Graph Algorithms on GPUs

Federico Busato Nicola Bombieri

Dept. Computer Science
University of Verona
Italy

Abstract

This chapter introduces the topic of graph algorithms on GPUs. It starts by presenting and comparing the main important data structures and techniques applied for representing and analysing graphs on GPUs at the state of the art. It then presents the theory and an updated review of the most efficient implementations of graph algorithms for GPUs. In particular, the chapter focuses on graph traversal algorithms (breadth-first search), single-source shortest path (Djikstra, Bellman-Ford, delta stepping, hybrids), and all-pair shortest path (Floyd-Warshall). By the end of the chapter, load balancing and memory access techniques are discussed through an overview of their main issues and management techniques.

Keywords: Graph algorithms, BFS, SSSP, APSP, Load balancing.

1. Graph Representation for GPUs

The graph representation adopted when implementing a graph algorithm for GPUs strongly affects the implementation efficiency and performance. The three most common representations are *adjacency matrices*, *adjacency lists*, and
5 *egdes lists* [1, 2]. They have different characteristics and each one finds the best application in different contexts (i.e., graph and algorithm characteristics).

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

☆Fully documented templates are available in the elsarticle package on CTAN.
*Corresponding author
Email address: support@elsevier.com (Federico Busato Nicola Bombieri)
URL: www.elsevier.com (Federico Busato Nicola Bombieri)
[1]Since 1880.

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

Table 1: Main feature of data representations.

Given a graph $G = (V, E)$, where V is the set of vertices, E is the set of edges, and d_{max} is the largest diameter of the graph, Table 1 summarizes the
15 main features of the data representations, which will be discussed in detail in the next paragraphs.

1.1. Adjacency Matrices

An adjacency matrix allows representing a graph with a $V \times V$ matrix $M = [f(i,j)]$ where each element $f(i,j)$ contains the attributes of the edge (i,j).
20 If the edges do not have any attribute, the graph can be represented with a boolean matrix to save memory space (see Figure 1).

Common algorithms that use this representation are *all-pair shortest path* and *transitive closure* [3, 4, 5, 6, 7, 8, 9]. If the graph is weighted, each value of $f(i,j)$ is defined as follows:

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

On GPUs, both directed and undirected graphs represented by an adjacency matrix take $O(|V|^2)$ memory space, since the whole matrix is stored in memory with a large continuous array. In GPU architectures, it is also important, for performance reasons, to align the matrix with memory to improve coalescence of memory accesses. In this context, the CUDA language provides the function *cudaMallocPitch* [10] to pad the data allocation, with the aim of meeting the alignment requirements for memory coalescing. In this case the indexing changes as follow:

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

The $O(|V|^2)$ memory space required is the main limitation of the adjacency matrices . Even on recent GPUs, they allow handling fairly small graphs. As an example, considering a GPU device with 4GB of DRAM, the biggest graph that
25 can be represented through an adjacency matrix can have a maximum number of vertices equals to 32,768 (which, for actual graph datasets, is considered restrictive). In general, adjacency matrices best apply to represent small and dense graphs (i.e. $|E| \approx |V|^2$). In some cases, such as for the all-pairs shortest path problem, graphs larger than the GPU memory are partitioned and each
30 part processed independently [9, 8, 7].

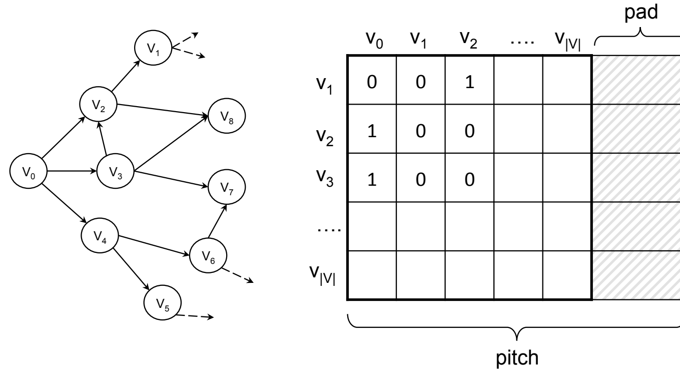

FIG. 1: *Matrix representation of a graph in memory*

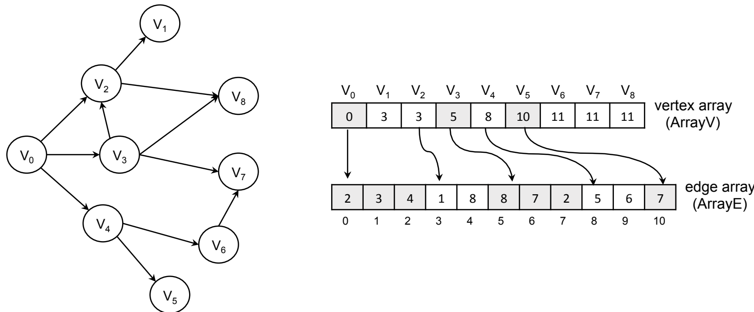

FIG. 2: *Adjacency list representation of a weighted graph*

1.2. Adjacency Lists

The adjacency lists are the most common representation for sparse graphs, where the number of edges is typically a constant factor larger than $|V|$. Since the sequential implementation of the adjacency lists relies on pointers, it is not suitable for GPUs. they are replaced, in GPU imlementations, by the *Compressed Sparse Row* (CSR) or *Compressed Row Storage* (CRS) sparse matrix format [11, 12].

In general, an adjacency list consists of an array of vertices (ArrayV) and an array of edges (ArrayE), where each element in the vertex array stores the starting index (in the edge array) of the edges outgoing from each node. The edge array stores the destination vertices of each edge (see Figure 2). This allows visiting the neighbours of a vertex v by reading the edge array from ArrayV[v] to ArrayV[v + 1].

The attributes of the edges are in general stored in the edge array through an *array of structures* (AoS). For example, in a weighted graph, the destination and the weight of an edge can be stored in a structure with two integer values (*int2* in CUDA [13]). Such a data organization allows many scattered memory accesses to be avoided and, as a consequence, the algorithm performance to be

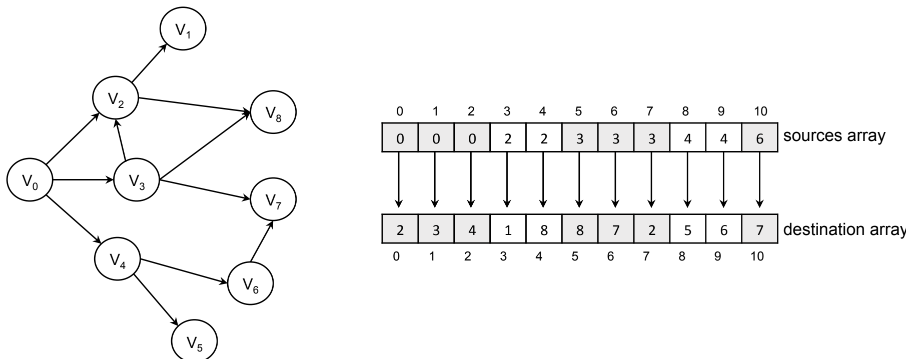

FIG. 3: *Edges list representation of a weighted graph*

improved.

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

The main issues of the CSR format are the load balancing and the memory coalescing, due to the irregular structure of such a format. If the algorithm involves visiting each vertex at each iteration, the memory coalescing for the vertex array is simple to achieve but, on the other hand, it is difficult to achieve for the edge array. Achieving both load balancing and memory coalescing requires advanced and sophisticated implementation techniques (see Chapter (5)).

For many graph algorithms, the adjacency list representation guarantees better performance than adjacency matrix and edge lists [14, 15, 16, 17, 18].

1.3. Edges List

The edge list representation of a graph, also called *coordinate list* (COO) sparse matrix [19], consists of two arrays of size $|E|$ that store the source and the destination of each edge (see Figure 3). To improve the memory coalescing, similarly to CSR, the source, the destination and other edge attributes (such as the edge weight) can be stored in a single structure (AoS) [20].

