An efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures

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

Abstract—Finding the shortest paths from a single source to all other vertices is a common problem in graph analysis. The Bellman-Ford's algorithm is the solution that solves such a single-source shortest path (SSSP) problem and better applies to be parallelized for many-core architectures. Nevertheless, the high degree of parallelism is guaranteed at the cost of low work efficiency, which, compared to similar algorithms in literature (e.g., Dijkstra's) involves much more redundant work and a consequent waste of power consumption. This article presents a parallel implementation of the Bellman-Ford algorithm that exploits the architectural characteristics of recent GPU architectures (i.e., NVIDIA Kepler, Maxwell) to improve both performance and work efficiency. The article presents different optimizations to the implementation, which are oriented both to the algorithm and to the architecture. The experimental results show that the proposed implementation provides an average speedup of 5x higher than the existing most efficient parallel implementations for SSSP, that it works on graphs where those implementations cannot work or are inefficient (e.g., graphs with negative weight edges, sparse graphs), and that it sensibly reduces the redundant work caused by the parallelization process.

Index Terms—SSSP, Bellman-Ford, GPU, CUDA, Kepler.

1 Introduction

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 paths (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 [2] and Bellman-Ford's [3], [4] 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. Indeed, the parallel solutions proposed in literature for graphics processing units (GPUs) [5], [6] are asymptotically less efficient than the fastest CPU implementations.

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 [7].

In the context of parallel implementations for GPUs, where the energy and power consumption is becoming a constraint in addition to performance [8], 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 [9]. 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 [10], [11], systems biology [12], and regenerative braking energy for railway vehicles [13].

In addition, the most recent GPU architectures (e.g., NVIDIA Kepler GK110 [13] and Maxwell [14]), not only offer much higher processing power than the prior GPU generations, but, also, they provide new programming capability that allows improving the efficiency of the parallel implementations.

This article presents *H-BF*, a high-performance implementation of the Bellman-Ford algorithm for GPUs, which exploits the more advanced features of GPU architectures to improve the execution speedup with respect to any implementation at the state of the art for solving the SSSP problem. In particular, *H-BF* implements a

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

parallel version of the Bellman-Ford algorithm based on *frontiers* [1] and *active vertices* [30] with the aim of optimizing, besides the performance, the algorithm work efficiency. The article presents different optimizations implemented in *H-BF*, which are oriented both to the algorithm and to the architecture to underline the synergy of parallelism in the algorithm, programming and architecture.

Experimental results have been conducted on graphs of different sizes and characteristics to compare, firstly in terms of performance and, then, in terms of work efficiency, the *H-BF* implementation (which is available for download in *http://profs.sci.univr.it/~bombieri/H-BF/index.html*) with the most efficient sequential and parallel implementations at the state of the art of both Dijkstra's and Bellman-Ford's algorithms.

The article is organized as follows. Section 2 summarizes preliminaries concepts on CUDA, Kepler, Maxwell GPUs and Bellman-Ford's algorithm. Section 3 presents the related work. Sections 4 and 5 present the optimizations of the proposed approach oriented to the algorithm and GPU architecture, respectively. Section 6 reports the experimental results, while Section 7 is devoted to concluding remarks.

2 Background

This section summarizes preliminary concepts on CUDA, advanced GPU architectures, and Bellman-Ford algorithm.

2.1 CUDA, Kepler, and Maxwell GPUs

Compute Unified Device Architecture (CUDA) is a C library extension developed by NVIDIA to provide a programming interface to GPU devices [15]. 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*. 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 1-, 2-, or 3-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. Multiple threads of the same warp (i.e., half a warp) execute one common instruction at a time on different data (i.e., SIMD architecture). Two half-warps interleave in single instruction multiple threaded (SIMT architecture) to hide memory access latency.

In 2012, NVIDIA released the *Kepler GK110* architecture [16], 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 shows how dynamic parallelism can be efficiently used in the Bellman-Ford implementation.

Warp shuffle instructions are another new feature of Kepler used in this work. They implement very efficient communication among threads within a warp. With shuffle instructions, threads within a warp can directly access to other thread registers by skipping shared memory accesses. 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 introduced the *read-only data cache*. Such a separate cache (i.e., with separate memory pipe), which is available to each symmetric multiprocessor, SM, (generally 48KB per SM) allows improving performance through bandwidth-limited kernels. The implementation proposed in this work takes advantage of this feature to alleviate the L1 cache pressure during data loads from global memory .

Kepler architecture also expands the native support for *64-bit atomic operations* in global memory. This feature is used in this work to improve the work efficiency.

More recently, GPU architectures like NVIDIA *Maxwell* [14] have been released with the aim of improving energy efficiency, to be used in power-limited environments like notebooks and small form factor PCs. The design of the Maxwell-based GPU available in the commerce (GM107) retains and extends the same CUDA programming model as in previous architectures, such as Fermi and Kepler. That is, applications that follow the best practices (e.g., the optimizations proposed in this article) for the Kepler architecture should typically see speedups on the Maxwell architecture without any code changes. However, some fine-tuning is still possible. This mainly involves kernel configurations, since some architectural characteristics have been changed (e.g., the maximum number of thread blocks per SM has been increased from 16 to 32, the shared memory capacity has been increased). The optimization techniques presented in this article applies to the Kepler architecture onwards and, thus, also to the Maxwell one.

2.2 The Bellman-Ford's algorithm

Given a graph $G(V, E)$ (directed or undirected), a source vertex s and a weight function $w : E \rightarrow \mathbb{R}$, the Bellman-

Ford algorithm visits G and finds the shortest path to reach every vertex of V from source s. The pseudocode of the original sequential algorithm is the following:

Algorithm 1 BELLMAN-FORD'S ALGORITHM

```
for all vertices $u \in V(G)$ do
    $d(u) = \infty$
d(s) = 0
for all edges $(u, v) \in E(G)$ do
    RELAX $(u, v, w)$
```

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:

Algorithm 2 RELAX PROCEDURE

```
RELAX(u, v, w)
if $d(u) + w < d(v)$ then
    $d(v) = d(u) + w$
```

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

3 RELATED WORK

At the state of the art, the reference approaches to SSSP are the Dijkstra's [2] and Bellman-Ford's [3], [4] algorithms. These two classic algorithms span a parallel vs. efficiency spectrum. Dijkstra's allows the most efficient ($O(|V| \log |V| + |E|)$) sequential implementations [17], [18] 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 [5], [6]. On the other hand, at the cost of a lower efficiency ($O(|V||E|)$), the Bellman-Ford's algorithm has shown to be more easily parallelizable for GPUs, by providing speedups up to two orders of magnitude compared to the sequential counterpart [19], [7].

