Locality-Based Relaxation: An Efficient Method for GPU-Based Computation of Shortest Paths

Mohsen Safari[1] and Ali Ebnenasir[2]

[1] Department of Computer Engineering
University of Zanjan
Zanjan, Iran
mohsen_safari@znu.ac.ir

[2] Department of Computer Science
Michigan Technological University, U.S.A.
aebnenas@mtu.edu

Abstract. This paper presents a novel parallel algorithm for solving the Single-Source Shortest Path (SSSP) problem on GPUs. The proposed algorithm is based on the idea of *locality-based relaxation*, where instead of updating just the distance of a single vertex v, we update the distances of v's neighboring vertices up to k steps. The proposed algorithm also implements a communication-efficient method (in the CUDA programming model) that minimizes the number of kernel launches, the number of atomic operations and the frequency of CPU-GPU communication without any need for thread synchronization. This is a significant contribution as most existing methods often minimize one at the expense of another. Our experimental results demonstrate that our approach outperforms most existing methods on real-world road networks of up to 6.3 million vertices and 15 million arcs (on weaker GPUs).

1 Introduction

Graph processing algorithms have a significant impact on several domains of applications as graphs are used to model conceptual networks, systems and natural phenomena. One of the most important problems in graph processing is the Single-Source Shortest Path (SSSP) problem that has applications in a variety of contexts (e.g., traffic routing [27], circuit design [22], formal analysis of computing systems [23]). Due to the significance of the time/space efficiency of solving SSSP on large graphs, researchers have proposed [7] parallel/distributed algorithms. Amongst these, the algorithms that harness the computational power of Graphical Processing Units (GPUs) using NVIDIA's Compute Unified Device Architecture (CUDA) have attracted noticeable attention in research community [10]. However, efficient utilization of the computational power of GPUs is a challenging (and problem-dependent) task. This paper presents a highly efficient method that solves SSSP on GPUs for road networks with large dimensions.

A CUDA program is parameterized in terms of thread IDs and its efficiency mostly depends on all threads performing useful work on the GPU. GPUs include a multi-threaded architecture containing several Multi-Processors (MPs), where each MP has some Streaming Processors (SPs). A CUDA program has a CPU part and a GPU part. The CPU part is called the *host* and the GPU part is called the *kernel*, capturing an array of threads. The threads are grouped in blocks and each block will run in one MP. A few threads (e.g., 32) can logically be grouped as a *warp*. The sequence of execution starts by copying data from host to device (GPU), and then invoking the kernel. Each thread executes the kernel code in parallel with all other threads. The results of kernel computations can be copied back from device to host. CUDA's memory model is hierarchical, starting from the fastest: registers, in-block shared memory and global memory. The communication between GPU and CPU can be done through shared variables allocated in the global memory. CUDA also supports *atomic* operations, where some operations (e.g., addition of a value to a memory location) are performed in a non-interruptible fashion. To optimize the utilization of computational resources of GPUs, a kernel must (i) ensure that all threads perform useful work and ideally no thread remains idle (i.e., work efficiency); (ii) have fewer atomic commands; (iii) use thread synchronization rarely (preferably not at all), and (iv) have little need for communication with the CPU. The *divergence* of a computation occurs when the number of idle threads of a warp increases.

Most existing GPU-based algorithms [12, 13, 26, 25, 15, 5] for solving SSSP rely on methods that associate a group of vertices/arcs to thread blocks, and optimize a proper subset of the aforementioned factors, but not all. This is because in general it is hard to determine the workload of each kernel for optimum efficiency a priori. In the context of SSSP, each thread updates the distance of its associated vertex in a round-based fashion, called *relaxation*. For example, Harish *et al.* [12, 13] present a GPU-based implementation of Dijkstra's shortest path algorithm [9] where they design two kernels; one for relaxing the recently updated vertices, called the *frontier*, and the second one for updating the list of frontier vertices. Singh *et al.* [26] improve Harish *et al.*'s algorithm by using memory efficiently and using just one kernel. They also present a parallelization of Bellman-Ford's algorithm [3, 11], but use three atomic operations in the kernel. Kumar *et al.* [15] also present a parallelization of Bellman-Ford's algorithm in a two-kernel CUDA program. Busato *et al.* [5] exploit the new features of modern GPUs along with some algorithmic optimizations in order to enhance work efficiency. Meyer and Sanders [18] present the delta-stepping method where vertices are classified and relaxed in buckets based on their distance from the source. Davidson *et al.* [8] extend the idea of delta-stepping in a queue-based implementation of Bellman-Ford's algorithm where the queue contains the vertices whose outgoing arcs must be relaxed. There are several frameworks [29, 14, 28] for graph processing on GPUs whose main objective is to facilitate the formulation of graph problems on GPUs; nonetheless, the time efficiency of these approaches may not be competitive with hardcoded GPU programs.