Storing some vertex attributes in external arrays is also necessary in many graph algorithms. For this reason, the edge list is sorted by the first vertex in each ordered pair, such that adjacent threads are assigned to edges with the same source vertex. This allows improving coalescence of memory accesses for retrieving the vertex attributes. In some cases, sorting the edge list in the lexicographic order may also improve coalescence of memory accesses for retrieving the attributes of the destination vertices [21]. The edge organization in a sorted list allows reducing the complexity (from $O(|E|)$ to $O(\log|E|)$) of verifying whether an edge is in the graph, by means of a simple binary search [22].

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

The edge list representation is suitable in those algorithms that iterate over all edges. For example, it is used in the GPU implementation of algorithms like *betweenness centrality* [21, 23]. In general, this format does not guarantee performance comparable to the adjacency list but it allows achieving both perfect load balancing and memory coalescing with a simple thread mapping. In graphs with a non-uniform distribution of vertex degrees, the COO format is generally more efficient than CSR [21, 24].

2. Graph Traversal Algorithms: the Breadth First Search (BFS)

Breadth-first search (BFS) is a core primitive for graph traversal and the basis for many higher-level graph analysis 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. Algorithm 1 summarizes the traditional sequential algorithm [1], where Q is a FIFO queue data structure that stores not yet visited vertices, Distance$[v]$ represents the distance of vertex v from the source vertex s (number of edges in the path), and Parent$[v]$ represents the parent vertex of v. An unvisited vertex v is denoted with Distance$[v]$ equal to ∞. The asymptotic time complexity of the sequential algorithm is $O(|V| + |E|)$.

In the context of GPUs, the BFS algorithm is the only graph traversal method applied since it exposes a high level of parallelism. In contrast, the depth first search (DFS) traversal is never applied due to its intrinsic sequentiality.

2.1. The Frontier-based Parallel Implementation of BFS

The most efficient parallel implementations of BFS for GPUs exploit the concept of *frontier* [1]. They generate 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.

The frontier propagation relies on two data structures, F and F'. F represents the actual frontier, which is read by the parallel threads to start the propagation step. F' is written by the threads to generate the frontier for the next BFS step. At each step, F' is filtered and swapped into F for the next

Algorithm 1 Sequential BFS Algorithm

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

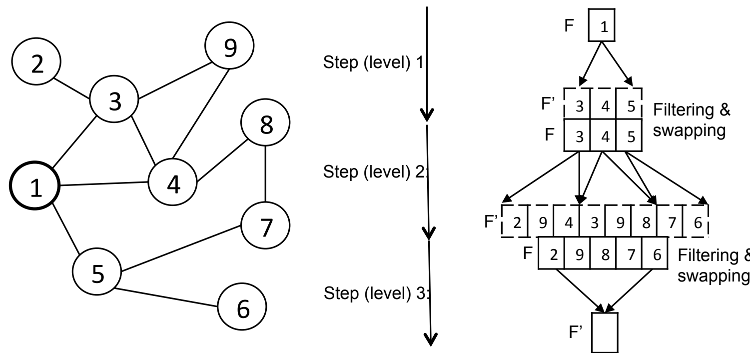

FIG. 4: *Example of BFS visit starting from vertex* "0".

iteration. Figure 4 shows an example, in which starting from vertex "1", the BFS visit concludes in three steps[2].

The filtering steps aim 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 3 visited by a thread from vertex 4 in step two of Figure 4). When more threads visit the same neighbour in the same propagation step (e.g., vertex 9 visited by threads 3 and 4 in step two), they generate *duplicate* vertices in the

[2]For the sake of clarity, the figure shows F' firstly written and then filtered. As explained in the following paragraphs, to reduce the global memory accesses, some implementations firstly filter the next frontier and, then, they write the F' data.

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. The most efficient BFS implementations detect and eliminate duplicates by exploiting hash tables, Kepler 8-byte memory access mode, and warp shuffle instructions [16, 15].

Several techniques have been proposed in literature to efficiently parallelize the BFS algorithm for GPUs. Harish and Narayanan [25] proposed the first approach, which relies on exploring 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, that is based on CSR representation, 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 $O(|V||D| + |E|)$, where $O(|V||D|)$ 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, such an implementation is slower than the sequential version.

A partial solution to the problem of workload imbalance has been proposed in [18] by adopting the same graph representation. 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.

[26] 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 $O(|V| + |E||D|)$, 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 $O(|V|^2)$.

[21] and [24] 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 [17]. The algorithm exploits a single hierarchical queue shared across all thread blocks and an inter-block synchronization [27] to save queue accesses in global memory. Nevertheless, the small frontier size requested to

Algorithm 2 Overview of a prefix-sum procedure implemented with shuffle instructions

```
exclusiveWarpPrefixSum

    for (i = 1; i ≤ 16; i = i ∗ 2) do
        n = __shfl_up(v, i, 32)
        if lane_id ≥ i then
            v += n
    end
    __shfl_up(v, 1, 32)
    if lane_id = 0 then
        v = 0
```

avoid global memory writes and the visit exclusively based on vertex parallelism limit the overall speedup. In addition, the generally high-degree vertices are handled through an expensive pre-computation phase rather than at run time.

Merrill, Garland and Grimshaw [16] present an algorithm implementation 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.

Beamer, Asanović and Patterson propose a CPU multi-core hybrid approach, which combines the frontier based algorithm along with a bottom-up BFS algorithm. The bottom-up algorithm can heavily reduce the number of edges examined compared to common parallel algorithms. The bottom-up BFS traversal searches vertices of the next iteration (at distance $L+1$) in the reverse direction by exploring the unvisted vertices of the graph. This approach requires only a thread per unvisted vertex that explores the neighbour until a previous visited vertex is found (at distance L). The bottom-up BFS is particularly efficient on low-diameter graphs where, at the ending iterations, a substantial fraction of neighbours are valid parents. In the context of GPUs, such a bottom-up approach for graph traversal has been implemented by Wang et al. in the Gunrock framework [28] and by Hiragushi et al. [29].

2.2. BFS-4K

BFS-4K [15] is a parallel implementation of BFS for GPUs that exploits the more advanced features of GPU-based platforms (i.e., NVIDIA Kepler, Maxwell [30, 31]) to improve the execution speedup w.r.t. the sequential CPU implementations and to achieve an asymptotically optimal work complexity.

BFS-4K implements different techniques to deal with the potential workload imbalance and thread divergence caused by any actual graph non-homogeneity (e.g., number of vertices, edges, diameter, and vertex degree):

- *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 [32, 33]. Such a procedure is implemented through warp shuffle instructions of the Kepler architecture. 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. Algorithm 2 shows a high-level representation of such a prefix-sum procedure implemented with a warp shuffle instruction (i.e., $__shfl_up()$).

- *Dynamic virtual warps.* The *virtual warp* technique presented in [18] is applied to minimize the waste of GPU resources and to reduce the divergence during the neighbour inspection phase. 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.

  Differently from [18], BFS-4K implements a strategy to dynamically calibrate the warp size at each frontier propagation step. 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.

- *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.

- *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. 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.

Algorithm 3 Main steps of the hash table managing algorithm

HASH64

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** $thread_{id} \neq thread_{idR} \wedge v = v_R$

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. 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.

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

	Harish[25]	Virtual Warps[18]	Edge Parallelism[21]	Luo[17]	Garland[16]	BFS-4K[15]
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 parallelism	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[27], Thread barriers	Host-Device, Inter-block[27]	Host-Device, Inter-block[27], Thread barriers

TABLE 2: Comparison of the most representative BFS implementations at the state of the art with BFS-4K.