Meyer and Sanders [20] 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 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. Crobak et al. [21] and Chakaravarthy et al. [22] presented two different solutions to efficiently expose parallelism of this algorithm on the massively multi-threaded shared memory system IBM Blue Gene/Q.

Parallel SSSP algorithms for multi-core CPUs have been also proposed by Kelley and Schardl [23], who presented a parallel implementation of Gabow's scaling algorithm [24] that outperforms Dijkstra's on random graphs. Shun and Blelloch [25] presented a Bellman-Ford's scalable parallel implementation for 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 [26], Pregel [27] and Pegasus [28].

In the context of GPUs, Martin et al. [5] and Ortega et al. [6] proposed two different solutions to parallelize the Dijsktra's algorithm. Although both the solutions provide a good speedup in many cases, they have shown to be asymptotically less efficient than the fastest CPU implementations due to the intrinsic sequential nature of the Dijsktra's algorithm.

In contrast, Harish et al. [19] and Burtscher et al. [7] proposed two different parallel implementations of the Bellman-Ford's algorithm. Both the solutions always provide good speedups with respect to the sequential counterpart and, in any case, speedups higher than the Dijkstra's solutions. Nevertheless, they showed to have a poor work efficiency since they only target to performance.

Davidson et al. [9] proposed three different work-efficient solutions for the SSSP problem. *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 [29], which equally assigns the outgoing edges of the frontier to the GPU threads 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. [9] 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 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 compared 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 sequential implementation on graphs with large diameter since they suffer from thread under-utilization caused by such unbalanced graphs.

This article presents *H-BF*, a parallel implementation of the Bellman-Ford algorithm based on frontier propagation. Differently from all the approaches in literature, *H-BF* implements:

- An optimization technique by which, during the frontier propagation, the graph edges are classified and, depending on the class, they are properly handled to reduce the number of expensive Bellman-Ford basic iterations (i.e., *relax* operations).
- A duplicate removal strategy, which is based on 64-bit atomic instructions, to sensibly reduce the duplicate vertices in the frontier at each algorithm

iteration, with a consequent reduction of memory accesses and atomic operations.
- A technique to improve the memory coalescing through the use of cache modifiers and texture memory.
- A dynamic virtual warp strategy whereby the warp size is calibrated at each frontier propagation step to address the problem of workload imbalance.
- The dynamic parallelism, by which multi-block kernels are properly configured and invoked to manage the workload imbalance due to the difference of the vertex degrees.

In particular, the proposed implementation exploits the features of the most recent GPU architectures such as dynamic parallelism, warp-shuffle, read-only cache, and 64-bit atomic instructions to guarantee an efficient implementation of the characteristics listed above.

Algorithm 3 Frontier-based Bellman-Ford's algorithm

```
for all vertices $u \in V(G)$ do
    $d(u) = \infty$
$d(s) = 0$
$F_1 \leftarrow \{s\}$
$F_2 \leftarrow \emptyset$
while $F_1 \neq \emptyset$ do
    parallel for vertices $u \in F_1$ do
        $u \leftarrow$ DEQUEUE($F_1$)
        parallel for vertices $v \in adj[u]$ do
            if $d(u) + w < d(v)$ then
                $d(v) = d(u) + w$
                ENQUEUE($F_2, v$)
        end
    end
    SWAP($F_1, F_2$)
end
```

4 The Frontier-Based Algorithm and its Optimizations

The complexity of a SSSP algorithm is strictly related to the number of *relax* operations. The Bellman-Ford algorithm performs a number of *relax* operations higher than the 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 access.

To optimize the number of relax operations, *H-BF* implements the graph visiting by adopting the idea proposed in the sequential queue-based Bellman-Ford of Sedgewick et al. [30]. Such a sequential algorithm uses a FIFO data structure to keep track of *active vertices*, that is, all and only vertices whose tentative distance has been modified and, thus, that must be considered for the relax procedure at the next iteration. If $d(v)$ does not change during iteration i, there is no need to relax any edge outgoing from v in iteration $i+1$. As a consequence, v is not inserted in the queue to avoid useless computation.

Differently from Dijkstra's, the queue-based Bellman-Ford's algorithm does not rely on a priority queue and the vertex processing can be performed in any order. The parallel algorithm implemented in *H-BF* exploits the concept of *frontier* [1] rather than FIFO queue to visit the vertices concurrently. Given a graph G and a source vertex s, the parallel algorithm can be summarized as in Algorithm 3.

Considering two frontier data structures, F_1 and F_2, at each iteration i of the while loop, the algorithm concurrently extracts the vertices from F_1 and inserts all the *active* neighbours in F_2 for the next iteration step. Each iteration step concludes by swapping the F_2 contents (which will be the actual frontier at the next iteration step) in F_1. Fig. 1 shows an example of the basic algorithm iterations starting from vertex "0". For the sake of clarity, the figure only reports the actual frontier (F_1 of Algorithm 3, reported as F in Fig. 1) at each iteration step, and D as 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 23 relax operations over six iterations.

The parallel frontier-based Bellman-Ford's algorithm (Algorithm 3) preserves the semantics of the original Bellman-Ford's algorithm (Algorithm 1). The only difference between the sequential and parallel algorithms is that the first adopts a *queue structure* in which all nodes are stored and processed sequentially. The second adopts a *frontier structure* (as proposed by Cormen et al. [1]), in which all and only active nodes are processed in parallel. The parallel processing of active nodes does preserve the semantics of the algorithm. This is due to the fact that (i) each node processing is independent from the others, and (ii) including non-active nodes in the processing phase of any propagation step does not change the result (next frontier), as proved by Sedgewick et al. [30] and by Pape [31].

Frontier-based data structures have been similarly applied in literature for implementing parallel breadth-first search (BFS) visits [32], [33]. 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 proposed Bellman-Ford implementation, a vertex can be inserted $O(|E|)$ times in the worst case.

