BFS-4K: an Efficient Implementation of BFS for Kepler GPU Architectures

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

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

Index Terms—Parallel graph algorithms, CUDA, GPU, BFS, Kepler

1 Introduction

Graphs are a common representation in many problem domains, including engineering, finance, medicine, and scientific applications. Breadth-first search (BFS) is a crucial graph traversal algorithm used by many graph-processing applications. Different problems, such as VLSI chip layout, phylogeny reconstruction, data mining, and network analysis, map to very large graphs, often involving millions of vertices. Even though very efficient *sequential* implementations of BFS exist [1]–[3], they have work complexity of the order of number of vertices and edges. As a consequence, such sequential implementations become impractical when applied on very large graphs.

Recently, graphics processing units (GPUs) have become widespread platforms as they provide massive parallelism at low cost. Parallel executions on GPUs may achieve speedup up to three orders of magnitude with respect to the sequential counterparts on CPUs. Nevertheless, accelerating efficient and optimized sequential algorithms and porting (i.e., parallelizing) their implementation to such many-core architectures is a very challenging task. Several solutions in literature take advantage of the massive parallelism of GPUs [4]–[8] but they are often asymptotically less efficient than the fastest CPU implementations [3]. After a certain graph size and, thus, for graph sizes typical of many actual problem domains, the parallel implementations for GPUs become slower than the sequential implementations for CPUs.

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

Thread divergence, workload imbalance, and poorly coalesced memory accesses are the most representative issues that come up when traversing a graph with a parallel implementation. In particular, sparse graphs, scale free networks or graphs with power-law distribution in general show up the limits of the parallel implementations suffering from these problems [9]–[12].

On the other hand, GPU vendors continue to innovate and meet that demand for high performance parallel computing with extremely powerful GPU computing architectures (e.g., NVIDIAs new Kepler GK110 [13]). The most recent architectures, not only offer much higher processing power than the prior GPU generations, but, also, they provide new programming capability to improve the efficiency of the parallel implementations.

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

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

The article is organized as follows. Section 2 presents some preliminary concepts on CUDA, Kepler, and the BFS algorithm. Section 3 presents a detailed analysis of the main representative implementations of parallel BFS for GPUs at the state of the art. Section 4 gives an overview of the proposed approach while the single techniques implemented in *BFS-4K* are detailed in Section 5. Section 6 presents the problem of duplicates and the proposed approach to deal with them. Finally, Section 7 presents the experimental results by underlying the single technique contributions in the overall visit performance and a comparison of *BFS-4K* with the BFS implementations for GPU at the state of the art. Section 8 is devoted to concluding remarks.

2 Background

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

2.1 CUDA and Kepler

Compute Unified Device Architecture (CUDA) is a C library extension developed by NVIDIA to provide a programming interface to GPU devices [14]. The *host* CPU is responsible for starting the main program and executing serial code, while delegating parallel execution of compute-intensive tasks to the GPU *device*. The CUDA programming requires the definition of C functions, called *kernels*, which are executed in parallel by multiple GPU *threads*. The threads run the same kernel concurrently, and each one is associated with a unique thread ID. A kernel is executed by a three-dimensional *grid* of thread *blocks*. Threads are arranged into three-dimensional thread blocks. Threads of the same block efficiently cooperate by sharing data through fast shared memory and by synchronizing their execution through extremely fast (i.e., HW implemented) barriers. In contrast, threads belonging to different blocks are not allowed (for performance reasons) to perform barrier synchronizations with each other.

Thread blocks are then subdivided into groups of 32 threads called *warps* to be physically executed by GPU cores. A thread warp (or warp) executes one common instruction at a time, meaning that multiple threads within the warp execute the same instruction on different data at the same time (i.e., SIMD architecture). To achieve full efficiency, all threads within a warp should follow the same control flow path. If threads in the same warp follow different paths, they are said to *diverge*. In case of *branch divergence*, the warp serially executes each branch. Threads that are not on the branch being currently taken are disabled with a consequent performance decrease.

In 2012, NVIDIA released the *Kepler GK110* architecture [13], which introduces many improvements and new features to better support parallelism in a wider application range. One of the most relevant features is *dynamic parallelism*, which allows the application execution to be controlled by the GPU (besides the CPU). This includes the support of program *recursion* and dynamic workload balancing, that is, handling not uniformly distributed data, such as unbalanced graphs, by creating additional threads during a single kernel execution and avoiding overhead due to many kernel invocations. Nevertheless, dynamic parallelism can also lead to performance decrease if used inappropriately. This work presents an analysis of the dynamic parallelism in BFS visits, by proposing a parametric use of it to correctly exploit its potentiality.

Warp shuffle instructions are another new feature of Kepler used in the proposed solution. They implement very efficient communication of threads within a warp. With shuffle instructions, threads within a warp can directly access other thread registers by skipping shared memory accesses. In addition, thread communication via warp shuffle allows the amount of shared memory required for blocks to be reduced with consequent general improvements of performance.

The Kepler architecture also introduces the *8-byte access mode* to the shared memory. The shared memory throughput is doubled by increasing the bank width to 8 bytes. The proposed algorithm implementation takes advantage of this feature to realize a hash table and an efficient technique for atomic and coalesced accesses.

2.2 Breadth First Search (BFS)

BFS is one of the most import graph algorithms. It is used in several different contexts such as image processing, state space searching, network analysis, graph partitioning, and automatic theorem proving. Given a graph $G(V, E)$, where V is the set of vertices and E is the set of edges, and a source vertex s, the BFS visit inspects every edge of E to find the minimum number of edges or the shortest path to reach every vertex of V from source s. The traditional sequential algorithm [3] can be summarized as follows:

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

	Harish [4]	Virtual Warps [5]	Edge Parallelism [7]	Luo [15]	Garland [16]	*BFS-4K*
Work complexity	$O(VD+E)$	$O(VD+E)$	$O(ED)$	$O(V+E)$	$O(V+E)$	$O(V+E)$
Space complexity	$O(3V+E)$	$O(2V+E)$	$O(2E)$	N/A	$\Omega(4V+2E)$	$\Omega(4V+E)$
Type of parallelismn	Vertices	Virtual Warp	Edges	Vertices	Vertices, Edges, CTA	Vertices, Edges, *Dynamic* Virtual Warp, Dynamic Parallelism
High-degree vertex management	no	yes	indifferent	no	yes	yes
Duplicate detection	no	no	no	no	yes	yes
Type of synchronization	Host-Device	Host-Device	Host-Device	Host-Device, Inter-block [17], Thread barriers	Host-Device Inter-block [17]	Host-Device, Inter-block [17], Thread barriers

