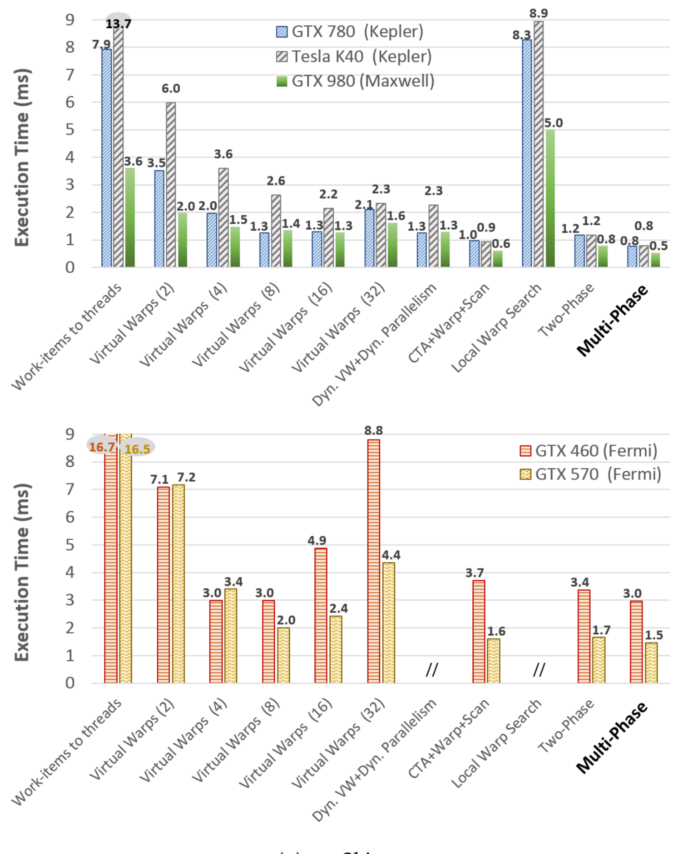


(a) *web-NotreDame*

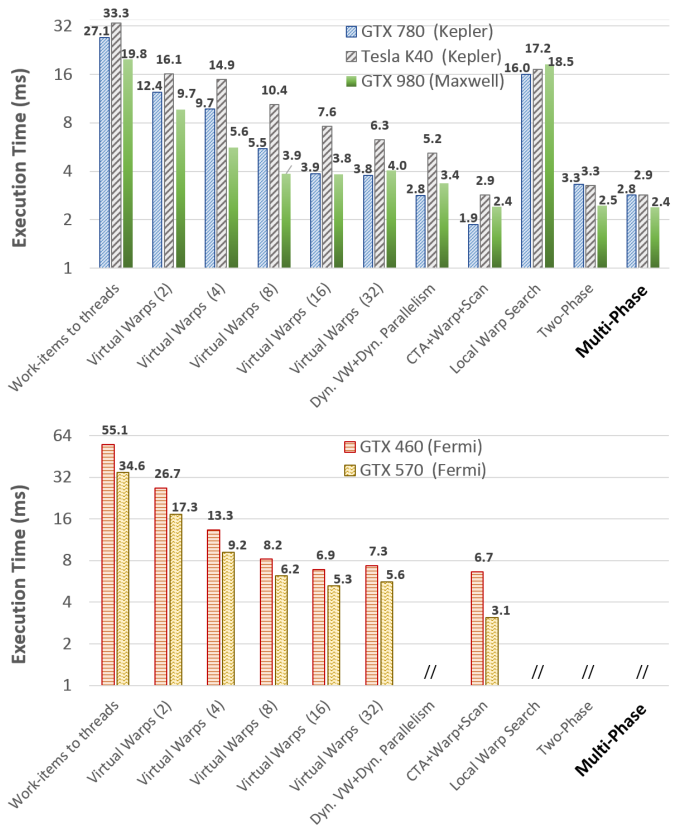


(b) *circuit5M*

FIG. 14: *Comparison of execution time on the benchmarks.*



(a) *as-Skitter*



(b) *kron_g500-logn20*

FIG. 15: *Comparison of execution time on the benchmarks.*

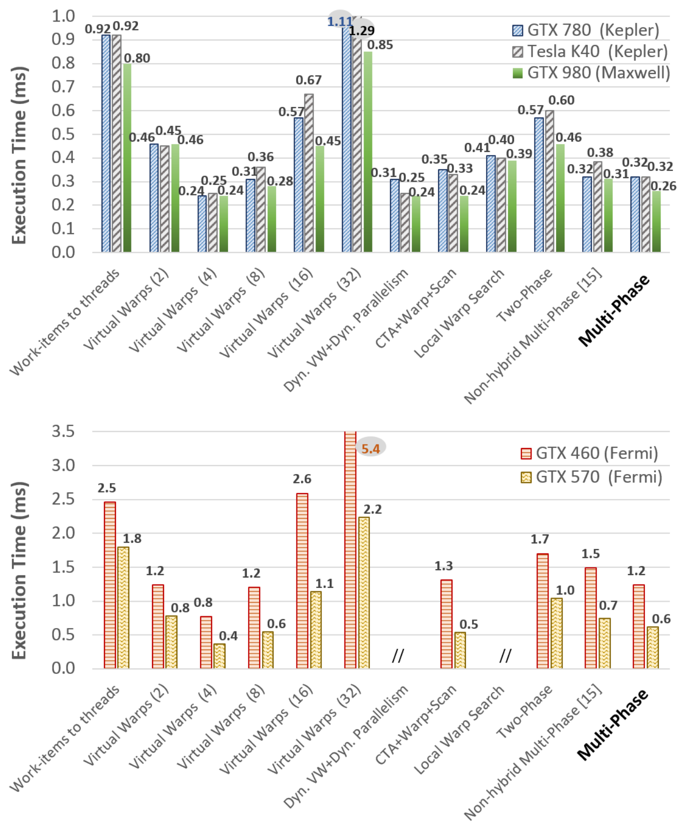


FIG. 16: *Comparison of execution time on the regular8 benchmark.*

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. 15a) and *kron_g500-logn20* (Fig. 15b), *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. 16), 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

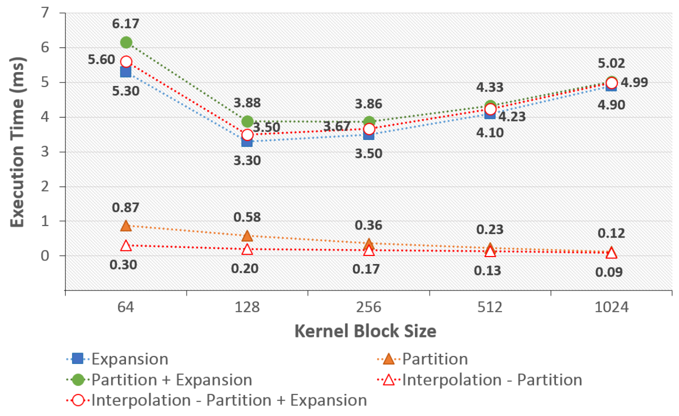


FIG. 17: *Execution time of Partition and Expansion phases by varying the block size (on 2^{26} items with unif. distributed random work-sizes).*