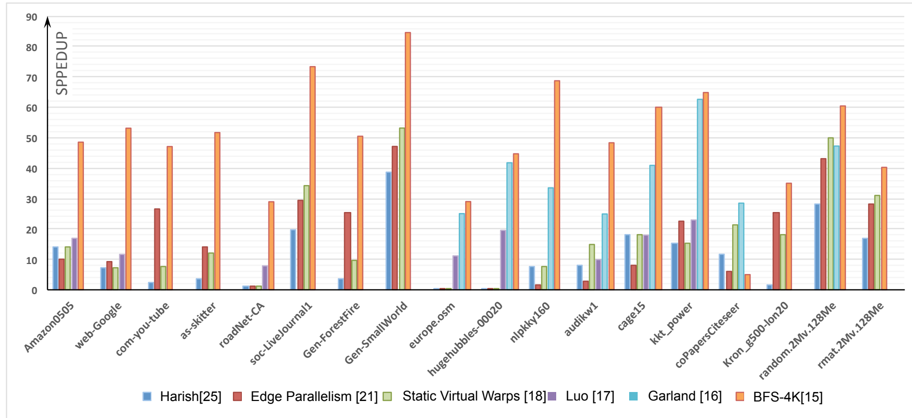

FIG. 5: *Performance comparison (speedup) of BFS-4K with the most representative implementations at the state of the art.*

275 BFS-4K 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. Table 2 summarizes the differences between the most representative BFS implementations at the state of the art and BFS-4K, while Figure 5 reports a representative comparison of 280 speedups among the BFS implementations for GPUs presented in paragraph 2.1, BFS-4K, and the sequential counterpart.

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 285 the user to optimize the visiting strategy through different knobs.

3. The Single-Source Shortest Path (SSSP) problem

Given a weighted graph $G = (V, E)$, where V is the set of vertices and $E \subseteq (V \times V)$ is the set of edges, the single-source shortest path (SSSP) problem consists of finding the shortest paths from a single source vertex to all other vertices [1]. Such a well-known and long-studied problem arises in many different domains, such as, road networks, routing protocols, artificial intelligence, social networks, data mining, and VLSI chip layout.

The de-facto reference approaches to SSSP are the Dijkstra's [34] and Bellman-Ford's [35, 36] algorithms. The Dijkstra's algorithm, by utilizing a priority queue where one vertex is processed at a time, is the most efficient, with a computational complexity almost linear to the number of vertices ($O(|V| \log |V| + |E|)$). Nevertheless, in several application domains, where the modelled data maps to very large graphs involving millions of vertices, any Dijkstra's sequential implementation becomes impractical. In addition, since the algorithm requires many iterations and each iteration is based on the ordering of previously computed results, it is poorly suited for parallelization.

On the other hand, the Bellman-Ford's algorithm relies on an iterative process over all edge connections, which updates the vertices continuously until final distances converge. Even though it is less efficient than Dijkstra's ($O(|V||E|)$), it is well suited to parallelization [37].

In the context of parallel implementations for GPUs, where the energy and power consumption is becoming a constraint in addition to performance [38], an ideal solution to SSSP would provide both the performance of the Bellman-Ford's and the work efficiency of the Dijkstra's algorithms. Some work has been recently done to analyse the spectrum between massive parallelism and efficiency, and different parallel solutions for GPUs have been proposed to implement parallel-friendly and work-efficient methods to solve SSSP [39]. Experimental results confirmed that these trade-off methods provide a fair speedup by doing much less work than traditional Bellman-Ford methods while adding only a modest amount of extra work over serial methods.

On the other hand, all these solutions as well as Dijkstra's implementations, do not work in graphs with negative weights [1]. Indeed, the Bellman-Ford algorithm is the only solution that can be also applied in application domains where the modelled data maps on graphs with negative weights, such as, power allocation in wireless sensor networks [40, 41], systems biology [42], and regenerative braking energy for railway vehicles [43].

3.1. The SSSP Implementations for GPUs

The Dijkstra's and Bellman-Ford's algorithms span a parallel vs. efficiency spectrum. Dijkstra's allows the most efficient ($O(V \log V + E)$) sequential implementations [44, 45] but exposes no parallelism across vertices. Indeed, the solutions proposed to parallelize the Dijkstra's algorithm for GPUs have shown to be asymptotically less efficient than the fastest CPU implementations [46, 47]. On the other hand, at the cost of a lower efficiency ($O(VE)$), the Bellman-Ford's algorithm has shown to be more easily parallelizable for GPUs, by providing

330 speedups up to two orders of magnitude with respect to the sequential counterpart [14, 37].

Meyer and Sanders [48] proposed the Δ-stepping algorithm, a trade-off between the two extremes of Dijkstra and Bellman-Ford. The algorithm involves a tunable parameter Δ, whereby setting $\Delta = 1$ yields a variant of Dijsktra's
335 algorithm, while setting $\Delta = \infty$ yields the Bellman-Ford algorithm. By varying Δ in the range $[1, \infty]$, we get a spectrum of algorithms with varying degrees of processing time and parallelism.

Meyer and Sanders [48] show that a value of $\Delta = \Theta(1/d)$, where d is the degree, gives a good tradeoff between work-efficiency and parallelism. In the
340 context of GPU, Davidson et al. [39] selects a similar heuristic, $\Delta = cw/d$, where d is the average degree in the graph, w is the average edge weight, and c is the warp width (32 on our GPUs).

Crobak et al. [49] and Chakaravarthy et al. [50] presented two different solutions to efficiently expose parallelism of this algorithm on the massively
345 multi-threaded shared memory system IBM Blue Gene/Q.

Parallel SSSP algorithms for multi-core CPUs have been also proposed by Kelley and Schardl [51], who presented a parallel implementation of Gabow's scaling algorithm [52] that outperforms Dijkstra's on random graphs. Shun and Blelloch [53] presented a Bellman-Ford's scalable parallel implementation for
350 CPUs on a 40-core machine. Recently, several packages have been developed for processing large graphs on parallel architectures including the parallel *Boost* graph library [54], Pregel [55] and Pegasus [56].

In the context of GPUs, Davidson et al. [39] proposed three different work-efficient solutions for the SSSP problem. The first two, *Near-Far Pile* and
355 *Workfront Sweep*, are the most representative implementations at state of the art. Workfront Sweep implements a queue-based Bellman-Ford algorithm that reduces redundant work due to duplicate vertices during the frontier propagation. Such a fast graph traversal method relies on the merge path algorithm [22], which equally assigns the outgoing edges of the frontier to the GPU threads
360 at each algorithm iteration. Near-Far Pile refines the Workfront Sweep strategy by adopting two queues similarly to the Δ-Stepping algorithm. Davidson et al. [39] also propose the *bucketing* method to implement the Δ-Stepping algorithm. Δ-Stepping algorithm is not well suited for SIMD architectures as it requires dynamic data structures for buckets. However, the authors provide
365 an algorithm implementation based on sorting that, at each step, emulates the bucket structure. The Bucketing and Near-Far Pile strategies heavily reduce the amount of redundant work with respect to the Workfront Sweep method but, at the same time, they introduce overhead for handling more complex data structure (i.e., frontier queue). These strategies are less efficient than the se-
370 quential implementation on graphs with large diameter since they suffer from thread under-utilization caused by such unbalanced graphs.

Algorithm 4 Sequential Bellman-Ford Algorithm

INITIALIZE(G, s)
for all edges $(u, v) \in E(G)$ **do**
  RELAX (u, v, w)
end

3.2. H-BF: an Efficient Implementation of the Bellman-Ford's algorithm

Given a graph $G(V, E)$, a source vertex s and a weight function $w : E \to \mathbb{R}$, the Bellman-Ford algorithm visits G and finds the shortest path to reach every 375 vertex of V from source s. Algorithm 4 summarizes the original sequential algorithm, where the *Relax* procedure of an edge (u, v) with weight w verifies whether, starting from u, it is possible to improve the approximate (*tentative*) distance to v (which we call $d(v)$) found in any previous algorithm iteration. The relax procedure can be summarized as follows:

380 EDGE RELAX:

if $d(u) + w < d(v)$ **then**
  $d(v) = d(u) + w$

The algorithm, whose asymptotic time complexity is $O(|V||E|)$, updates the 385 distance value of each vertex continuously until final distances converge.

H-BF [57] is a parallel implementation of the Bellman-Ford algorithm based on frontier propagation. Differently from all the approaches in literature, H-BF implements several techniques to improve the algorithm performance and, at the same time, to reduce the useless work done for solving SSSP involved 390 by the parallelization process. H-BF implements such techniques by exploiting the features of the most recent GPU architectures such as dynamic parallelism, warp-shuffle, read-only cache, and 64-bit atomic instructions.