Fig. 1. Comparison of the most representative BFS implementations at the state of the art with *BFS-4K*

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

3 Related work

Harish and Narayanan [4] proposed the first approach to accelerate BFS on GPUs. The proposed algorithm explores all the graph vertices at each iteration (i.e., at each visiting level) to see whether the vertex belongs to the current frontier. This allows the algorithm to save GPU overhead by not maintaining the frontier queues. Nevertheless, the proposed approach leads to a sensible workload imbalance whenever the graph is non homogeneous in terms of vertex degree. In addition, let D be the graph diameter, the computational complexity of such a solution is $O(VD+E)$, where $O(VD)$ is spent to check the frontier vertices and $O(E)$ is spent to explore each graph edge. While this approach fits on dense graphs, in the worst case of sparse graphs (where $D=O(V)$) the algorithm has a complexity of $O(V^2)$. This implies that, for large graphs, the proposed algorithm is slower than the sequential version of the algorithm (see Section 2.2).

A partial solution to the problem of workload imbalance has been proposed in [5]. Instead of assigning a thread to a vertex, the authors propose thread groups (which they call *virtual warps*) to explore the array of vertices. The group size is typically 2, 4, 8, 16, or 32, and the number of blocks is inversely proportional to the virtual warp size. This leads to a limited speedup in case of low degree graphs, since many threads cannot be exploited at the kernel configuration time. Also, the virtual warp size is static and has to be properly set depending on each graph characteristics.

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

[7] and [8] present alternative approaches based on edge parallelism. Instead of assigning one or more threads to a vertex, the thread computation is distributed to edges. As a consequence, the thread divergence is limited and the workload is balanced even with high-degree graphs. The main drawbacks is the overhead introduced by the visit of all graph edges at each level. In many cases, the number of edges is much greater than the number of vertices. In these cases, the parallel work is not sufficient to improve the performance against vertex parallelism.

An efficient BFS implementation with computational complexity $O(V+E)$ is proposed in [15]. The algorithm exploits a single hierarchical queue shared across all thread blocks and an inter-block synchronization [17] to save queue accesses in global memory. Nevertheless, the small frontier size requested to avoid global memory writes and the visit exclusively based on vertex parallelism limit the overall speedup.

Merrill, Garland and Grimshaw [16] present an algorithm that achieves work complexity O(V+E). They make use of parallel prefix-scan and three different approaches to deal with the workload imbalance: vertex expansion and edge contraction, edge contraction and vertex expansion, and hybrid. The algorithm also relies on a technique to reduce redundant work due to *duplicate* vertices on the frontiers.

This article presents *BFS-4K*, a parallel BFS implementation that achieves work complexity $O(V+E)$. Differently from all the approaches in literature, *BFS-4K* implements:

- A two-level exclusive prefix-sum to efficiently manage the frontier propagation steps.
- A dynamic virtual warps whereby the warp size is

calibrated at each frontier propagation step.
- The dynamic parallelism, which allows the main GPU kernel to properly configure and invoke a *child kernel* for overcoming the workload imbalance due to the different degrees of vertices.
- An edge-discover technique based on binary search, in which threads are assigned to edges rather than vertices through an efficient binary search procedure.
- Two main GPU kernels, which are alternately used and combined with the features presented above during frontier propagation.
- A duplicate detection and correction strategy, which is based on hash table and 8-bank access mode to sensibly reduce the memory accesses and improve the detection capability.
- A technique to induce coalescence in the global memory accesses, which is combined with prefix-sum and adopted by the visiting techniques listed above.

In particular, the proposed implementation exploits the features of the Kepler architecture such as dynamic parallelism, warp-shuffle, and 8-bank access mode, to guarantee an efficient implementation of the characteristics listed above. Figure 1 summarizes the differences between the most representative BFS implementations at the state of the art and *BFS-4K*.

4 BFS4 Overview

Given a graph $G(V, E)$ and a source vertex s, *BFS-4K* exploits the concept of *frontier* [3] to achieve work efficiency $O(V+E)$ for the parallel BFS visits of G. The tool generates a breadth-first tree that has root s and contains all reachable vertices. The vertices in each *level* of the tree compose a *frontier (F)*. *Frontier propagation* checks every neighbour of a frontier vertex to see whether it is visited already. If not, the neighbour is added into a new frontier.

BFS-4K implements the frontier propagation through two data structures, Fd and Fd_{new}. Fd represents the actual frontier, which is read by the parallel threads to start the propagation step. Fd_{new} is written by the threads to generate the frontier for the next BFS step. At each step, Fd_{new} is filtered and swapped into Fd for the next iteration. Figure 2 shows an example, in which starting from vertex "0", the BFS visit concludes in three steps[1].

The filtering steps aims at guaranteeing correctness of the BFS visit as well as avoiding useless thread work and waste of resources. When a thread visits a neighbour already visited, that neighbour is eliminated from the frontier (e.g., vertex 2 visited by a thread from vertex 3 in step two of Figure 2). When more threads visit the same neighbour in the same propagation step (e.g., vertex 8 visited by threads 2 and 3 in step two), they generate *duplicate* vertices in the frontier. Duplicate vertices cause redundant work in the subsequent propagation steps (i.e., more threads visit the same path) and useless occupancy of shared memory. *BFS-4K* implements a duplicate detection and correction strategy based on hash tables, Kepler 8-byte memory access mode, and warp shuffle instructions, as explained in Section 6.