H-BF implements three main optimizations to improve both the performance and the work efficiency:

1) The *edge classification*. During each frontier propagation step, the edge outgoing the vertices of the frontier are classified and processed differently to

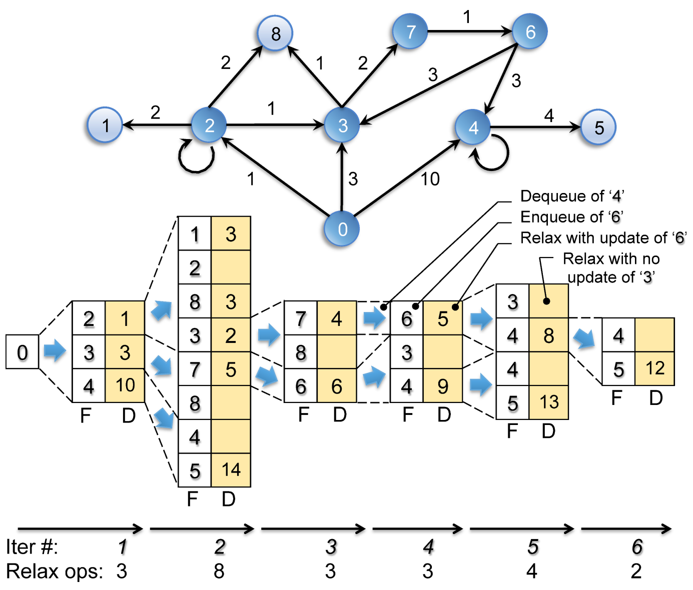


FIG 1: Example of the basic algorithm iterations starting from vertex "0"

simplify as much as possible the *relax* operations, as explained in Section 4.1.

2) The *duplicate removal*. It allows avoiding redundancy when processing duplicate vertices in the frontier propagation steps, as explained in Section 4.2.

4.1 The edge classification optimization

During the graph visit, the number of relax operations can be significantly reduced by observing the properties of the edges outgoing the active vertices in the frontier. In particular, given a vertex u and an outgoing edge $(u, adj[u])$, we identify four different classes to which the edge may belong. Depending on the class, the edge may involve operations lighter than the standard *relax*, with a consequent impact on performance:

1) *Self-loop class* ($(u, adj[u])$ edges where $u = adj[u]$). Since a self-loop cannot change the tentative distance of u, the relax operation can be avoided (see, for example, edges $(2, 2)$ and $(4, 4)$ in Figure 2). The efficiency improvement provided by the self-loop identification is proportional to the number of self-loops in the graph. It best applies to graphs where each vertex includes a self-loop (e.g., *msdoor* and *circuit* in the experimental results).
2) *Source edge class* ($(u, adj[u])$ edges where $u = s$). The relax operation for the source vertex is substituted with a direct update of the tentative distance for each source neighbour ($v \in adj[s]$) since, certainly, they have not been set previously. This optimization best applies to graphs with small diameter and, even more, when the source in such graphs is a high-degree vertex.
3) *In-degree edge class* ($(u, adj[u])$ edges where in-degree of $adj[u]$ is equal to 1). The vertices with in-degree equal to one (e.g., $(0, 2)$, $(2, 1)$, $(4, 5)$, $(7, 6)$, and $(3, 7)$ in Figure 2) are never visited concurrently and, thus, the atomic operations are avoided and replaced with a direct distance update.
4) *Out-degree edge class* ($(u, adj[u])$ edges where out-degree of $adj[u]$ is equal to 0). The vertices with out-degree equal to zero in directed graphs (e.g., 1, 5, and 8 in Figure 2) and equal to one in undirected graphs are ignored during the algorithm iterations (the relax operation and the enqueue into the next frontier are skipped). The correct distance is assigned at the end of the algorithm iterations without using atomic instructions. *H-BF* implements this optimization through an extra kernel, which is invoked after the main algorithm procedure. Such a kernel involves a negligible amount of computational work with respect to the (useless) relax operations performed for these edges in the standard approach.

Algorithm 4 EDGE CLASSIFICATION OPTIMIZATION

```
RELAX_OPT(u, v, w)

if  $u = v$  OR  out-degree(v) = 0  then
    skip
else if  $u = s$  OR  in-degree(v) = 1  then
    $d(v) = d(u) + w$
else
    ATOMICMIN($d(v)$, $d(u) + w$)
```

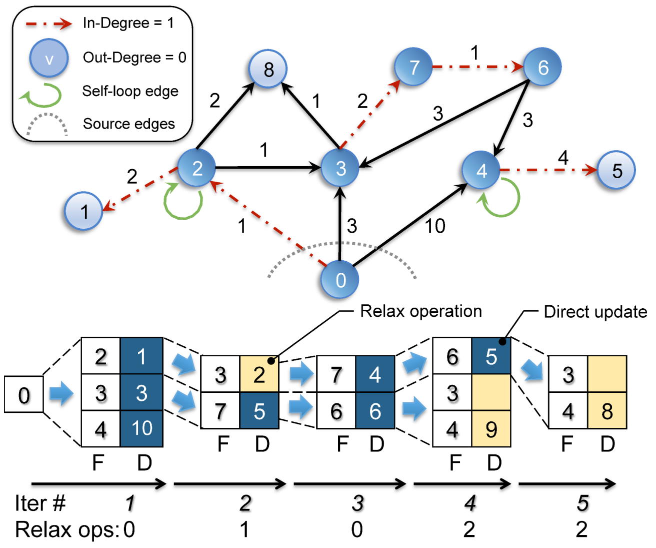


FIG 2: Example of Edge-Classification optimization

Algorithm 4 summarizes the main important steps of the *edge classification*, while Figure 2 shows such an optimization applied to the example of Figure 1, by underlining how it reduces the number of relax operations of about five times with respect to the standard approach. The example in Figure 2 converges in a total of 5 relax operations over five iterations.