The complexity of a SSSP algorithm is strictly related to the number of *relax* operations. The Bellman-Ford algorithm performs a higher number of *relax* 395 operations than Dijkstra or Δ-stepping algorithms while, on the other hand, it has a simple and lightweight management of the data structures. The *relax* operation is the most expensive in the Bellman-Ford algorithm and, in particular, in a parallel implementation, each *relax* involves an atomic instruction for handling race conditions, which takes much more time than a common memory 400 access.

To optimize the number of relax operations, H-BF implements the graph visit by exploiting the concept of *frontier*. For this problem, the frontier, F, is a FIFO queue that, at each algorithm iteration, contains *active vertices*, i.e., all and only vertices whose tentative distance has been modified and, thus, that 405 must be considered for the relax procedure at the next iteration. Given a graph G and a source vertex s, the parallel frontier-based algorithm can be summarized reported in Algorithm 5, where *adj[u]* returns the neighbours of vertex u. Figure 6 shows an example of the basic algorithm iterations starting from vertex

Algorithm 5 Frontier-based Bellman-Ford Algorithm

INITIALIZE(G, s)
$F \leftarrow \{s\}$
while $F \neq \emptyset$ **do**
  $u \leftarrow$ DEQUEUE(F)
  for all vertices $v \in adj\,[u]$ **do**
    if $d(u) + w < d(v)$ **then**
      $d(v) = d(u) + w$
      ENQUEUE(F, v)
  end
end

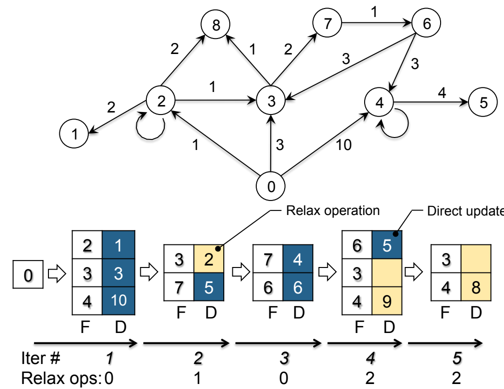

FIG. 6: *Example of the basic algorithm iterations starting from vertex "0"*

"0", where F is the active vertex queue and D is the corresponding data structure containing the tentative distances. The example shows, for each algorithm iteration, the dequeue of each vertex form the frontier, the corresponding relax operations, i.e., the distance updating for each vertex (if necessary), and the vertex enqueues in the new frontier. In the example, the algorithm converges in a total of 5 relax operations over 5 iterations.

The frontier structure is similar to that applied for implementing the parallel breadth-first search (BFS) presented in paragraph 2.1. The main difference from BFS is the number of times a vertex can be inserted in the queue. In BFS, a vertex can be inserted in such a queue only once, while, in the Bellman-Ford implementation, a vertex can be inserted $O(|E|)$ times in the worst case.

Figure 7 summarizes the speedup of the different implementations with respect to the sequential frontier-based Bellman-Ford implementation. The results show how H-BF outperforms all the other implementations in every graph. The speedup on graphs with very high diameter (left-most side of the figure) is quite low for every parallel implementation. This is due to the very low degree of par-

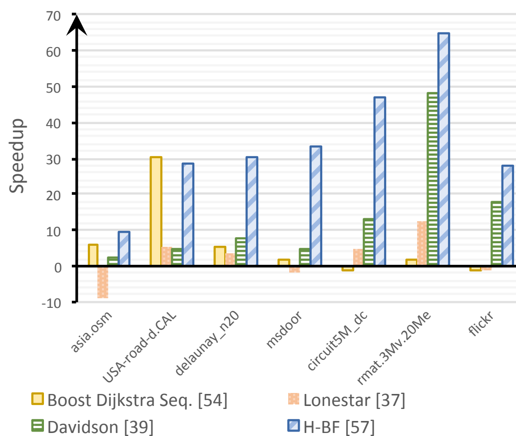

FIG. 7: *Comparison of speedups*

allelism for propagating the frontier in such graph typology. In these graphs, H-BF is the only parallel implementation that outperforms the Boost Dijkstra solution in *asia.osm*, while it preserves comparable performance in *USA-road.d-CAL*. On the other hand, the sequential Boost Dijkstra implementation largely outperforms all the other parallel solutions in literature.

H-BF provides the best performance (time and MTEPS) on the graphs in the right-most side of Figure 7. H-BF provides high speedup also in *rmat.3Mv.20Me* and *flickr*, which are graphs largely unbalanced. This underlines the effectiveness of H-BF to deal with such an unbalancing problem in traversing graphs. The optimization based on the *64-bit atomic* instruction strongly impacts on performance for graphs with small diameters. This is due to the fact that such graph visits are characterized by a rapid grow of the frontier, which implies a high number of duplicate vertices. The edge classification technique implemented in H-BF successfully applies to the majority of the graphs. In particular, *asia.osm* has a high number of vertices with in-degree equal to one, while in *msdoor* and *circuit5M_dc* each vertex has a self-loop. Scale-free graphs (e.g., *rmat.3Mv.20Me* and *flickr*) are generally characterized by a high number of vertices with low out-degree.

4. The All-Pair Shortest Path (APSP) problem

APSP is a fundamental problem in computer science that finds application in different fields such as transportation, robotics, network routing, and VLSI design. The problem is to find paths of minimum weight between all pairs of vertices in graphs with weighted edges. The common approaches to solve the APSP problem rely on iterating the SSSP algorithm from all vertices (*Johnson's algorithm*), *matrix multiplication*, and *Floyd-Warshall's algorithm*.

The Johnson's algorithm performs the APSP in two steps. First, it detects the negative cycles by applying the Bellman-Ford's algorithm and, than, it runs

Algorithm 6 Repeated Squaring APSP Algorithm

```
D^0 = W
for L = 1 to log V do
    for i = 0 to V do
        for j = 0 to V do
            D^L[i, j] = ∞
            for k = 0 to V do
                D^L[i, j] = min(D^{L−1}[i, j], D^{L−1}[i, k] + W[k, j])
            end
        end
    end
end
```

the Dijsktra's algorithm from all vertices. This approach has $O(|V|^2 \log |V| + |V||E|)$ time complexity and it is suitable only for sparse graphs. The second approach applies the matrix multiplication over min, plus semiring to compute the APSP in $O(|V|^3 \log |V|)$. The matrix multiplication method derives from the following recursive procedure. Let w_{ij} be the weight of edge (i, j), $w_{ii} = 0$ and d_{ij}^{ℓ} be the shortest path from i to j using ℓ or fewer edges, we compute d_{ij}^{ℓ} by using the recursive definition:

$$\begin{cases} d_{ij}^{0} = w_{ij} \\ d_{ij}^{\ell} = \min\left\{ d_{ij}^{\ell-1}, \min\limits_{1 \le k \le n} d_{ik}^{\ell-1} + w_{kj} \right\} \end{cases} \rightarrow d_{ij}^{\ell} = \min_{1 \le k \le n} \left\{ d_{ik}^{\ell-1} + w_{kj} \right\}$$

450 We note that making the substitutions min $\rightarrow$ + and + $\rightarrow$ $\cdot$ the definition is equivalent to the matrix multiplication procedure. Algorithm 6 reports the pseudocode.

Finally, the Floyd-Warshall's algorithm, which is the standard approach for the all pair shortest path problem in case of edges with negative weights, and 455 does not suffer from performance degradation for dense graphs. The algorithm has $O(|V|^3)$ time complexity and requires $O(|V|^2)$ memory space.

Let $G = (V, E)$ be a weighted graph with an edge-weight function $w : E \rightarrow \mathbb{R}$ and $W = w(i, j)$ representing the weighted matrix, the pseudocode of the algorithm is shown in Algorithm 7.