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

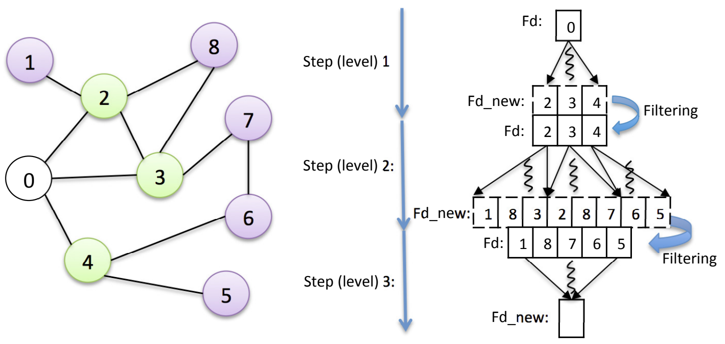


Fig. 2. Example of BFS visit starting from vertex "0"

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

- *Exclusive prefix-Sum.* To improve data access time and thread concurrency during the propagation steps, the frontier data structures are stored in shared memory and handled by a *prefix-sum* procedure. Such a procedure is implemented through warp shuffle instructions of the Kepler architecture, as explained in Section 5.1.
- *Dynamic virtual warps.* The *virtual warp* technique presented in [5] is applied to minimize the waste of GPU resources and to reduce the divergence during the neighbour inspection phase. Differently from [5], this work proposes a strategy to dynamically calibrate the warp size at each frontier propagation step, as explained in Section 5.2.
- *Dynamic parallelism.* In case of vertices with degree much greater than the average, (e.g., scale free networks or graphs with power-law distribution in general), *BFS-4K* applies the dynamic parallelism provided by the Kepler architecture instead of virtual warps. Dynamic parallelism implies an overhead that, if not properly used, may worse the algorithm performance. *BFS-4K* checks, at run time, the characteristics of the frontier to decide whether and how applying this technique, as explained in Section 5.3.
- *Edge-Discover.* With the edge-discover technique, threads are assigned to edges rather than vertices to improve the thread workload balancing during

frontier propagation. The edge-discover technique makes intense use of warp shuffle instructions. *BFS-4K* checks, at each propagation step, the frontier configuration to apply this technique rather than dynamic virtual warps, as explained in details in Section 5.4.

- *Single-block vs. Multi-block kernel. BFS-4K* relies on a two-kernel implementation. The two kernels are alternately used and combined with the features presented above during frontier propagation. Section 5.5 presents an analysis of the two-kernel features and explains how they are applied to better exploit the GPU stream multiprocessor properties.
- *Coalesced read/write memory accesses*. To reduce the overhead caused by the many accesses in global memory, *BFS-4K* implements a technique to induce coalescence among warp threads through warp shuffle, as explained in Section 5.6..

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

5 Implementation Features in Details

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

5.1 Exclusive Prefix-Sum

Given a list of input values and a binary associative operator, a *prefix-scan* procedure computes an output list of elements in which each element is the reduction of the elements occurring earlier in the input list. Prefix-scan has been largely investigated in the past years and several solutions have been presented for both array processor architectures [18]–[20] and GPUs [21]–[24].

When the operator is the addition, the prefix-scan represents a *prefix-sum*. Prefix-sum is useful when parallel threads must allocate dynamic data within shared data structures such as global queues. Given a total amount of data to be allocate for each thread, prefix-sum calculates the offsets to be used by the threads to start writing the output elements [16].

BFS-4K exploits prefix-sum procedures to manage the frontier queues as well as the edge-discover visit (see Section 5.4). During frontier propagation, the prefix-sum is used to compute the scatter offset needed by each thread to assemble, in parallel, global edge frontiers from expanded neighbours and when producing unique unvisited vertices into global vertex frontiers. Since the first offset must be zero, the prefix-sum results are shifted to right of one position to implement the *exclusive* variant.

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

Fig. 3. Overview of a prefix-sum procedure implemented with shuffle instructions

BFS-4K implements a two-level exclusive prefix-sum, that is, at warp-level and block-level. The first is implemented by using Kepler warp-shuffle instructions, which guarantee the result computation in $\log n$ steps rather than $2 \log n$ as in the most efficient implementations in literature that rely on shared memory (e.g., [16]). Figure 3 shows a high-level representation of such a prefix-sum procedure implemented with a warp shuffle instruction (i.e., $\_\_shfl\_up()$).

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

Finally, at each frontier propagation step, *BFS-4K* checks whether every frontier vertices have at most one neighbour (i.e., scatter offset either 0 or 1 for each thread). The check, which work complexity is $O(1)$, aims at running, when possible, a more efficient *binary* variant of the exclusive prefix-sum [25], which has been implemented with the intrinsics instructions __BALLOT and __POPC.

5.2 Dynamic Virtual Warps

The concept of *virtual warp* has been presented in [5] to address the problem of workload imbalance in GPU programming. The idea is to allocate a chunk of tasks to each warp and to execute different tasks as serial rather than assigning a different task to each thread. Multiple threads are used in a warp for explicit SIMD operations only, thus preventing branch-divergence altogether.

The speedup provided by virtual warps is strictly related to the virtual warp size. As shown in the experimental results [5], a wrong size setting could also lead to a speedup decrease.

In the BFS context, the virtual warps technique can be applied to increase the thread coalescence during the accesses to the adjacent lists and to reduce their divergence in the frontier propagation steps. The main limitation of such a technique in BFS occurs when the virtual warp size does not properly fit the vertex degree, thus leading to unused threads. In case of vertices with very different degree over the propagation steps, the size choice may not be always appropriated. Thus, differently from [5], *BFS-4K* implements a *dynamic* virtual warp, whereby the warp size is calibrated at each frontier propagation step i, as follows:

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

where *#ResThreads* refers to the maximum number of resident threads in case of multi-block kernel while thread block size in case of single-block kernel (see Section 5.5). $nearest\_pow2$ is the lower nearest power of two that rounds the division, while $|F_i|$ is the size of the actual frontier.

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

The choice of the warp size also directly affects the problem of duplicate vertices. A small size, which leads to finer granularity of warp work and fine grained synchronization, involves less duplicate vertices during frontier propagation. In contrast, large sizes of warps may reduce the synchronization overhead but they lead to more duplicates, thus requiring more resources for the duplicate detection and correction, as explained in Section 6.