The edge classification optimization has been implemented, in *H-BF*, through a *marking* phase, in which

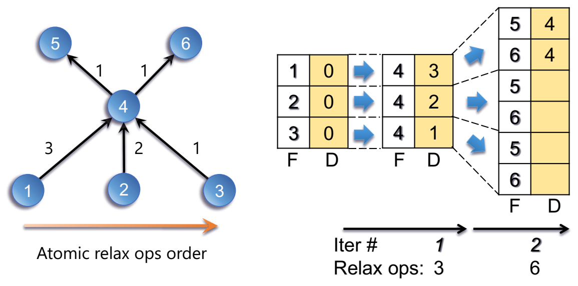


FIG 3: The duplicate problem in the frontier propagation.

each edge is classified by two bits. The marking bits are added to the bits encoding the edge. In particular, in the adopted adjacency list data structure, each edge is encoded with the *id* of the target vertex. In common GPU architectures, where the global DRAM memory is on the order of 4GBytes, such an *id* may require at most 30 bits, since, considering 4Bytes-sized *ids*, 2^{30} is the maximum number of vertices that we can handle[1]. The two most significant bits in a 32 integer id of a vertex, which are, thus, always available, are exploited for such a classification. Reading those bits to identify the edge class does not involve overhead since it is included in the memory access for reading the vertex *id*.

4.2 Duplicate removal with Kepler 64-bit atomic instructions

In the execution of a parallel Bellman-Ford implementation, *duplicate* vertices are generated when, during an algorithm iteration, more threads concurrently access the same vertex for the relax operation. This causes a vertex to be redundantly considered for relax and enqueued more times in the next frontier. Figure 3 shows an example. Initially, the frontier queue consists of vertices 1, 2, and 3. In the first iteration, the algorithm dequeues the three vertices and performs, in parallel, the relax operation over edges $(1, 4)$, $(2, 4)$, and $(3, 4)$. The memory accesses for updating the tentative distance of 4 are serialized through atomic operations to handle race conditions, and the next frontier is generated by en-queuing duplicates of vertex 4 (see iteration # 1 in the example). In turn, the duplicates are redundantly evaluated for the successive iteration and relax operations. In the example, the duplicate problem of the parallel implementation, by considering the atomic operation order shown in the figure, causes 9 relax operations instead of the three of any serial implementation.

In literature, a technique to detect and remove duplicates has been proposed by Davidson et al [32]. Such a technique allows eliminating duplicates by interleaving the main computation kernel with two additional kernels. The first aims at marking every frontier vertex through a lookup table in global memory, while the second aims at accessing the look up table to check whether the vertex index exists before proceeding with the relax phase. The *non-duplicate* vertices are compacted before carrying on with a new algorithm iteration. This strategy involves four memory accesses (two of them not coalesced) for each vertex in the frontier, and it introduces overhead for the compacting routine and for synchronizing with the host.

1. Actually, the available global memory for storing the vertex *ids* is much less since the implementation requires additional data structures for storing frontiers, edges, weights, vertex offsets, and vertex weights.

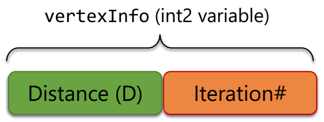


FIG 4: Use of 64-bit atomic instructions in the implementation of duplicate removal.

H-BF implements a different technique to completely avoid duplicate vertices during the graph visit by adding information (extra to the distance value) to each vertex. The distance (D) of each vertex is coupled with the number of the current algorithm iteration. The coupled information (`vertexInfo`) is stored into a 64-bit `int2` CUDA datatype, where D resides in the 32 most significant bits while the iteration number in the 32 least significant bits (see Figure 4). Access to such a variable is implemented through single-transaction 64-bit atomic operations, which are available for the GPU architectures with compute capability 3.5 onwards (i.e., Kepler, Maxwell GPUs).

Algoruthm 5 represents the high-level implementation of the technique, which highlights how each vertex enqueue is controlled by a condition on the current algorithm iteration. If the iteration number stored in the variable is equal to the actual iteration, the vertex is already in the queue, otherwise it is going to be visited, and thus inserted for the first time in the frontier. Algoruthm 6 represents the low-level implementation of such a control, by showing how the atomic primitives provided by Kepler have been applied.

Algorithm 5 ATOMIC64 RELAX PSEUDOCODE

```
RELAX_ATOM(u, v, w)

if $d(u) + w < d(v)$ then
    $d(v) = d(u) + w$
    if IterationNumber[$v$] $\neq$ currentIterationNumber then
        ENQUEUE($v$)
```

Algorithm 6 ATOMIC64 RELAX IMPLEMENTATION

```
RELAX_ATOM(u, v, w)

$u$_info = MERGE( $d(u)$, currentIteration )
old_info = ATOMICMIN( &vertexInfo[v], $u$_info )
if ( old_info.iteration $\neq$ currentIteration )
    ENQUEUE($v$)
```

5 Architecture-oriented Optimizations

Besides optimizations that target work efficiency (i.e., edge classification and duplicate removal), this article presents different optimizations that aim at improving the memory accesses bandwidth and the workload balancing.

5.1 Memory coalescing, cache modifiers, and texture memory

Coalesced memory access or *memory coalescing* refers to combining multiple memory accesses into a single transaction. When threads of the same warp concurrently access to aligned addresses in global memory, they are coalesced by the device hardware into a single transaction. In the NVIDIA Fermi, Kepler, and Maxwell architectures, the maximum coalescence in memory accesses is achieved when threads of the same warp access 128 Bytes in an contiguous region. The coalescing control in GPUs is hardware-implemented and relies on the use of cache memory. The memory coalescing is the key to reduce the overhead involved by the DRAM memory latency, which, in GPU architectures, is amplified by the thousands of threads accessing such a "slow" memory.

Cache modifiers [34] are a feature provided by Kepler GPU architectures that allows the L1 cache to be enabled or disabled at run time. This allows reducing the miss rate of cache accesses by skipping the cache use for those data not frequently used or too sparse in memory.