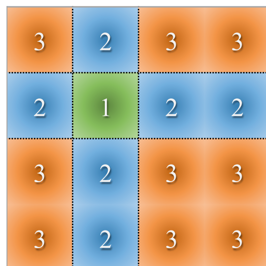

FIG. 8: *Blocked Floyd-Warshall algorithm. The numbers indicate the computation order of each tile.*

Algorithm 7 Floyd-Warshall

$D^0 = W$
for $k = 0$ to V **do**
  for $i = 0$ to V **do**
    for $j = 0$ to V **do**
      $D^k[i, j] = min(D^{k-1}[i, j], D^{k-1}[i, k] + D^{k-1}[k, j])$
    end
  end
end

460 *4.1. The APSP Implementations for GPUs*

The first GPU solution for the APSP problem has been proposed by Harish and Narayanan [25], which use their parallel SSSP algorithm from all vertices of the graph. Also Ortega et al. [58] resolve the APSP problem in the same way, by proposing a highly tunable GPU implementation of the Dijkstra's algorithm.
465 The most important idea, which gave the basis for a subsequent efficient GPU implementations of the Floyd-Warshall's algorithm has been proposed by Venkataraman et al. [3] in the context of multi-core CPUs. The proposed solution takes advantage of the cache utilization. It first partitions the graph matrix into multiple tiles that fit in cache and, then, it iterates on each tile
470 multiple times. In particular, such a blocked Floyd-Warshall's algorithm is composed by three main phases (see Figure 8).

1. The computation in each iteration starts from a tile in the diagonal of the matrix, from the upper left to the bottom right. Each tile in the diagonal is independent from the rest of the matrix and can be processed in-place.

475 2. In the second phase, all tiles that are in the same row and in the same column of the independent tiles are computed in parallel. All tiles in this phase are dependent only from itself and from the independent tiles.

3. In the third phase, all remaining tiles are dependent from itself and from

the main row and the main column that have been computed in the previous phase.

The blocked Floyd-Warshall's algorithm has been implemented for GPU architetures by Katz and Kider [4], which strongly exploit the shared memory as local cache. Lund et al. [5] improved such a GPU implementation by optimizing the use of registers and by taking advantage of memory coalescing. Buluç et al. [6] presented a recursive formulation of the APSP based on the Gaussian elimination (LU) and matrix multiplication with $O(|V|^3)$ complexity, which exposes a good memory locality.

Later, Harish et al. [59] revisited the APSP algorithm based on matrix multiplication, and they presented two improvements: *streaming blocks* and *lazy minimum evaluation*. The streaming block optimization describes a method to partition the adjacency matrix and to efficiently transfer each partition to the device through asynchronous read and write operations. The second optimization aims at decreasing the arithmetic computation by avoiding the minimum operation when one operand is set to infinite. The presented algorithm achieves a speedup from 5 to 10 over the Katz and Kider's algorithm. Nevertheless, it is slower than the Gaussian elimination method of Buluç et al. On the other hand, they showed that their algorithm is more scalable, and that the optimization of the lazy minimum evaluation is not orthogonal to the Gaussian elimination method.

Tran et al. [9] proposed an alternative algorithm based on matrix multiplication and on the *repeated squaring* technique (Algorithm 6). It outperforms the base Floyd-Warshall's algorithm when the graph matrix exceeds the GPU memory.

Matsumoto et al. [7] proposed a hybrid CPU-GPU based on OpenCL, which combines the blocked Floyd-Warshall's algorithm for a coarse-grained partition of the graph matrix and the matrix multiplication as a main procedure.

Finally, Djidjev et al. [8] proposed an efficient implementation of APSP on multiple GPUs for graphs that have good *separators*.

5. Load balancing and memory accesses: issues and management techniques

Load unbalancing and non-coalesced memory accesses are the main problems to deal with when implementing any graph algorithm for GPUs. They are caused by the non-homogeneity of real graphs. Different techniques have been presented in the literatures to decompose and map the graph algorithm workload to threads [25, 18, 15, 16, 60, 61, 62]. 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 graphs. The simplest solutions [25, 18] best apply to very regular workloads while they cause strong unbalancing and, as a consequence, lost of performance in case of irregular workloads. More complex solutions [15, 16, 60, 61, 62] 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.

In general, the techniques for decomposing and mapping a workload to GPU threads for graph applications rely on the *prefix-sum* data structure[3] [16]. Given a workload to be allocated (e.g., a set of graph vertices or edges) over the GPU threads, prefix-sum calculates the offsets to be used by the threads to access to the corresponding *work-units* (fine-grained mapping) or block of work-units, which we call *work-items* (coarse-grained mapping). All these decomposition and mapping techniques can be organized in three classes: *Static mapping*, *semi-dynamic mapping*, and *dynamic mapping*.

5.1. Static mapping techniques

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

5.1.1. Work-items to threads

It represents the simplest and fastest mapping approach by which each work-item is mapped to a single thread [25]. Fig. 9(a) shows an example, in which eight items are assigned to a corresponding number of threads. For the sake of clarity, only four threads per warp have been considered in the example to underline two levels of possible unbalancing of this technique. First, irregular (i.e., unbalanced) work-items mapped to threads of the same warp lead the warp threads to be in idle state (i.e., branch divergence). t_1, t_3, and t_0 of $warp_0$ in Fig. 9(a) are an example. Then, irregular work-items lead to whole warps to be in idle state (e.g., $warp_0$ w.r.t. $warp_1$ in 9(a)). As a third level of unbalancing, this technique can lead to whole blocks of threads to be in idle state.

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

[3]The prefix-sum array is generated, depending on the mapping technique, in a preprocessing phase [63], at run-time if the workload changes at every iteration [16, 15], or it could be already part of the problem [64].

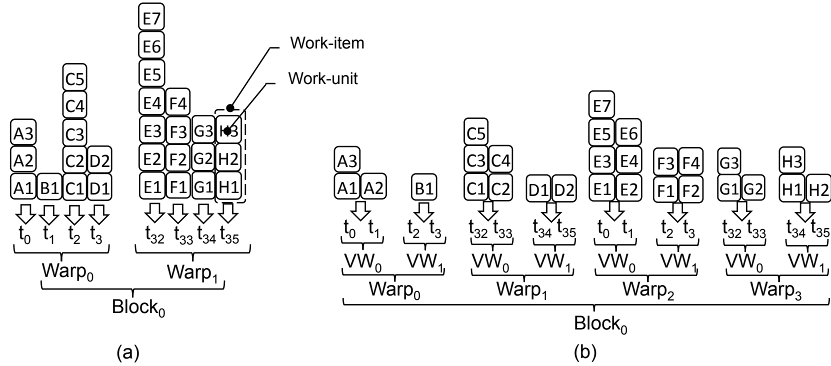

FIG. 9: *Example of static mapping techniques: (a) Work-items to threads, and (b) Virtual warps*

5.1.2. 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 [18]. Fig. 9(b) shows an example in which the chunks correspond to the work-items and, for the sake of clarity, the virtual warps have size equal to two threads. Virtual warps allow the workload assigned to threads of the same group to be almost equal and, as a consequence, it allows reducing branch divergence. In addition, this technique improves the coalescing of memory accesses since more threads of a virtual warp access to adjacent addresses in global memory (e.g., t_0, t_1 of $Warp_2$ in Fig. 9(b)). These improvements are proportional to the virtual warp size. Increasing the warp size leads to reducing branch divergence and better coalescing the work-unit accesses in global memory. Nevertheless, virtual warps have several limitations. First, the maximum size of virtual warps is limited by the number of available threads in the device. Given the number of work-items and a virtual warp size, the required number of threads is expressed as follows:

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

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

The algorithm run by each thread to access the corresponding work-units is summarized as in Algorithm 8, where VW_INDEX and LANE_OFFSET are the

Algorithm 8 Virtual Warp Load Balancing

```
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
```

585 virtual warp index and offset for the thread (e.g., VW_0, and 0 for t_0 in the example of Fig. 9(b)), INIT represents the starting work-unit id, and the *for* cycle represents the accesses of the thread to the assigned work-units (e.g., A_1, A_3 for t_0 and A_2 for t_1)..

5.2. Semi-dynamic mapping techniques

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

5.2.1. Dynamic Virtual Warps + Dynamic Parallelism

This technique has been introduced in [15] and relies on two main strategies. First, it implements a virtual warp strategy in which the virtual warp size 595 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 600 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 paragraph 5.1.2), 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-605 items (e.g., in datasets representing scale free networks or graphs with power-law distribution in general).

On the other hand, dynamic parallelism, which exploits the most advanced features of the GPU architectures (e.g., from NVIDIA Kepler on) [30] allows recursion to be implemented in the kernels and, thus, threads and thread blocks 610 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 [15].