5.3 Dynamic Parallelism

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

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

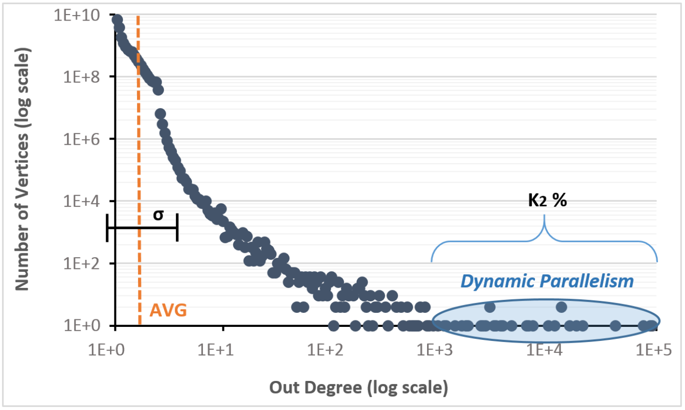


Fig. 4. Example of dynamic parallelism applied to a subset of frontier vertices of a power-law graph

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

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

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

In *BFS-4K*, the child kernels are configured to ensure the minimum overhead of the child thread synchronization, and the best balancing among parent and child threads. Figure 5 shows an example of three different kernel settings in terms of number of blocks (with a fixed block size), given a parent kernel (leftmost side of figure) and a child kernel (lower side of the figure). Case *(a)* represents an oversized kernel, in which the blocks are more than the vertex neighbours and, thus, they conclude shorter than the other threads of the parent. Nevertheless, the many child blocks involve many atomic operations to update the frontier data structures and an underutilization of the fast local queues. In contrast, case (c) represents an undersized kernel, in which less blocks manage many vertex neighbours. Even though it involves less atomic operations, this configuration leads to imbalance with regards to the parent threads, since the parent kernel must wait for all the threads (including

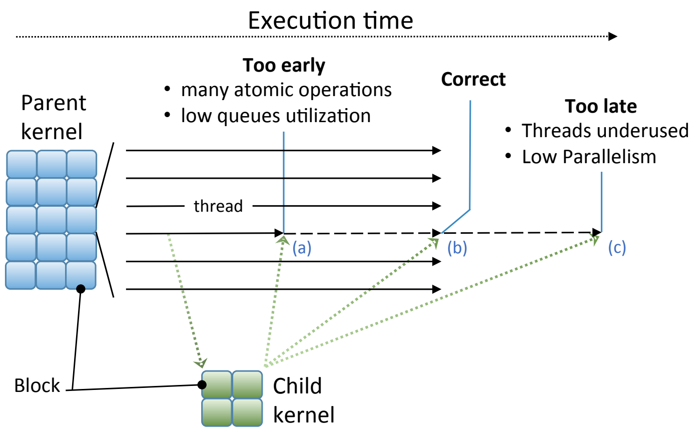


Fig. 5. Block number setting: (a) oversized kernel, (b) correctly sized kernel, (c) undersized kernel

child threads) to end before carrying on with the next propagation step (see thread synchronization in Kepler dynamic parallelism [13]).

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

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

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

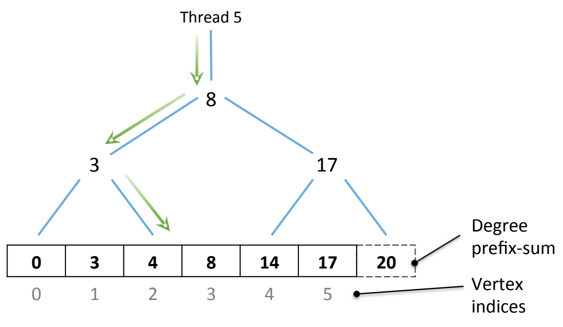
In our experimental results, $K_3 = 16$ (i.e, each thread of the child kernels sequentially manages a queue of 16 vertex neighbour) provides the best BFS performance for the analysed graphs.

5.4 Edge-discover

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

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

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



(a)

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

(b)

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

With this procedure, the workload is always balanced, the local queues are filled equally and the duplicates are considerably reduced since the parallel visit is for edges (see Section 6). The local queue management and the global memory accessing and synchronization are similar to those implemented in the dynamic virtual warp strategy.

Finally, *BFS-4K* implements an extended edge-discover technique (EXT) to optimize the visit of middle size degree vertices. When the last thread of a warp finds a vertex with a degree greater than the warp size, it shares the offset with a shuffle operation and directly assigns threads without performing a new iteration of binary search. As shown in Section 7, this optimization provides a sensible speedup improvement in the BFS visit of several graphs.

BFS-4K applies the edge-discover technique as an alternative of dynamic virtual warps to be combined with dynamic parallelism. With this new combination, the

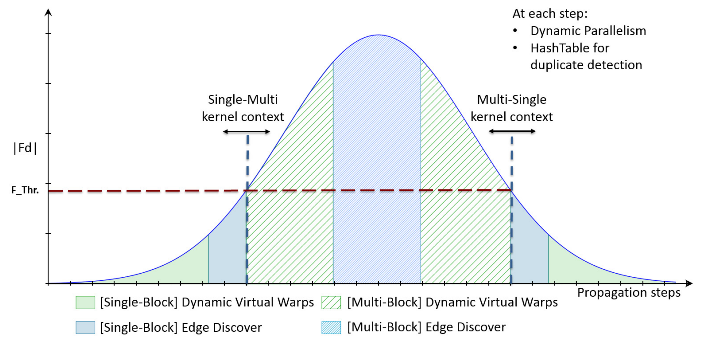


Fig. 7. Single and multi-block kernel use during frontier propagation steps

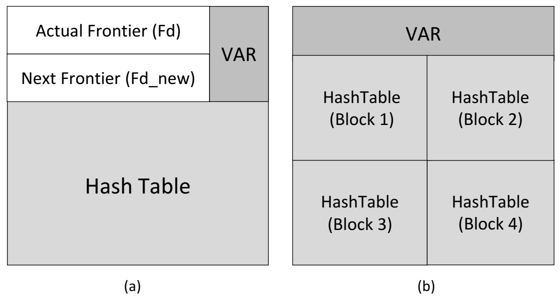


Fig. 8. Shared memory organization: (a) single-block kernel, (b) multi-block kernel

threshold of dynamic parallelism (K_2) can be increased more than in the former combination. This is due to the fact that the warp parallelism on edges allows high-degree vertices to be handled more efficiently than the warp parallelism on vertices. Nevertheless, the overhead introduced by the thread assignment limits the edge-discover application.

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

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

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

5.5 Single-block vs. Multi-block Kernel

In a parallel BFS visit based on frontier propagation, the frontier size follows a trend as that shown in Figure 7. In the first and last steps of the overall frontier propagation the available parallelism is particularly limited. As a consequence, in these propagation periods, it is more convenient to handle the frontier vertices with a single block of threads. This allows the whole frontier to be maintained in shared memory and the block threads to exploit the efficient synchronization and communication mechanisms.