In order to efficiently solve SSSP in large directed graphs, we present a GPU-based algorithm that minimizes the number of atomic operations, the number of kernel launches and CPU-GPU communication while increasing work efficiency. The proposed algorithm is based on the novel idea of *locality-based relaxation*, where we relax the distance of a vertex up to a few steps in its vicinity. Figure 1 illustrates the proposed concept of *locality-based relaxation* where the thread associated with v and w not only updates the distance of v's (respectively, w's) immediate neighbors, but propagates the impact of this relaxation on the neighboring vertices that can be reached from v (respectively, w) in k steps. Moreover, we provide a mechanism for systematic (and dynamic) scheduling of threads using flag arrays where each bit represents whether a thread should execute in each kernel launch. The proposed scheduling approach significantly decreases the frequency of communication between CPU and GPU. We experimentally show that locality-based relaxation increases time efficiency up to 30% for $k < 5$. Furthermore, our locality-based relaxation method mitigates the divergence problem by increasing the workload of each thread systematically, thereby decreasing the number of kernel launches and the probability of divergence.

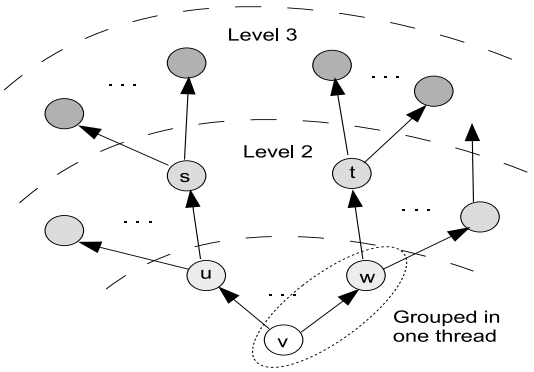

Fig. 1. Locality-Based Relaxation.

Our experimental results demonstrate that the proposed approach outperforms most existing methods (using a GeForce GT 630 with 96 cores). We conduct our experiments on the road network graphs of New York, Colorado, Pennsylvania, Northwest USA, California-Nevada and California with up to 1.9 million vertices and 5.5 million arcs, and Western USA with up to 6.3 million vertices and 15.3 million arcs. Our implementation and data sets are available at `http://gpugraphprocessing.github.io/SSSP/`. The proposed algorithm enables a computation and communication-efficient method by using (i) a single kernel launch per iteration of the host; (ii) only one atomic operation per kernel, and (iii) no thread synchronization.

Organization. Section 2 defines directed graphs, the shortest path problem and a classic GPU-based solution thereof. Section 3 introduces the idea of locality-based relaxation and presents our algorithm (implemented in CUDA) along with

its associated experimental results. Section 4 discusses some important factors that could impact GPU-based solutions of SSSP. Finally, Section 5 makes concluding remarks and discusses future extensions of this work.

2 Preliminaries

In this section, we present some basic concepts about GPUs and CUDA's programming model. Moreover, we formulate the problem statement.

2.1 Synchronization Mechanisms in CUDA

In CUDA's programming model, programmers can define thread blocks in one, two or three dimensions; however, the GPU scheduler decides how to assign thread blocks to MPs; i.e., programmers have no control over the scheduling policy. Moreover, inter-block communications must be performed via the global memory. CUDA supports atomic operations to prevent data races, where a *data race* occurs when multiple threads access some shared data simultaneously and at least one of them performs a write. CUDA also provides a mechanism for barrier synchronization amongst the threads within a block, but there is no programming primitive for inter-block synchronization.

2.2 Directed Graphs and SSSP

Let $G = (V, A, w)$ be a weighted directed graph, where V denotes the set of vertices, A represents the set of arcs and the weight function $w : A \to \mathbb{Z}$ assigns a non-negative weight to each arc. A *simple path* from some vertex $s \in V$ to another vertex $t \in V$ is a sequence of vertices $v_0, \cdots, v_k$, where $s = v_0$ and $t = v_k$, each arc $(v_i, v_{i+1}) \in A$ and no vertex is repeated. A shortest path from s to t is a simple path whose summation of weights is minimum amongst all simple paths from s to t. The Single-Source Shortest Path (SSSP) problem is stated as follows:

- INPUT: A directed graph $G = (V, A, w)$ and a source vertex $s \in V$.
- OUTPUT: The weight of the shortest path from s to any vertex $v \in V$, where $v \neq s$.