In *H-BF*, memory coalescing has been implemented and combined with cache modifiers as follows. Considering each algorithm step (see Section 4):

1) The first step (DEQUEUE(F)) aims at reading the frontier vertices, which are stored in global memory in consecutive locations thanks to the use of adjacency list data structures. *H-BF* forces the *cache streaming policy* to take advantage of L1/L2 caches to perform coalesced memory accessed, thus avoiding cache pollution[2]. The same cache policy is used to load edge offsets of each vertex, which are scattered and occur only one time. In this way, the caches are mainly reserved to the other steps.
2) The threads read information of edges ($v \in adj[u]$), which, similarly to the vertices, are stored in global memory in consecutive locations. Nevertheless, since the edges are much more with respect to the vertices, overloading the L1 cache may decrease the performance. *H-BF* disables the L1 cache for such data while takes advantage of the L2 cache and read-only texture memory (through the `__ldg()` Kepler operators) for caching these accesses.
3) *H-BF* implements the relax phase through atomic instructions, which do not allow exploiting memory coalescing or caching. However, in several cases, the relax operations are replaced by direct updating of the vertex distance $d(v)$ (see class 3 in Section 4.1). Memory accesses for such an operation are inevitably scattered and each distance reading occurs only once. *H-BF* directly accesses to the global memory by skipping (and avoiding cache pollution) through low-level PTX instructions [34].
4) The ENQUEUE(F, v) operation performed by each thread consists of updating the frontier data structure in global memory with each vertex v, which information is stored in the thread register. *H-BF* handles such a massively parallel memory writing by stepping into the SM shared memory to organize the data before moving into the global memory. The data organization aims at ordering the data values to enable memory coalescing. Figure 5 shows the main idea. The shared memory is partitioned into slots, one per warp. Each thread writes the vertices composing the own partial frontier into the shared memory. The threads write in parallel and start from the shared memory address (offset) computed through a *prefix-sum* procedure [35]. Then, all threads in a warp collaborate to read from the warp slot in shared memory and to perform a coalesced writing in the global memory. In Kepler architectures, the total memory dedicated to registers in each SM exceeds the size of the shared memory. This implies that a warp slot may be used more times for different transactions. Considering, for example, a 48KBytes shared memory and 2048 threads per SM, each slot is 768 Bytes sized (maximum 192 vertices per slot) and allows maximum 6 coalesced transactions to be performed. Then, the slot is released for a new set of data. In general, the thread registers are enough to store the whole frontier. In those particular cases the frontier size exceeds the available registers, the frontier updating in global memory is split in many iterations (register filling, writing in shared memory of the partial frontier, coalesced transaction in global memory, register filling, and so on).

2. The L1 cache could be disabled or partially enabled by the compiler to be used for other memory accesses like, for example, the edge reading.

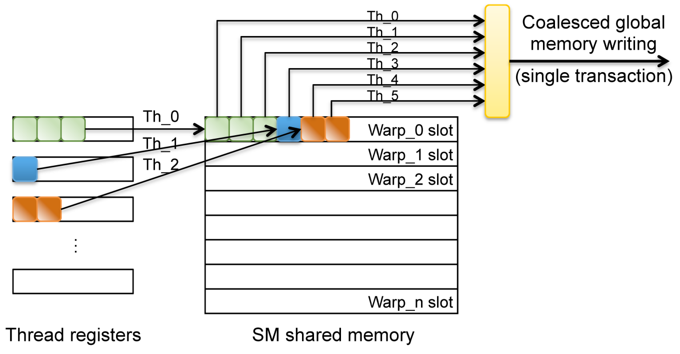


Fig 5: Example of memory coalescing for the enqueue phase

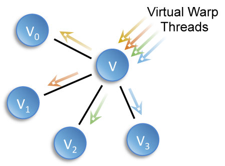


FIG 6: The Virtual Warp concept

5.2 Dynamic virtual warps

Virtual warp programming has been introduced in [36] to address the problem of workload imbalance and thread divergence in GPU graph algorithms. Such a thread scheduling strategy consists of organizing the GPU threads into groups (i.e., virtual warps) smaller than a warp and whose size is statically tuned. The idea is to assign smaller tasks to few threads (i.e., less than a warp size) to reduce as much as possible the thread divergence. This helps increasing the speedup even when the parallelism degree is low. For example, the graph algorithms exploit the virtual warp strategy to allocate one virtual warp per vertex with the aim of partitioning and equally assigning the work over the edges outgoing the vertex to each thread (Figure 6 shows an example) and finds the best application in low-degree graphs.

H-BF exploits the virtual warp technique to increase the thread coalescence during the accesses to the adjacent lists and to reduce their divergence in the frontier propagation steps. In this context, the main limitation of such a technique 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 degrees over the propagation steps (e.g., power-law graphs), the size choice may be appropriated for some vertices only. Thus, differently from [36], *H-BF* implements a *dynamic* virtual warp, whereby the warp size is calibrated at each frontier propagation step i, as follows:

$$WarpSize_i = nearest\_pow2\left(\frac{\#Res\text{Threads}}{|F_i|}\right) \in [4, 32]$$

where *#ResThreads* is the maximum number of resident threads in the GPU device and $nearest\_pow2$ is the lower nearest power of two that rounds the division. $|F_i|$ is the size of the actual frontier.

The virtual warp size may range between 1 and the maximum size of a warp (i.e., 32 for NVIDIA GPUs). Nevertheless, we heuristically found that sizes smaller than 4 threads per warp lead to a decrease of performance due to the excessive non-coalescence (close to a mere serialization) of threads. In addition, the technique proposed in [36] suffers from two problems. First, it overloads the the warp scheduler when the virtual warp size is small and the number of virtual warps is large. Then, it provides workload balancing at warp-level while while, considering that the workload assigned to each virtual warp may be different, it does not provide workload balancing at block level. Indeed, a heavier virtual warp

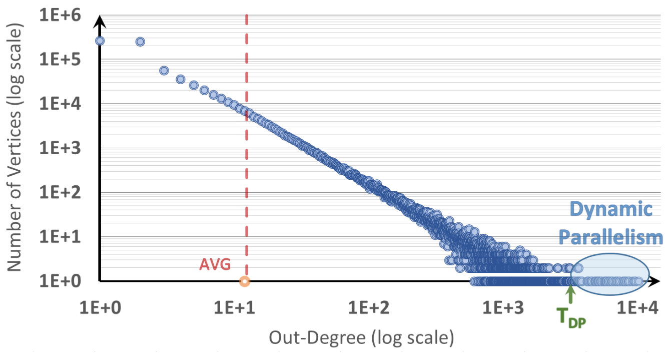


FIG 7: Example of dynamic parallelism applied to a subset of frontier vertices of a power-law graph (flickr)

may lead to the situation in which lighter warps of the same block terminate (and thus some SM cores become ready for new warps) but any new block allocation is prevented until the end of all warps.