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

A threshold ($F\_Threshold$) is statically calibrated depending on the graph characteristics. At each propagation step, *BFS-4K* runs the single-block or the multi-block kernel if the current frontier size is smaller or larger, respectively, than the threshold. In general, the single-block kernel is run in the first and last propagation steps, while the multi-block kernel is run in the middle steps, as shown in Figure 7.

The threshold calibration impacts on the organization of the shared memory of each streaming multiprocessor. Figure 8 depicts the shared memory organization in case of single or multi-block kernel. In the first case, the shared memory stores the frontier data structures (Fd and Fd_new), kernel variables, and the hash table for implementing duplicate detection and correction. The memory is sized as follows:

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

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

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

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

In case of multi-block kernel, the shared memory is organized as depicted in Figure 7(b). In this case, a hash table instance is dedicated to each thread block, and the tables are sized as follows:

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

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

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

5.6 Coalesced Read/Write Memory Accesses

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

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

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

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

The coalescence problem for memory *reads* is suffered from the virtual warp technique only. In contrast, coalescence for memory writes is suffered from all the techniques in general (i.e., virtual warps, dynamic parallelism, and edge discover). At each propagation step, the threads exploit *local queues*, which are data structures in thread registers, to store and filter the neighbour vertices. After the filtering phase, each thread updates the own frontier segment in the global memory (Fd_{new}). In the classic context, the Fd_{new} updating is performed in parallel, where each thread sequentially writes the own vertices starting from the scatter offset calculated by prefix-sum (see Section 5.1). This leads to coalescence problems since the memory accesses rely on the number of vertices to be written in global memory.

BFS-4K implements a technique to induce coalescence in memory writes as follows (see Figure 9):

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

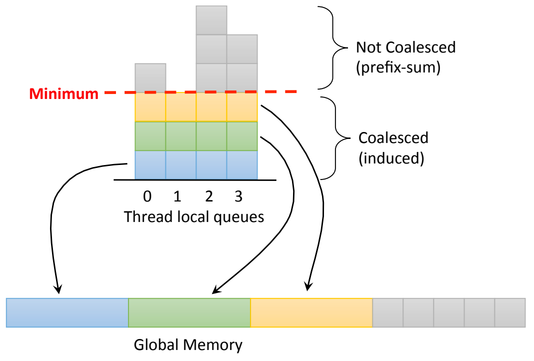


Fig. 9. Example of induced coalescence in memory accesses

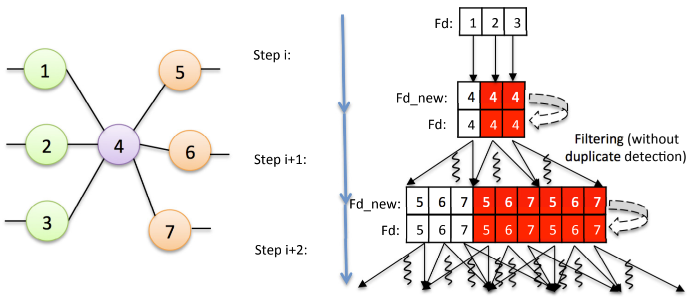


Fig. 10. Example of duplicates exponential growth

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

6 Duplicate Detection and Correction

Duplicate vertices are a relevant problem in the parallel BFS visit of graphs. Duplicate vertices are generated whenever two or more threads visit the same vertex at the same time and, as a consequence, they cause redundant work among threads during frontier propagation. Figure 10 shows an example that underlines how such a redundant work grows exponentially through the frontier propagation steps.

BFS-4K implements a hash table in shared memory (i.e., one per streaming multiprocessor) to detect and correct duplicates, and takes advantage of the 8-bank shared memory mode of Kepler to guarantee high performance of the table accesses. At each propagation step, each frontier thread invokes the `hash64` procedure depicted in Figure 11 to update the hash table with the visited vertex (v).

```
__device__ bool hash64 ( vertex $v$ )
1:    H_SZ : Hash_Table_Size
2:    $h$ = hash($v$)          $\rightarrow h \in [0, \text{H\_SZ}]$
3:*   HashTable[$h$] = merge($v, thread_{id}$)
4:    $recover$ = HashTable[$h$];
5:*   $(v_R,\ thread_{idR})$ = split($recover$)
6:    return $thread_{id} \neq thread_{idR} \wedge v = v_R$
                    *volatile int2 are not supported in CUDA
```

Fig. 11. Main steps of the hash table managing algorithm

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

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

Conflicts are proportionally related to the size of the hash table and, thus, to the size of shared memory allocated for the hash table. As explained in Section 5.5 and shown in Figure 7, the setting of the *FrontierLimit* knob to run a single block rather than a multi-block kernel directly impacts on the hash table size and, thus, to the capability of *BFS-4K* of detecting duplicate vertices rather than conflicts.

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

$$HashTable[h] = merge(v_x, thread_1)$$
$$HashTable[h] = merge(v_x, thread_2)$$
$$recover = HashTable[h] \qquad \text{// by } thread_1$$
$$recover = HashTable[h] \qquad \text{// by } thread_2$$

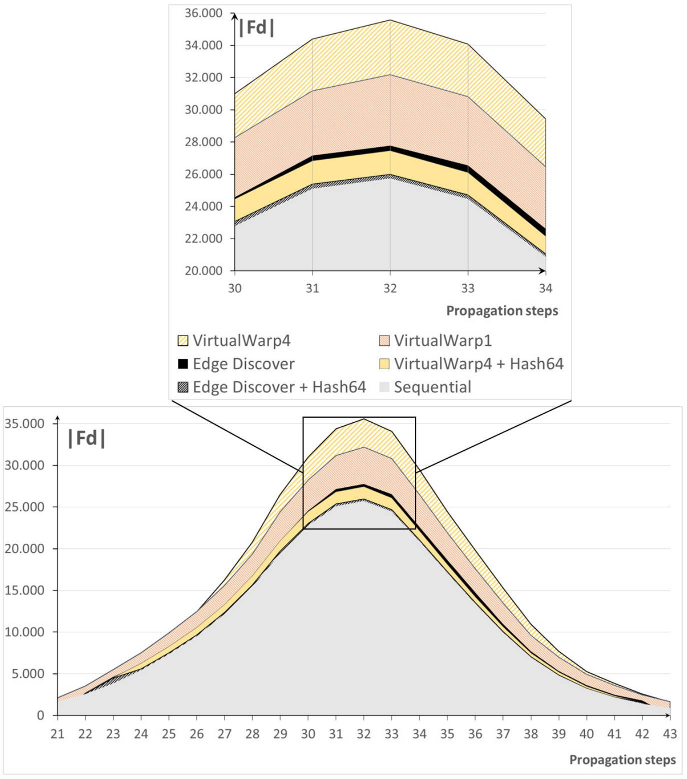


Fig. 12. Example of duplicates caused by different visiting techniques and the effect of the proposed detection strategy