2.3 Basic Functions

Two of the most famous algorithms for solving SSSP include Dijkstra's [9] and Bellman-Ford's [3, 11] algorithms. These algorithms use a *Distance* array, denoted $d[]$. Initially, the distance of the source vertex is zero and that of other vertices is set to infinity. After termination, $d[v]$ includes the shortest distance of each vertex v from the source s. *Relaxation* is a core function in both algorithms where for each arc (u, v), if $d[v] > d[u] + w(u, v)$ then $d[v]$ is updated to $d[u] + w(u, v)$. We use the functions *notRelaxed* and *Relax* to respectively represent when an arc should be relaxed and performing the actual relaxation (see

Algorithms 1 and 2). *atomicMin* is a built-in function in CUDA that assigns the minimum of its two parameters to its first parameter in an atomic step.

Algorithm 1 notRelaxed(u,v)

```
1: if d[v] > d[u] + w(u, v) then
2:     return true;
3: else
4:     return false;
```

Algorithm 2 Relax(u,v)

```
1: atomicMin(d[v], d[u] + w(u, v));
```

2.4 Harish *et al.*'s Algorithm

In this subsection, we represent Harish *et al.*'s [12, 13] GPU-based algorithm for solving SSSP in CUDA. While their work belongs to almost 10 years ago, some researchers [26, 29] have recently used Harish *et al.*'s method as a base for comparison due to its simplicity and efficiency. Moreover, our algorithm in this paper significantly extends their work. Harish *et al.* use the Compressed Sparse Row (CSR) representation of a graph where they store vertices in an array $startV$ and the end vertices of arcs in an array $endV$ (see Figure 2). Each entry in $startV$ points to the starting index of its adjacency list in array $endV$. Harish *et al.* use the following arrays: fa as a boolean array of size $|V|$, the weight array w of size $|A|$, the distance array d of size $|V|$ and the update array up of size $|V|$. They assign a thread to each vertex. Their algorithm in [13] invokes two kernels in each iteration of the host (see Algorithm 3). The first kernel (see Algorithm 4) relaxes each vertex u whose corresponding bit $fa[u]$ is equal to *true* indicating that u needs to be relaxed. Initially, only $fa[s]$ is set to *true*, where s denotes the source vertex. The distance of any neighbor of a vertex u that is updated is kept in the array up, and $fa[u]$ is set to *false*. After the execution of the first kernel, the second kernel (see Algorithm 5) assigns the minimum of $d[v]$ and $up[v]$ to $d[v]$ for each vertex v, and sets $fa[v]$ to *true*. Harish *et al.* [12] use two kernels in order to avoid read-write inconsistencies. Their algorithm terminates if there are no more distance value changes (indicated by flag variable f remaining false).

3 Locality-Based Relaxation

In this section, we present an efficient GPU-based algorithm centered on the idea of locality-based relaxation. Subsection 3.1 discusses the idea behind our algorithm and Subsection 3.2 presents our algorithm. Subsection 3.3 explains the data set we use in our experiments. Subsection 3.4 demonstrates our experimental results and shows how our algorithm outperforms most existing methods on large graphs representing road networks. Finally, Subsection 3.5 analyzes the impact of locality-based relaxation on time efficiency.

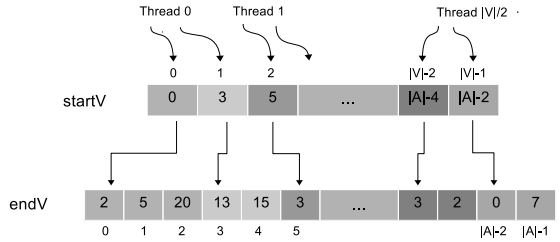

Fig. 2. Compressed Sparse Row (CSR) graph representation.

3.1 Basic Idea

Harish *et al.*'s [12] algorithm can potentially be improved in three directions. First, the for-loop in Lines 2-5 of the host Algorithm 3 requires a data exchange between the GPU and CPU in each iteration of the host through flag f. Second, their algorithm launches two kernels in each iteration of the host. Third, the kernels in Algorithms 4 and 5 contribute to propagating the wave of relaxation for just one step. We pose the hypothesis that *allocating more load to threads by (1) relaxing a few steps instead of just one, and/or (2) associating a few vertices to each thread can increase work/time efficiency.* Moreover, we claim that *a repetitive launch of kernels for some fixed number of times without any communication with the CPU can decrease the communication costs.*

Algorithm 3 Harish's algorithm: Host

1: $d[s] := 0$, $d[V - \{s\}] := \infty$,$up[s] := 0$, $up[V - \{s\}] := \infty$, $fa[s] := true$, $fa[V - \{s\}] := false$, $f := true$
2: **while** $f = true$ **do**
3: $f := false$
4: CUDA_Kernel1
5: CUDA_Kernel2