To overcome these limitations, dynamic virtual warps and dynamic paral-615 lelism are combined into a single mapping strategy and applied alternatively at run time. The strategy applies dynamic parallelism to the work-items having size greater than a threshold, while it applies dynamic virtual warps to the

Algorithm 9 Strip-Mined Gathering Algorithm

```
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
```

others. It best applies to applications with few and strongly unbalanced work-items that may vary at run time (e.g., applications for sparse graph traversal). This technique guarantees load balancing among threads of the same warps and among warps. It does not guarantee balancing among blocks.

5.2.2. CTA+Warp+Scan

In the context of graph traversal, Merrill et al. [16] 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 (see Algorithm 9) of such a first step (which is called *strip-mined gathering*) is run by each thread (Th_{ID}).

   In the pseudo-code, row 3 implements the non-deterministic ordering (based on iterative match/winning among threads), rows 5-8 calculate information on the work-item to be copied in shared memory, while rows 10-14 implement the item partitioning for the CTA. This phase introduces sensible overhead for the two CTA synchronizations and, rows 5-8 are run by one thread only.

2. In the second step, the strip-mined gathering is run with a lower threshold ($WARP_{TH}$) and at warp level. That is, it targets smaller work-items and a cooperative thread array consists of threads of the same warp. This allows avoiding any synchronization among threads (as they are implicitly synchronized in SIMD-like fashion in the warp) and addressing work-items with sizes proportional to the warp size.

3. In the third step the remaining *work-items* are processed by all block threads. The algorithm computes a block-wide *prefix-sum* on the work-items and stores the resulting prefix-sum array in the shared memory. Finally, all threads of the block get use of such an array to access to the corresponding work-unit. If the array size exceeds the shared memory space, the algorithm iterates.

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

5.3. Dynamic mapping techniques

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

5.3.1. 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. This technique provides perfect balancing among threads (i.e., one work-unit is mapped to one thread), warps and blocks of threads. Nevertheless, the large size of the prefix-sum array involves an arithmetic intensive computation (i.e., $\#threads \times binarysearch()$) and all the accesses performed by the threads to solve the mapping very scattered. This often eludes the benefit of the provided perfect balancing.

5.3.2. Local Warp Search

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

In the context of graph traversal, this approach has been further improved by exploiting data locality in registers [15]. 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 [30].

In general, the local warp search strategy provides a very fast work-units to threads mapping and guarantees coalesced accesses to both the prefix-sum array and work-units in global memory. On the other hand, since the sum of work units included in each chunk of prefix-sum array is greater than the warp size, the binary search on the shared memory (or registers for the enhanced version

for Kepler) is repeated until all work-units are processed. This leads to more work-units to be mapped to the same thread. Indeed, although this technique guarantees a fair balancing among threads of the same warp, it suffers from work unbalance between different warps since the sum of work-units for each warp can be not uniform in general. For the same reason, it does not guarantee balancing among blocks of threads.

5.3.3. Block Search

To deal with the local warp search limitations, Davidson et al. [60] introduced the block search strategy through *cooperative blocks*. Instead of warps performing 32-element loads, in this strategy each block of threads loads a *maxi chunk* of prefix-sum elements from the global to the shared memory, where the maxi chunk is as large as the available space in shared memory for the block. The maxi chunk size is equal for all the blocks. Each maxi chunk is then partitioned by considering the amount of work-units included and the number of threads per block. Finally, each block thread performs only one binary search to find the corresponding slot. With the block search strategy, all the units included in a slot are mapped to the same thread. This leads to several advantages. First, all the threads of a block are perfectly balanced. The binary searches are performed in shared memory and the overall amount of searches is sensibly reduced (i.e., they are equal to the block size). Nevertheless, this strategy does not guarantee balancing among different blocks. This is due to the fact that the maxi chunk size is equal for all the blocks, but the chunks can include a different amount of work-units. In addition, this strategy does not guarantee memory coalescing among threads when they access the assigned work-units. Finally, this strategy cannot exploit advanced features for intra-warp communication and synchronization among threads, such as, warp shuffle instructions etc.

5.3.4. Two-phase Search

Davidson et al. [60], Green et al [61] and Baxter [62] proposed three equivalent methods to deal with the inter-block load unbalancing. All the methods rely on two phases: *partitioning* and *expansion*.

First, the whole prefix-sum array is partitioned into *balanced* chunks, i.e., chunks that point to the same amount of work-units. Such an amount is fixed as the biggest multiple of the block size that fits in the shared memory. As an example, considering blocks of 128 threads, two prefix-sum chunks pointing to $128 \times K$ units, and 1300 slots in shared memory, K is set to 10. The chunk size may differ among blocks. 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.

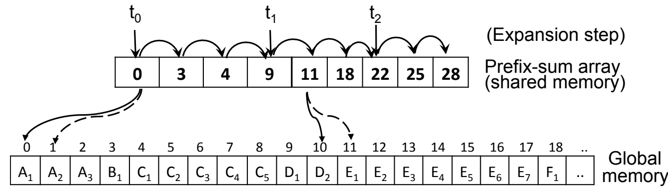

FIG. 10: *Example of expansion phase in the two-phase strategy (10 work-units per thread)*

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 paragraphs). 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. 10 shows an example of expansion phase, in which three threads (t_0, t_1, and t_2) of the same warp access to the local chunk of prefix-sum array to get the corresponding starting point of assigned work-unit. Then, they sequentially access the corresponding K assigned units ($A_1 - D_1$ for t_0, $D_2 - F_2$ for t_1, etc.) in global memory.

In conclusion, the two-phase search strategy allows the workload among threads, warps, and blocks to be perfectly balanced at the cost of two series of binary searches. The first is run in global memory for the partitioning phase, while the second, which most affects the overall performance, is run in shared memory for the expansion phase.

The number of binary searches for partitioning is proportional to the K parameter. High values of K involves less and bigger chunks to be partitioned and, as a consequence, less steps for each binary search. Nevertheless, the main problem of such a dynamic mapping technique is that the partitioning phase leads to very scattered memory accesses of the threads to the corresponding work-units (see lower side of Fig. 10). Such a problem worsens by increasing the K value.

5.4. The multi-phase search technique

As an improvement of the dynamic load balancing techniques presented above, [65] proposes the *multi-phase mapping* strategy, which aims at exploiting the balancing advantages of the two-phase algorithms while overcoming the limitations concerning the scattered memory accesses. It consists of two main contributions: Coalesced expansion and Iterated search.

5.5. Coalesced Expansion

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

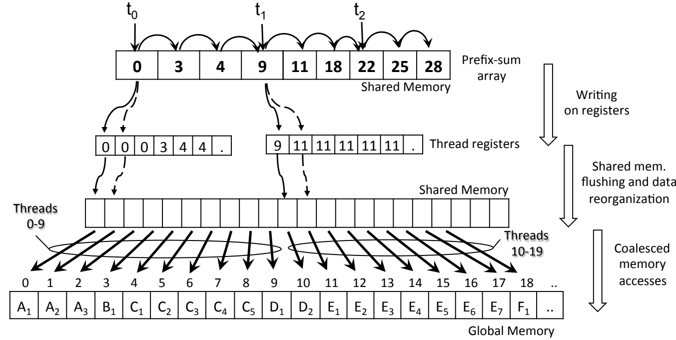

FIG. 11: *Overview of the coalesced expansion optimization (10 work-units per thread)*

The technique applies for both reading and writing accesses to the global memory as for the two-phase approach. For the sake of clarity, we consider *writing* accesses in the following.

1. Instead of sequentially writing on the work-units in global memory, each thread sequentially writes a small amount of work-units in the local registers. Fig. 11 shows an example. The amount of units is limited by the available number of free registers.

2. After a thread block synchronization, the local shared memory is flushed and the threads move and reorder the work-unit array from the registers to the shared memory.

3. Finally, the whole warp of threads cooperates for a coalesced transaction of the reordered data into the global memory. It is important to note that this step does not require any synchronization since each warp executes independently on the own slot of shared memory.