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).

5.1 *Multi-phase Mapping* Analisys

Fig. 17 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 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. 17).

Fig. 18 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

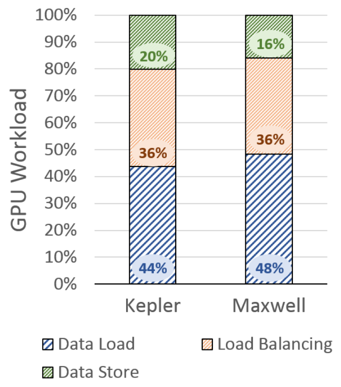


FIG. 18: *GPU workload breakdown of Multi-phase algorithm on Kepler and Maxwell Architectures*

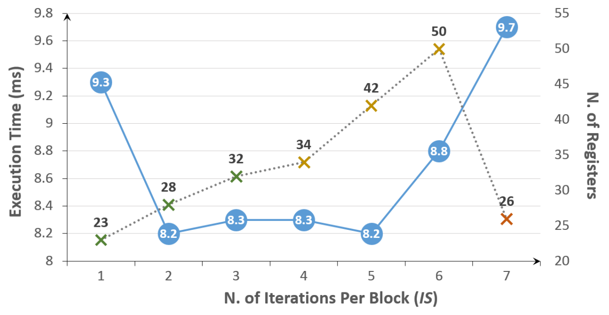


FIG. 19: *Execution time by varying the number of iterations (on 2^{26} work-items with uniformly distributed random work-item sizes).*

optimizations presented in Section 3.3, in particular, loop unrolling and vectorized memory accesses.