H-BF overcomes such a problem by assigning more than one vertex per *virtual warp*. The warp scheduler overhead is minimized since there are less thread blocks in the kernel grid and the thread local queues are filled with more items that, in average, are more uniformly distributed. This provides better load balancing and coalesced global memory accesses. We heuristically fixed such a workload to 32 vertices per warp. We found that such a value leads to an increase of performance for all the analysed graphs. Higher values lead to a slight performance improvement only in graph with very high average degree.

5.3 Dynamic parallelism

The dynamic virtual warp strategy provides a fair workload balancing when applied to irregular graphs. Nevertheless, to further improve the speedup in case of very irregular graphs (i.e., scale free networks or graphs with power-law distribution), *H-BF* exploits the dynamic parallelism feature of the Kepler architectures. Dynamic parallelism allows implementing recursion in the kernels and, thus, dynamically creating threads and thread blocks at run time without requiring kernel returns. In the *H-BF* context, the idea is to invoke a multi-block kernel properly configured to manage the workload imbalance due to the difference of the vertex degrees. Nevertheless, the (even low) overhead caused by the dynamic kernel stack may elude this feature advantages when replicated for all frontier vertices unconditionally.

H-BF applies dynamic parallelism to a limited number of frontier vertices at each frontier propagation step. Given the degree distribution of the visited graph, *H-BF* applies dynamic parallelism to the sub-set of vertices that have degree far from the average (AVG) and that exceeds a threshold, T_{DP} (Figure 7 shows an example).

H-BF combines dynamic parallelism with dynamic virtual warps. The threshold T_{DP} is a knob to be set in *H-BF*, which switches from the use of the former technique to the use of the latter. As explained in the experimental results, we heuristically fixed $T_{DP} = 4096$ vertices for all the analysed graphs.

6 Experimental Results

6.1 Experimental setup

H-BF has been run on two sets of graphs. The first set is from the 9th and 10th DIMACS implementation challenges [37], [38] and from the University of Florida Sparse Matrix Collection [39]. It consists of graphs from different contexts such as, road networks, three-dimensional meshes, circuit simulations, social and synthetic graphs. The second set is from SNAP [40], 10th DIMACS, and GTGraph generator [38]. It consists of graphs from contexts like 2D dynamic simulations, communication networks, road networks, autonomous systems, and synthetic based on the Erdős-Rényi model. The graphs of the second set include some edges with negative weights though no negative cycles.

Table 1 shows the graph characteristics in terms of directed/undirected, vertices, edges, average degree, degree standard deviation, degree mode, graph diameter, and degree distribution of the edges (abscissa) over the vertices (ordinate). The degree distribution, which is shown in log scale, expresses the potential unbalancing of a parallel algorithm to visit the graph. For example, graphs like *msdoor* or *random_0.1Mv.20Me* have the best balancing as they include many vertices with the same (high) degree. In contrast, *rmat.3Mv.20Me* or *wiki-talk* are strongly unbalanced as they include many vertices with low degree, few vertices with high degree and, in general, the degree is not uniform over the vertices.

H-BF has been run on a NVIDIA (Kepler) GEFORCE GTX 780 device, which has 12 SMs, 192 Cores per SM, 3 GB of DRAM, and 5 GHz PCI Express 2.0 x16, with CUDA Toolkit 6.0, AMD Phenom II X6 1055T (3GHz) host processor, and Debian 7 operating system.

6.2 Execution time analysis and comparison

Table 2 reports the results in terms of execution time and millions of traversed edges per second (MTEPS) and the comparison of *H-BF* with the most representative SSSP implementations (both sequential and parallel for GPUs) at the state of the art. They include the Boost library sequential Dijkstra [41], which is based on priority queues and relaxed heap, and a queue-based sequential Bellman-Ford. As parallel implementations for GPUs, we selected the Lonestar GPU graph suite [7], which is a parallel implementation of Bellman-Ford, and Workfront Sweep and Near-Far Pile, which are the most efficient parallel implementations of Davidson et al. [9] (see Section 3). The results are presented as the average time and the average MTEPS obtained by running the tool from 100 sources randomly chosen, where, for each source, the connected component has at least 10^5 vertices.

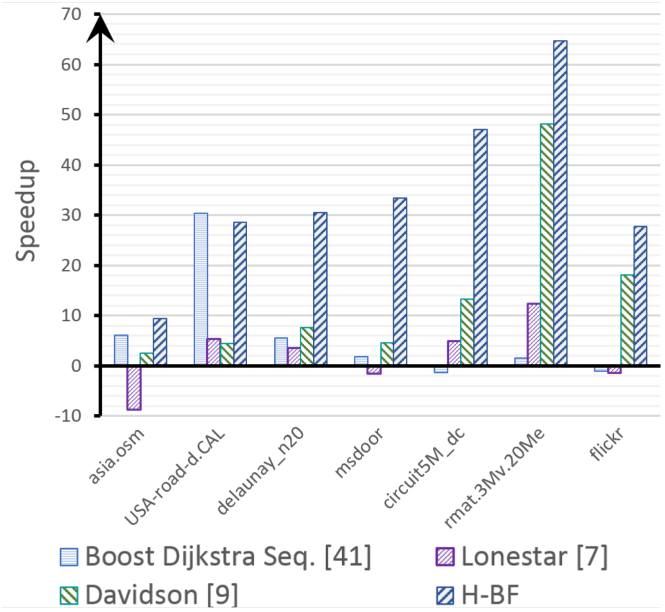


Fig 8: Comparison of speedups

Figure 8 summarizes the speedup of the different implementations with respect to the sequential queue-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 parallelism 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.