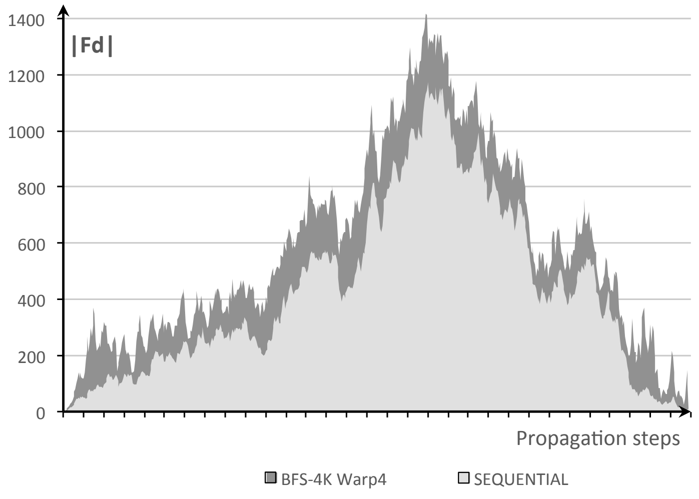
allows the procedure to detect the duplicate, while the sequence:

$$HashTable[h] = merge(v_x, thread_1)$$
$$recover = HashTable[h] \qquad \text{// by } thread_1$$
$$HashTable[h] = merge(v_x, thread_2)$$
$$recover = HashTable[h] \qquad \text{// by } thread_2$$

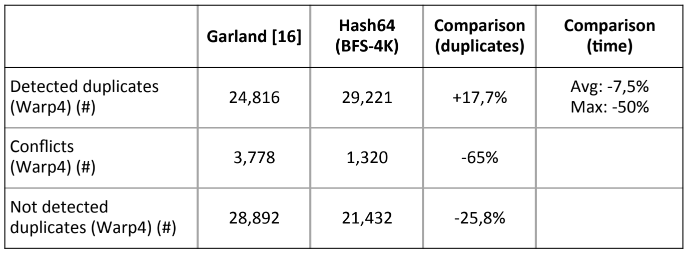
does not allow the procedure to detect the duplicate, which is conservatively maintained in the frontier. Duplicates generated by threads of different blocks are not detectable.

Since the duplicate issue occurs mainly among threads of the same warp, the problem affects more the visit by virtual warps than by edge-discover. Indeed, since in edge-discover the exploration is performed on edges, the chances to visit the same vertex more times is considerably small. Figure 12 shows the problem with the different visit strategies and the efficiency of the implemented technique of duplicate detection.

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



(a)



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

(b)

Fig. 13. Comparison between the duplicate detection techniques implemented in [16] and in *BFS-4K*

laps *VirtualWarp4* + *Hash64* and has not been reported in the figure for the sake of clarity.

For the best of our knowledge, the duplicate detection and correction problem has been addressed in literature only in [16]. Differently from our solution, [16] implements a hash table per warp and a procedure that writes and reads $lane_{id}$ (instead of $thread_{id}$) and v non atomically in the hash table. This involves more overhead due to the number of memory accesses and, by implementing disjointed hash tables, it suffers more from conflicts and non detectable duplicates. Figure 13 shows a representative example in which the techniques implemented in [16] and in *BFS-4K* are compared. In particular, Figure 13(a) shows the duplicates generated by adopting virtual warp of size 4 over the propagation steps. Figure 13(b) reports the total number of detected duplicates for both the solutions and the corresponding improvement on the overall performance (average and maximum improvement). The figure also reports the total number of conflicts and non detected duplicates.

7 Experimental results

BFS-4K has been run on two main sets of graphs. The first set is from Stanford Network Analysis Platform

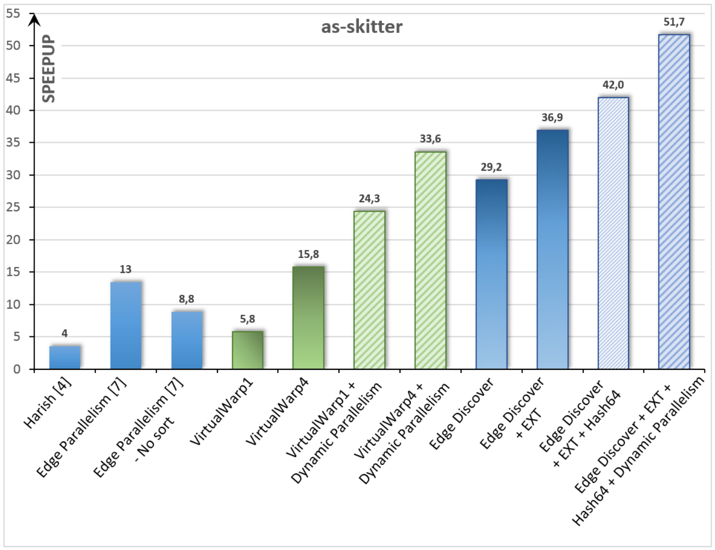


Fig. 14. Impact of virtual warp, edge discover, dynamic parallelism and duplicate detection

(SNAP) [26]. It includes graphs from different contexts, such as, product co-purchasing networks, web page hyperlink graphs, network with ground-truth communities, road networks, social networks, time-evolving graphs and small-word phenomenon graphs. The second set is from the 10th DIMACS Implementation Challenge [27]. The $random.2Mv.128Me$ and $rmat.2Mv.128Me$ datasets have been generated by using GTGraph [28]. Table 1 shows each graph characteristics in terms of number of vertices (V, in millions), edges (E, in millions), size of graph diameter, average degree, standard deviation, and mode. *BFS-4K* has been run on a NVIDIA GEFORCE GTX 780 device [29] with CUDA Toolkit 5.0, with AMD Phenom II X6 1055T (3GHz) host processor (Ubuntu 10.04 operating system).