Fig. 19 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.

6 Conclusions

This article 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 article then presented an advanced dynamic technique, called *Multi-phase Mapping*, which addresses the workload unbalancing problem by fully exploiting the GPU device characteristics. The paper 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 article 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.

References

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

[2] ——, "Scans as Primitive Parallel Operations," *IEEE Transactions on computers*, vol. 38, no. 11, pp. 1526–1538, 1989.

[3] M. Billeter, O. Olsson, and U. Assarsson, "Efficient Stream Compaction on Wide SIMD Many-core Architectures," in *Proceedings of the ACM Conference on High Performance Graphics 2009*, 2009, pp. 159–166.

[4] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli, "Fast Scan Algorithms on Graphics Processors," in *Proceedings of the 22nd ACM Annual International Conference on Supercomputing*, ser. ICS '08, 2008, pp. 205–213.

[5] D. Merrill and A. Grimshaw, "Parallel Scan for Stream Architectures," Department of Computer Science, University of Virginia, Tech. Rep. CS-200914, 2009.

[6] S. Sengupta, M. Harris, M. Garland, and J. D. Owens, "Scientific Computing with Multicore and Accelerators," 2011.

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

[8] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU Graph Traversal," in *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming*, vol. 47, no. 8, 2012, pp. 117–128.

[9] P. Harish and P. J. Narayanan, "Accelerating Large Graph Algorithms on the GPU Using CUDA," in *Proceedings of the 14th IEEE International Conference on High Performance Computing*, ser. HiPC'07, 2007, pp. 197–208.

[10] 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.

[11] 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.

[12] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths," in *Proceedings of IEEE Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, 2014, pp. 349–359.

[13] 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*, 2012, pp. 331–340.

[14] S. Baxter, "Modern gpu library." [Online]. Available: http://nvlabs.github.io/moderngpu/

[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*, 2014.

[16] C. Yang, Y. Wang, and J. D. Owens, "Fast Sparse Matrix and Sparse Vector Multiplication Algorithm on the GPU," *Proceedings of IEEE Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, pp. 841–847, 2015.

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

[18] F. Busato and N. Bombieri, "An Efficient Implementation of the Bellman-Ford Algorithm for Kepler GPU Architectures," *IEEE Transactions on Parallel Distributed Systems*, vol. 27, no. 8, pp. 2222–2233, 2016.

[19] 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.

[20] V. Volkov, "Better Performance at Lower Occupancy," in *Proceedings of the GPU Technology Conference, GTC*, vol. 10, 2010, p. 16.

[21] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization," in *Proceedings of ACM international conference for high performance computing, networking, storage and analysis*, 2011, p. Art. n. 12.

[22] NVIDIA Corporation, "Kepler Tuning Guide."

[23] ——, "CUDA C programming guide v7.5."

[24] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[25] N. Bell and M. Garland, "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors," in *Proceedings of the ACM Conference on High Performance Computing Networking, Storage and Analysis*, 2009, p. Art. n.18.

[26] D. Langr and P. Tvrdik, "Evaluation Criteria for Sparse Matrix Storage Formats," *IEEE Transactions on Parallel Distributed Systems*, vol. 27, no. 2, pp. 428–440, 2016.

[27] Y. Saad, *Iterative methods for sparse linear systems*. Siam, 2003.

[28] 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*, vol. 37, no. 6, pp. C643–C666, 2015.

[29] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk, "Merge Path - Parallel Merging Made Simple," in *Proceedings of IEEE Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, 2012, pp. 1611–1618.

Federico Busato received the Master degree in Computer Science from the University of Verona in 2014. Currently he is a Ph.D. student at the University of Verona, Department of Computer Science. His research activity focuses on high performance computing and graph theory.

Nicola Bombieri received the PhD in Computer Science from the University of Verona in 2008. Since 2008, he is researcher and Professor Assistant at the Dept. of Computer Science of the University of Verona. His research activity focuses on high performance computing, design and verification of embedded systems, and automatic generation and optimization of embedded SW. He has been involved in several national and international research projects and has published more than 80 papers on conference proceedings and journals.