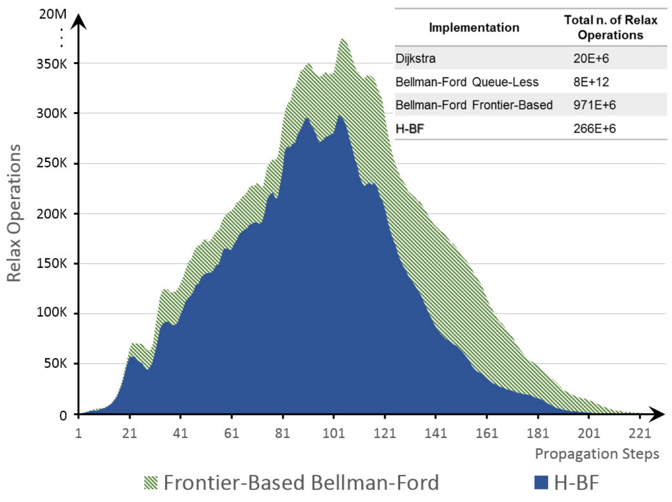
We observed the best *H-BF* performance (time and MTEPS) on the graphs in the right-most side of Figure 8 that allow high parallelism due to small diameter and high average degree. *H-BF* provides high speedup also in *rmat.3Mv.20Me* and *flickr*, which are graphs largely unbalanced (see standard deviation and power-law degree distribution in Table 1). This underlines the effectiveness of the proposed methods to deal with such an unbalancing problem in traversing graphs. We also verified that 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 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.

The second set contains graphs with negative weights (and no negative cycles) and, thus, the Dijkstra-based sequential implementation as well as the other parallel solutions at the state of the art could not be tested. For these graphs, we compared *H-BF* with respect to the Bellman-Ford sequential implementation and we evaluated the effects of each proposed optimization (see Sections 4 and 5) on the overall speedup. Table 3 reports the results. Our basic frontier-based solution provides a speedup that ranges from 12.5x to 20x with respect to the sequential counterpart. The proposed optimizations improve such a speedup from a minimum of twice (from 58.1x to 110.5x by enabling the *duplicate removal*) to almost four times (from 15.4x to 52.6x by enabling the *edge classification*).

We evaluated the impact of the *warp workload* optimization (section 5.2) to deal with the lack of parallelism in the *hugetrace_00000* graph, since it represents a 2D dynamic simulation with a low and perfect uniform degree and it is representative to be hardly visited in parallel. The *warp workload* optimization improves the load balancing and the coalesced global memory accesses by filling the local queues with more vertices.

The second graph, *wiki-talk*, is a community network with very low average degree and power-law distribution. The *edge classification* optimization in this graph allows improving the performance by more than three times. The *edge classification* optimization is particularly effective in graphs with power-law distribution since they present a high number of low-degree vertices that, in many cases, have in-degree equal to 1 and out-degree equal to 0. This allows avoiding expensive atomic operations and vertex reinsertions in the frontier. For this graph, we reported both the time spent for the main computation and the additional time to perform the two complementary kernels (in round brackets). With a high maximum degree and the highest standard deviation, the *as-Skitter* graph has the most workload unbalancing. In this case, we underlined the effects of the *Dynamic Virtual Warp* and *Dynamic Parallelism* optimizations. The combination of these techniques allows reaching high throughput with irregular workload, by dealing with both low and high degree vertices. Finally, we considered a random-generated graph with a very low diameter and a high average degree. Traversing any graph with these characteristics leads to a high number of redundant vertices since many threads have high probability to concurrently access the same vertex for the relax operation. In this case, the *duplicate removal* optimization allows improving the performance of twice by avoiding multiple extractions of the same vertex from the frontier.

Finally, Figure 9 shows the global effect of the presented optimizations on the *H-BF* work efficiency. The figure reports such an analysis by comparing a Bellman-Ford queue-less (i.e., without frontier) sequential implementation, our basic Bellman-Ford queue-based sequential implementation, the Boost Dijkstra queue-based sequential implementation [41], and *H-BF* in terms of total number of relax operations performed during the SSSP elaboration on the *msdoor* graph (the analysis results are similar for the other graphs). For the sake of clarity, the Boost Dijkstra result is not reported in the figure since it consists of a very long horizontal line (one relax operation for each 20M of edges). As expected, the Dijkstra's and Bellman-Ford's queue-less are the most and the least work efficient implementations, respectively. The use of the frontier concept on the Bellman-Ford implementation sensibly reduces the relax operations. *H-BF* further reduces such a work to a final difference of one order of magnitude with respect of Dijkstra's rather than six orders of magnitude of the original Bellman-Ford's queue-less implementation.



Implementation	Total n. of Relax Operations
Dijkstra	20E+6
Bellman-Ford Queue-Less	8E+12
Bellman-Ford Frontier-Based	971E+6
H-BF	266E+6

FIG 9: Impact of the proposed optimizations on the implementation work efficiency

7 Concluding Remarks

This article presented *H-BF*, a parallel implementation of the Bellman-Ford algorithm for Kepler GPU architectures. The article presented different optimizations oriented both to the algorithm and to the architecture, which have been implemented in *H-BF* to improve the performance and, at the same time, to optimize the work inefficiency typical of the Bellman-Ford algorithm. Experimental results have been conducted on graphs of different sizes and characteristics to compare the proposed approach with the most representative sequential and parallel implementations at the state of the art for solving the SSSP problem. Finally, the article presented an analysis of the impact of the proposed optimization strategies over different graph characteristics to understand how they impact on the *H-BF* work efficiency. An OpenCL implementation of the proposed solution is currently under study. The challenge is to observe how much the performance of the OpenCL and CUDA implementations differ since they provide different low-level instructions as well as the opportunity of implementing different hardware-oriented techniques.

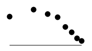
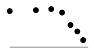
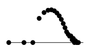
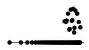
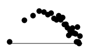
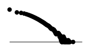
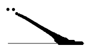
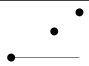
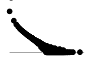
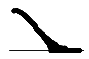
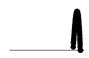
Graph Name	Directed / Undireced	Group	Vertices	Edges	Avg. Degree	Std. Deviation	Max. Degree	Diameter	Degree Distribution
asia.osm	U	Dimacs 10th [37]	12.0M	25.4M	2.1	0.5	9	38,576	
USA-road-d.CAL	D	Dimacs 9th [42]	1.9M	4.7M	2.5	0.9	7	2,575	
delaunay_n20	U	Dimacs 10th [37]	1.9M	6.3M	6.0	1.3	23	380	
msdoor	U	INPRO [39]	415K	20.6M	49.7	11.7	78	167	
circuit5M_dc	D	Freescale [39]	3.5M	19.2M	5.4	2.1	27	135	
rmat.3Mv.20Me	U	GTGraph [38]	3.0M	20.0M	6.7	10.2	521	15	
flickr	D	Gleich [39]	820K	9.8M	12.0	87.7	10,272	12	
Hugetrace_00000	U	Dimacs 10th [37]	4.6M	13.8M	3.0	0.0	3	4,119	
wiki-talk	D	SNAP [40]	2.4M	5.0M	2.1	99.9	100,022	9	
as-Skitter	U	SNAP [40]	1.7M	22.2M	13.1	136.9	35,455	31	
random_2Mv.128Me	U	GTGraph [38]	2.0M	128.0M	64.0	8	114	5	