Figure 14 shows an example of the impact of each *BFS-4K* feature (Section 5) and the duplicate detection and correction technique (Section 6) on the overall speedup. The feature contributions are shown for a sample graph (*as-skitter*), by taking the speedup of the parallel BFS implementations in [4] and [7] versus the sequential implementation as reference point. The figure underlines that the best speedup is achieved by the combination of these features. In particular, the best feature configuration and combination can be obtained by properly setting the presented *knobs*. As explained in the follows, such a setting is correlated to the characteristics of the visited graphs and the characteristics of the GPU device.

Table 2 and Figure 15 report the performance comparison of *BFS-4K* with the most representative implementations at the state of the art in terms of visiting time and speedup, respectively. The performance of the state-of-the-art implementations are the best ones we obtained by tuning the kernel configurations (in terms of number of threads per block and number of blocks

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

TABLE 1
Characteristics of the graph datasets on which *BFS-4K* has been evaluated

		Harish [4] (ms)	Edge Parall. [7] (ms)	Static Virtual Warp [5] (ms)	Luo [15] (ms)	Garland [16] (ms)	BFS-4K (ms)
Set 1	Amazon0505	5.2	7.2	5.2 (W1)	4.3	–	1.5
	web-Google	12.0	9.2	12.0 (W1)	7.3	–	1.6
	com-youtube	57.0	5.5	19.0 (W4)	out-of-time	–	3.1
	as-skitter	95.0	24.0	28.0 (W4)	out-of-time	–	6.5
	roadNet-CA	120.7	154.4	120.7 (W1)	20.2	–	5.5
	soc-LiveJournal1	91.0	61.0	52.0 (W2)	out-of-time	–	24.4
	Gen-ForestFire	37.0	5.4	14.0 (W4)	out-of-time	–	2.7
	Gen-SmallWorld	33.0	27.0	24.0 (W2)	out-of-time	–	15.1
Set 2	europe.osm	59,620.0	78,422.0	59,620.0 (W1)	684.0	305	264.8
	hugehubbles-00020	8,123.0	11,922.0	8,123.0 (W1)	220.0	103	95.9
	nlpkkt160	351.0	1486.0	351.0 (W1)	out-of-memory	80.4	39.2
	audikw1	68.0	185.0	36.0 (W4)	54	21.5	11.1
	cage15	95.0	213.0	95.0 (W1)	96	42.2	28.8
	kkt_power	36.0	24.5	36.0 (W1)	24	8.8	8.5
	coPapersCiteseer	21.2	40.6	11.4 (W4)	out-of-memory	8.6	4.9
	kron_g500-lon20	675.0	47.4	67.0 (W32)	out-of-time	out-of-memory	34.4
	random.2Mv.128Me	112.0	73.0	63.0 (W16)	out-of-time	66.5	52.0
	rmat.2Mv.128Me	103.0	62.0	56.0 (W4)	out-of-time	out-of-memory	43.4

TABLE 2
Performance comparison (BFS visiting time) of *BFS-4K* with the most representative implementations at the state of the art

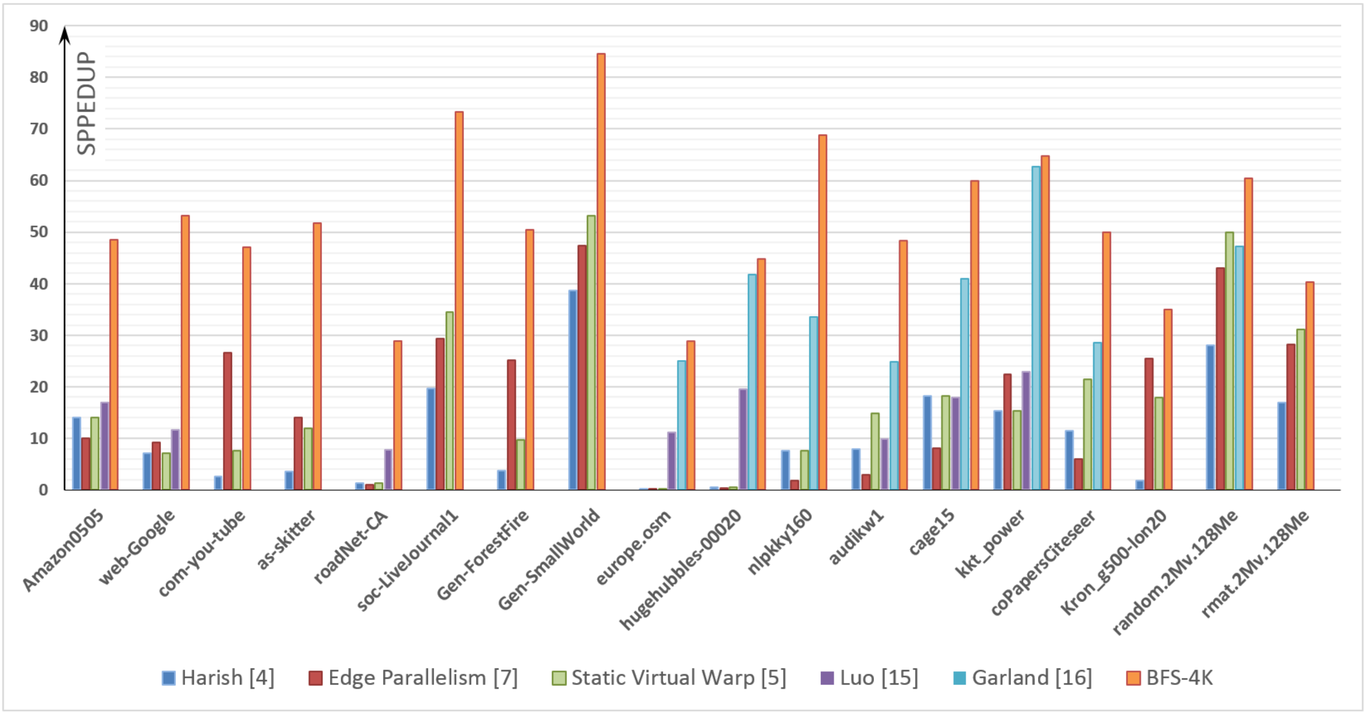


Fig. 15. Performance comparison (speedup) of *BFS-4K* with the most representative implementations at the state of the art

per grid) for our GPU device. For the static virtual warp technique [5], Table 2 reports the size of virtual warp statically set to obtain the best performance results. The Garland's implementation [16] does not support the template representation of the first set of graphs.