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

5.6. Iterated Searches

The shared memory size and the size of thread blocks play an important role in the coalesced expansion phase. The bigger the block size, the shorter the partition array stored in shared memory. On the other hand, the bigger the block size, the more the synchronization overhead among the block warps, and the more the binary search steps performed by each thread (see final considerations of the Two-phase search in paragraph 5.3.4).

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

1. In the partition phase, the prefix sum array is partitioned into balanced chunks. Differently from the two-phase search strategy, the size of such chunks is fixed as a multiple of the available space in shared memory:

$$Chunk_{size} = Block_{size} \times K \times IS$$

795 where $Block_{size} \times K$ represents the biggest number of work-units (i.e., a multiple of the block size) that fits in shared memory (as in the two-phase algorithm), while IS represents the *iteration factor*. The number of threads required in this step decreases linearly with IS.

2. Each block of threads loads from global to shared memory a chunk
800 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, i.e., it
805 sequentially writes an amount of work-units in the local registers. Such an amount is equal IS times larger than in the standard two-phase strategy.

5. The local shared memory is flushed and each thread moves a portion of the extended work-unit array from the registers to the shared memory. The portion size is equal to $Block_{size} \times K$. Then, the whole warp of threads
810 cooperates for a coalesced transaction of the reordered data into the global memory, as in the coalesced expansion phase. This step iterates IS times, until all the data stored in the registers has been processed.

With respect to the standard partitioning and expansion strategy, the iterated search optimization reduces the number of synchronization points by a
815 factor of $2 * IS$, avoids many block initializations, decreases the number of required threads, and maximizes the shared memory utilization during the loading of the prefix-sum values with more large consecutive intervals. Nevertheless, the required number of registers grows proportionally to the IS parameter. Considering that the maximum number of registers per thread is a fixed constraints
820 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.

Figures 12-15 summarize and compare the performance of each technique
825 over different graphs, each one having very different characteristics and structure. The results obtained with the *Direct Search* and *Block Search* techniques are much worse than the other techniques and, for the sake of clarity, have not been reported in the figures.

In the first benchmark (Fig. 12), as expected, *Work-items to threads* is the
830 most efficient balancing technique. This is due to the very regular workload and the small average work-item size. In this benchmark, any overhead for the

dynamic item-to-thread mapping may compromise the overall algorithm performance. However, *Multi-Phase Search* is the second most efficient technique. This underlines the reduced amount of overhead introduced by such a dynamic technique, which well applies also in case of very regular workloads.

In the *web-NotreDame* benchmark (Fig. 13), *Multi-Phase Search* is the most efficient technique and provides almost twice the performance with respect to the second best techniques (*Virtual Warps* and *Two-Phase*). On the other hand, *Virtual Warps* provides good performance if the virtual warp size is properly set, while it may sensibly worsen with wrongly-sized sizes. The virtual warp size has to be set statically. For the obtained results in these two benchmarks, we noticed that the optimal virtual warp size is proportional and follows approximately the average of work-item sizes.

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

Multi-Phase Search provides the best results also in the *circuit5M* benchmark (Fig. 14). In such a benchmark, the *CTA+Warp+Scan*, *Two-Phase Search*, and *Multi-Phase Search* dynamic techniques are one order of magnitude faster than the static-mapping techniques. In *web-Notredame* and in *circuit5M*, *Multi-Phase Search* shows the best results due to the low average (less than warp size) and high standard. deviation.

In the last benchmark, *kron_g500-logn20* (Fig. 15), *CTA+Warp+Scan* provides the best results, since the CTA and Warp phases are frequently activated and exploited. However the performance of *Multi-Phase* are comparable. *Dynamic Virtual Warps* and *Virtual Warps* provide similar performance. Indeed, these two techniques are very efficient on high-average datasets, since, with a thread group size of 32, they completely avoid the warp divergence. Finally, the *Dynamic Parallelism* feature provided by Kepler, implemented in the corresponding semi-dynamic technique, finds the best application only when the work-item sizes and their average are very large. In any case, all the dynamic load balancing techniques, and in particular the *Multi-Phase Search*, perform better without such a feature in all the analysed datasets.

References

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

[2] R. Sedgewick, K. Wayne, Algorithms 4th Edition, Addison-Wesley, 2011.

[3] G. Venkataraman, S. Sahni, S. Mukhopadhyaya, A blocked all-pairs shortest-paths algorithm, Journal of Experimental Algorithmics (JEA) 8 (2003) 2–2.

[4] G. J. Katz, J. T. Kider Jr, All-pairs shortest-paths for large graphs on the gpu, in: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS

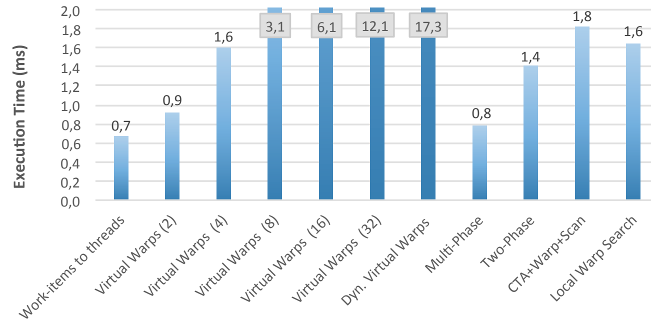

FIG. 12: *Comparison of execution time on the great-britain_osm dataset*

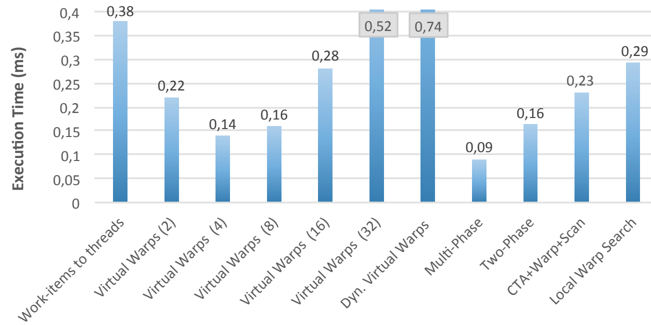

FIG. 13: *Comparison of execution time on the web-NotreDame dataset*

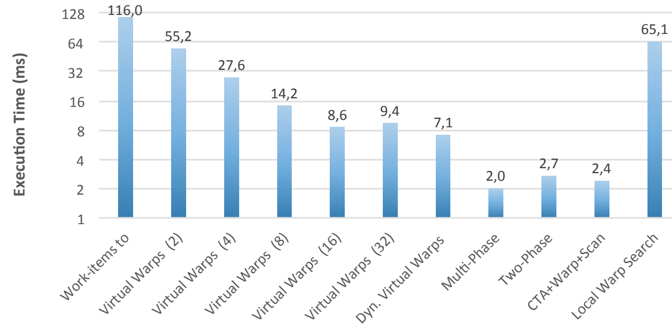

FIG. 14: *Comparison of execution time on the Circuit5M dataset*

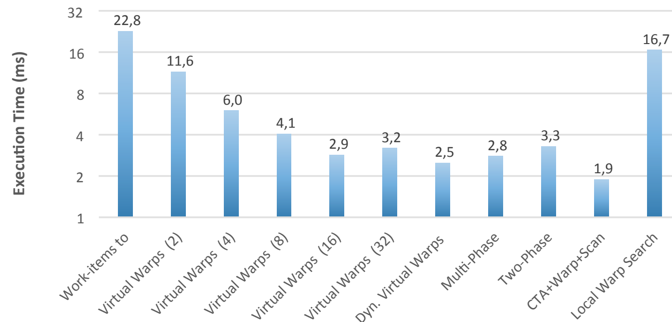

FIG. 15: *Comparison of execution time on the kron_g500-logn20*

symposium on Graphics hardware, Eurographics Association, 2008, pp.
875 47–55.

[5] B. Lund, J. W. Smith, A multi-stage cuda kernel for floyd-warshall, arXiv preprint arXiv:1001.4108.

[6] A. Buluç, J. R. Gilbert, C. Budak, Solving path problems on the gpu, Parallel Computing 36 (5) (2010) 241–253.

880 [7] K. Matsumoto, N. Nakasato, S. G. Sedukhin, Blocked all-pairs shortest paths algorithm for hybrid cpu-gpu system, in: High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on, IEEE, 2011, pp. 145–152.