TABLE 1: Characteristics of the graph datasets on which *H-BF* has been evaluated, including both real and synthetic datasets

Graph Name	Bellman-Ford Queue-Based Seq.		Boost Dijkstra Seq. [41]		LoneStar [7]		WorkFront Sweep / Near-Far Pile [32]		H-BF	
	Time	MTEPS	Time	MTEPS	Time	MTEPS	Time	MTEPS	Time	MTEPS
asia.osm	32.0 s	0.8	5.2 s	4.9	280 s	0.1	12.7 s	2	3.4 s	7.5
USA-road-d.CAL	20.6 s	0.2	588 ms	7.9	3.9 s	1.2	4.6 s	1	720 ms	6.4
delaunay_n20	3.2 s	2.0	581 ms	10.8	902 ms	7.0	420 ms	15	105 ms	60
msdoor	1.2 s	17.2	676 ms	30.6	1.9 s	10.8	206 ms	100	36 ms	570
circuit5M_dc	3.2 s	6.0	4.1 s	4.7	657 ms	29.2	240 ms	80	68 ms	282
rmat.3Mv.20Me	6.4 s	3.1	4.0 s	5.0	520 ms	38.5	133 ms	150	99 ms	201
flickr	887 ms	11.1	963 ms	10.2	1.2 s	8.2	49 ms	200	32 ms	307

TABLE 2: Performance comparison of *H-BF* with the most representative implementations at the state of the art.

Graph Name	Optimization	Notes	Belman-Ford Queue-Based Seq.	H-BF w/out Opt.	Speedup w/out Opt. vs. Seq.	H-BF with Opt.	Speedup with Opt. vs. Seq.
hugetrace_00000	Warp Workload	Sparse graph	82.0 s	4.1 s	20.0x	1.4 s	58.6x
wiki-talk	Edge Classification	Sparse graph	1.0 s	65 ms	15.4x	17(+2) ms	52.6x
as-Skitter	Dynamic Parallelism + Dynamic Virtual Warp	High Std. Deviation and Max. Degree	2.5 s	199 ms	12.5x	77 ms	32.5x
random_2Mv.128Me	64-bit Atomic Instr.	Small Diameter	84 s	1,445 ms	58.1x	760 ms	110.5x

TABLE 3: Impact of *H-BF* optimizations. Comparison between the speedups versus the sequential implementation obtained by enabling or disabling a specific optimization.

References

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

[2] E. W. Dijkstra, "A note on two problems in connexion with graphs," *NUMERISCHE MATHEMATIK*, vol. 1, no. 1, pp. 269–271, 1959.

[3] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

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

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

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

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

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

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

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

[11] M. Saad, "Joint optimal routing and power allocation for spectral efficiency in multihop wireless networks," *IEEE Transactions on Wireless Communications*, vol. 13, no. 5, pp. 2530– 2539, 2014.

[12] S. Klamt and A. von Kamp, "Computing paths and cycles in biological interaction graphs," *BMC Bioinformatics*, vol. 10, no. 6, pp. 1–11, 2014.

[13] S. Lu, P. Weston, S. Hillmansen, H. Gooi, and C. Roberts, "Increasing the regenerative braking energy for railway vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 15, no. 181, pp. 2506–2515, 2009.

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

[15] ——, "Cuda home page," https://developer.nvidia.com/cuda-zone.

[16] ——, "Kepler gk110," www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf.

[17] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, "Shortest paths algorithms: Theory and experimental evaluation," *Mathematical programming*, vol. 73, no. 2, pp. 129–174, 1996.

[18] F. B. Zhan and C. E. Noon, "Shortest path algorithms: an evaluation using real road networks," *Transportation Science*, vol. 32, no. 1, pp. 65–73, 1998.

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

[20] U. Meyer and P. Sanders, "Δ-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.

[21] J. R. Crobak, J. W. Berry, K. Madduri, and 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.

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

[23] K. Kelley and T. B. Schardl, "Parallel single-source shortest paths."

[24] H. N. Garbow, "Scaling algorithms for network problems," *Journal of Computer and System Sciences*, vol. 31, no. 2, pp. 148–168, 1985.

[25] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *ACM SIGPLAN Notices*, vol. 48, no. 8. ACM, 2013, pp. 135–146.

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

[27] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and 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.

[28] 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*, vol. 13, no. 3, pp. 219–237, 2005.

[29] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and 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.

[30] R. Sedgewick and K. Wayne, *Algorithms*. Pearson Education, 2011.

[31] U. Pape, "Implementation and efficiency of moore-algorithms for the shortest route problem," *Mathematical Programming*, vol. 7, no. 1, pp. 212–222, 1974.

[32] 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.

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

[34] NVIDIA, "Parallel thread execution isa version 4.1," *http://docs.nvidia.com/cuda/parallel-thread-execution/*, vol. 1, 2014.

[35] J. D. O. Mark Harris, Shubhabrata Sengupta, *GPU Gems 3: Parallel Prefix Sum (Scan) with CUDA*. Addison Wesley Professional, 2008, ch. 3.

[36] 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.

[37] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "10th dimacs implementation challenge: Graph partitioning and graph clustering, 2011."

[38] D. A. Bader and K. Madduri, "Gtgraph: A synthetic graph generator suite," *For the 9th DIMACS Implementation Challenge*, 2006.

[39] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[40] J. Leskovec and R. Sosič, "SNAP: A general purpose network analysis and graph mining library in C++," http://snap.stanford.edu/snap, Jun. 2014.

[41] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *Boost Graph Library: User Guide and Reference Manual, The*. Pearson Education, 2001.

[42] C. Demetrescu, A. Goldberg, and D. Johnson, "9th dimacs implementation challenge–shortest paths," *American Mathematical Society*, 2006.

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 70 papers on conference proceedings and journals.