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

We observed that K_1 is strictly related to the graph standard deviation and average degree. In particular, we measured the best speedups by increasing this knob value proportionally to the graph deviation and degree, starting from the lowest value ($K_1 = 4$) for graphs with low deviation and degree (e.g., road networks in general, *web-Google*, *kkt_power*, etc.), to the highest value ($K_1 = 32$) in graphs with high average (e.g., *random.2Mv.128Me*) or high standard deviation (e.g., *com-youtube*).

K_2 controls the use of dynamic parallelism, which achieves the best results with very high mode networks (e.g., mode greater than 2048 such as in *as-skitter*, *rmat.2Mv.128Me*, etc.) to deal with the sporadic high workloads. K_2, which maximum value is 0.15% in our experiments, should be higher for graphs with low average and inversely proportional to K_1. As explained in Section 5.3, the larger is the minimum warp size, the smaller is the sub-set of vertices that can be managed by dynamic kernels to improve the BFS performance. This is due to the fact that large virtual warps can handle the workload imbalance more efficiently (i.e, with less overhead) than dynamic parallelism.

K_3 impacts on the block size of *child kernels* when applying dynamic parallelism. The right value is more related to the GPU device characteristics and should be optimized heuristically. In our experimental results, $K3 = 16$ provides the best BFS performance for all the analysed graphs.

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

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

Finally, K_6 and K_7 sets the block size in the multi-block kernel and the threshold for switching the writing mode in global memory, respectively. They best values depend on the GPU device characteristics. In our experiments, we heuristically set $K_6 = 128$ and $K7 = 10$.

8 Concluding remarks

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

References

[1] J.-D. Cho, S. Raje, and M. Sarrafzadeh, "Fast approximation algorithms on maxcut, k-coloring, and k-color ordering for vlsi applications," *IEEE Trans. Comput.*, vol. 47, no. 11, pp. 1253–1266, Nov. 1998.
[2] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 1, pp. 121–141, Jan. 1979.
[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. MIT press, 2009.
[4] 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.
[5] 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.
[6] S. M. Yangdong Deng, Bo D. Wang, "Taming irregular eda applications on gpus," in *Proc. of the IEEE International Conference on Computer-Aided Design (ICCAD'09)*, 2009, pp. 539–546.
[7] Y. Jia, V. Lu, J. Hoberock, M. Garland, and J. C.Hart, *GPU Computing Gems Jade Edition: Chapter 2. Edge v. Node Parallelism for Graph Centrality Metrics*. Morgan Kaufmann Publishers, 2011, ch. 11.
[8] G. Singla, A. Tiwari, and D. P. Singh, "New approach for graph algorithms on gpu using cuda," *International Journal of Computer Applications*, 2013.
[9] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–11.
[10] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proceedings of the 22Nd ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10, 2010, pp. 303–314.
[11] Y. Xia and V. K. Prasanna, "Topologically adaptive parallel breadth-first search on multicore processors," in *Proceedings of the 21st International Conference on Parallel and Distributed Computing and Systems*, ser. PDCS09, 2009.
[12] D. P. Scarpazza, O. Villa, and F. Petrini, "Efficient breadth-first search on the cell/be processor," *IEEE Transactions on Parallel Distributed Systems*, vol. 19, no. 10, pp. 1381–1395, 2008.
[13] NVIDIA, "Kepler gk110," www.nvidia.com/content/PDF/kepler/NV_DS_Tesla_KCompute_Arch_May_2012_LR.pdf.
[14] ——, "Cuda home page," http://www.nvidia.co.uk/object/cuda-parallel-computing-uk.html.
[15] L. Luo, M. Wong, and W.-m. Hwu, "An effective gpu implementation of breadth-first search," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10, 2010, pp. 52–55.
[16] 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.
[17] S. Xiao and W. chun Feng, "Inter-block gpu communication via fast barrier synchronization," Dept. of Computer Science Virginia Tech, Tech. Rep., 2009.
[18] G. E. Blelloch, "Prefix sums and their applications," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-90-190, 1990.
[19] ——, "Scans as primitive parallel operations," *IEEE Transactions on Computers*, vol. 38, no. 11, pp. 1526–1538, 1989.
[20] S. Chatterjee, G. E. Blelloch, and M. Zagha, "Scan primitives for vector computers," in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, 1990, pp. 666–675.
[21] M. Billeter, O. Olsson, and U. Assarsson, "Efficient stream compaction on wide simd many-core architectures," in *Proceedings of the Conference on High Performance Graphics 2009*, 2009, pp. 159–166.
[22] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli, "Fast scan algorithms on graphics processors," in *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ser. ICS '08, 2008, pp. 205–213.
[23] D. Merril and A. Grimshaw, "Parallel scan for stream architectures," Department of Computer Science, University of Virginia, Tech. Rep. CS-200914, 2009.
[24] S. Sengupta, M. Harris, and M. Garland, "Efficient parallel scan algorithms for gpus," NVIDIA, Tech. Rep., 2009.
[25] M. Harris and M. Garland, *GPU Computing Gems Emerald Edition: Optimizing Parallel Prefix Operations for the Fermi Architecture*. Addison Wesley Professional, 2008, ch. 3.
[26] Stanford Network Analysis Platform, "Stanford university," 2013. [Online]. Available: http://snap.stanford.edu/data/index.html
[27] "10th DIMACS Implementation Challenge," http://www.cc.gatech.edu/dimacs10/index.shtml.
[28] "GTgraph: A suite of synthetic random graph generators," http://www.cse.psu.edu/ madduri/software/GTgraph/.
[29] "NVIDIA GEFORCE GTX 780," http://www.nvidia.com/gtx-700-graphics-cards/gtx-780/.
[30] Wolfram Mathematica 9, "Pseudo diameter," 2012. [Online]. Available: http://reference.wolfram.com/mathematica/GraphUtilities/ref/PseudoDiameter.html.

Federico Busato received the Master degree in Computer Science from the University of Verona in 2014. 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 50 papers on conference proceedings and journals.