[8] H. Djidjev, S. Thulasidasan, G. Chapuis, R. Andonov, D. Lavenier, Effi-
885 cient multi-gpu computation of all-pairs shortest paths, in: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, IEEE, 2014, pp. 360–369.

[9] Q.-N. Tran, Designing efficient many-core parallel algorithms for all-pairs shortest-paths using cuda, in: Information Technology: New Generations
890 (ITNG), 2010 Seventh International Conference on, IEEE, 2010, pp. 7–12.

[10] C. NVIDIA, Cuda api reference manual (2015).

[11] N. Bell, M. Garland, Efficient sparse matrix-vector multiplication on cuda, Tech. rep., Nvidia Technical Report NVR-2008-004, Nvidia Corporation (2008).

895 [12] N. Bell, M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, 2009, p. 18.

[13] C. Nvidia, Nvidia cuda c programming guide.

[14] P. Harish, P. Narayanan, Accelerating large graph algorithms on the gpu using cuda, in: High performance computing–HiPC 2007, Springer, 2007, pp. 197–208.

[15] F. Busato, N. Bombieri, BFS-4K: an efficient implementation of BFS for kepler GPU architectures, IEEE Transactions on Parallel Distributed Systems 26 (7) (2015) 1826–1838.

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

[17] L. Luo, M. Wong, W.-m. Hwu, An effective gpu implementation of breadth-first search, in: Proceedings of the 47th Design Automation Conference, DAC '10, 2010, pp. 52–55.

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

[19] H.-V. Dang, B. Schmidt, The sliced coo format for sparse matrix-vector multiplication on cuda-enabled gpus, Procedia Computer Science 9 (2012) 57–66.

[20] J. Siegel, J. Ributzka, X. Li, Cuda memory optimizations for large data-structures in the gravit simulator, Journal of Algorithms & Computational Technology 5 (2) (2011) 341–362.

[21] Y. Jia, V. Lu, J. Hoberock, M. Garland, J. C. Hart, Edge vs. node parallelism for graph centrality metrics, GPU Computing Gems: Jade Edition (2011) 15–28.

[22] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, Y. Birk, Merge path-parallel merging made simple, in: Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International, IEEE, 2012, pp. 1611–1618.

[23] A. McLaughlin, D. Bader, et al., Revisiting edge and node parallelism for dynamic gpu graph analytics, in: Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International, IEEE, 2014, pp. 1396–1406.

[24] G. Singla, A. Tiwari, D. P. Singh, New approach for graph algorithms on gpu using cuda, International Journal of Computer Applications.

[25] P. Harish, P. J. Narayanan, Accelerating large graph algorithms on the gpu using cuda, in: Proceedings of the 14th International Conference on High Performance Computing, HiPC'07, 2007, pp. 197–208.

[26] 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.

[27] S. Xiao, W. chun Feng, Inter-block gpu communication via fast barrier synchronization, Tech. rep., Dept. of Computer Science Virginia Tech (2009).

[28] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, J. D. Owens, Gunrock: A high-performance graph processing library on the gpu, in: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2015, pp. 265–266.

[29] T. Hiragushi, D. Takahashi, Efficient hybrid breadth-first search on gpus, in: Algorithms and Architectures for Parallel Processing, Springer, 2013, pp. 40–50.

[30] NVIDIA, Kepler gk110, www.nvidia.com/content/PDF/kepler/NV_DS_Tesla_KCompute_Arch_May_2012_LR.pdf.

[31] NVIDIA, Maxwell architecture, http://international.download.nvidia.com/geforce -com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF.

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

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

[34] E. W. Dijkstra, A note on two problems in connexion with graphs, NUMERISCHE MATHEMATIK 1 (1) (1959) 269–271.

[35] R. Bellman, On a routing problem, Quarterly of Applied Mathematics 16 (1) (1958) 87–90.

[36] L. R. Ford, Network flow theory, Santa Monica, Calif. : Rand Corp., 1956.

[37] M. Burtscher, R. Nasre, K. Pingali, A quantitative study of irregular programs on gpus, in: Workload Characterization (IISWC), 2012 IEEE International Symposium on, IEEE, 2012, pp. 141–151.

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

[39] A. Davidson, S. Baxter, M. Garland, J. Owens, Work-efficient parallel gpu methods for single-source shortest paths, 2014, pp. 349–359.

[40] X. Zhang, F. Yan, L. Tao, D. Sung, Optimal candidate set for opportunistic routing in asynchronous wireless sensor networks, 2014, pp. 1– 8.

[41] M. Saad, Joint optimal routing and power allocation for spectral efficiency in multihop wireless networks, IEEE Transactions on Wireless Communications 13 (5) (2014) 2530– 2539.

[42] S. Klamt, A. von Kamp, Computing paths and cycles in biological interaction graphs, BMC Bioinformatics 10 (6) (2014) 1–11.

[43] S. Lu, P. Weston, S. Hillmansen, H. Gooi, C. Roberts, Increasing the regenerative braking energy for railway vehicles, IEEE Transactions on Intelligent Transportation Systems 15 (181) (2009) 2506–2515.

[44] B. V. Cherkassky, A. V. Goldberg, T. Radzik, Shortest paths algorithms: Theory and experimental evaluation, Mathematical programming 73 (2) (1996) 129–174.

[45] F. B. Zhan, C. E. Noon, Shortest path algorithms: an evaluation using real road networks, Transportation Science 32 (1) (1998) 65–73.

[46] P. J. Martin, R. Torres, A. Gavilanes, Cuda solutions for the sssp problem, in: Proceedings of the 9th International Conference on Computational Science: Part I, ICCS '09, 2009, pp. 904–913.

[47] H. Ortega-Arranz, Y. Torres, D. Llanos, A. Gonzalez-Escribano, A new gpu-based approach to the shortest path problem, 2013, pp. 505–511.

[48] U. Meyer, P. Sanders, Δ-stepping: a parallelizable shortest path algorithm, Journal of Algorithms 49 (1) (2003) 114–152.

[49] J. R. Crobak, J. W. Berry, K. Madduri, D. A. Bader, Advanced shortest paths algorithms on a massively-multithreaded architecture, in: Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, IEEE, 2007, pp. 1–8.

[50] V. Chakaravarthy, F. Checconi, F. Petrini, Y. Sabharwal, Scalable single source shortest path algorithms for massively parallel systems, 2014, pp. 889–901.

[51] K. Kelley, T. B. Schardl, Parallel single-source shortest paths.

[52] H. N. Garbow, Scaling algorithms for network problems, Journal of Computer and System Sciences 31 (2) (1985) 148–168.

[53] J. Shun, G. E. Blelloch, Ligra: a lightweight graph processing framework for shared memory, in: ACM SIGPLAN Notices, Vol. 48, ACM, 2013, pp. 135–146.

[54] N. Edmonds, A. Breuer, D. Gregor, A. Lumsdaine, Single-source shortest paths with the parallel boost graph library, The Ninth DIMACS Implementation Challenge: The Shortest Path Problem, Piscataway, NJ (2006) 219–248.

[55] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, ACM, 2010, pp. 135–146.

[56] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, et al., Pegasus: A framework for mapping complex scientific workflows onto distributed systems, Scientific Programming 13 (3) (2005) 219–237.

[57] F. Busato, N. Bombieri, An efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures, IEEE Transactions on Parallel Distributed Systems PP (99) (2015) 1–13.

[58] H. Ortega-Arranz, Y. Torres, D. R. Llanos, A. Gonzalez-Escribano, A tuned, concurrent-kernel approach to speed up the apsp problem, in: Proc. 13th Int. Conf. Comput. Math. Methods Sci. Eng.(CMMSE), Citeseer, 2013, pp. 1114–1125.

[59] P. Harish, V. Vineet, P. Narayanan, Large graph algorithms for massively multithreaded architectures, International Institute of Information Technology Hyderabad, Tech. Rep. IIIT/TR/2009/74.

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

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

[62] Modern gpu library.
URL http://nvlabs.github.io/moderngpu/

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

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

[65] F. Busato, N. Bombieri, On the load balancing techniques for gpu applications based on prefix-scan, in: Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on, IEEE, 2015, pp. 88–95.