Data structure. We use the CSR data structure (see Figure 2) to store a directed graph in the global memory of GPUs, where vertices of the graph get unique IDs in $\{0, 1, \cdots, |V| - 1\}$.
Thread-Vertex affinity. In contrast to Harish *et al.* [12], we assign two vertices to each thread. (Our experiments show that assigning more than 2 vertices to each thread does not improve time efficiency significantly.) That is, thread t is responsible for the vertices whose IDs are stored in $startV[2t]$ and $startV[2t+1]$, where $0 \leq t < \lceil |V|/2 \rceil$ (see Figure 2), and $|V|$ is even. If $|V|$ is odd, then the last

thread will have only one vertex. There are two important rationales behind this idea. First, we plan to decrease the number of threads by half, but increase their load and investigate its impact on time efficiency. Second, we wish to ensure *data locality* for threads so that when a thread reads $startV[2t]$ it can read its neighboring memory cell too, hence potentially decreasing data access time.

Algorithm 4 Device: CUDA_Kernel1

```
1: For each thread assigned to vertices u
2: if fa[u] = true then
3:     fa[u] := false
4:     for each neighbor vertex v of u do
5:       Begin Atomic
6:         if up[v] > d[u] + w(u, v) then
7:             up[v] := d[u] + w(u, v)
8:       End Atomic
```

Algorithm 5 Device: CUDA_Kernel2

```
1: For each thread assigned to vertices v
2: if d[v] > up[v] then
3:     d[v] := up[v]
4:     fa[v] := true
5:     f := true
6: up[v] := d[v]
```

3.2 Algorithm

The algorithm proposed in this section includes two kernels (illustrated in Algorithms 8 and 9), but launches only one kernel per iteration. The host (Algorithm 6) initializes the distance array and an array of Boolean flags, called *FlagArray*, where $FlagArray[v] = true$ indicates that the neighbors of vertex v can be relaxed (up to k steps). Then, the host launches Kernel_1(i) for a fixed number of times, denoted N (see the for-loop), where $i \in \{0, 1\}$. We determine the value of N experimentally in an offline fashion. That is, before running our algorithm, we run existing algorithms on the graphs we use and compute the number of iterations for several runs. For example, we run Harish *et al.*'s algorithm on New York's road network for 100 random source vertices and observe that the minimum number of iterations in which this algorithm terminates is about 440. Thus, we set the value of N to $440/k$, where k is the distance up to which each thread performs locality-based relaxation. The objective is to reduce the frequency of CPU-GPU communications because no communication takes place between CPU and GPU in the for-loop in Lines 4-6 of Algorithm 6. While the the repeat-until loop in Algorithm 6 might have fewer number of iterations compared with the total number of iterations of the for-loop, the device (i.e., GPU) communicates with the host by updating the value of $Flag$ in each iteration of the repeat-until loop.

Algorithm 7 forms the core of the kernel Algorithms 8 and 9. Specifically, it generates a wave of relaxation from a vertex u that can propagate up to k steps,

where k is a predetermined value (often less than 5 in our experiments). Lines 4-10 of Algorithm 7 update the distance of each vertex v that is reachable from u in at most k steps. The relaxation wave propagates in a Depth First Search (DFS) fashion up to depth k (see Lines 8-10 of Algorithm 7). Upon visiting each vertex v via its parent w in the DFS tree, we check if the arc (w, v) is already relaxed. If so, we backtrack to w. Otherwise, we relax (w, v) and check if v is at depth k. If so, then we set the flag array cell corresponding to v in order to indicate that relaxation should be picked up from the frontier vertex v in the next kernel iteration. The impact of a wave of relaxation that starts from u is multiple waves of relaxation starting from current frontier vertices in the next iteration of the for-loop (respectively, repeat-until loop) in Algorithm 6. Thus, we conjecture that the total number of iterations of both loops in the host Algorithm 6 should not go beyond the length of the graph diameter divided by k, where the *diameter* is the longest shortest path between any pair of vertices.

Algorithm 6 Host

```
1: d[s] := 0, d[V − {s}] := ∞,
2: FlagArray[0][s] := true, FlagArray[0][V − {s}] := false, FlagArray[1][V] :=
   false, i ∈ {0, 1}, Flag := false
3: i := 0
4: for j := 1 to N do
5:     Launch Kernel_1(i mod 2)
6:     i := i + 1;
7: repeat {
8:     Flag := false // GPU and CPU communicate through Flag variable.
9:     Launch Kernel_2(i mod 2)
10:    i := i + 1
11: } until (Flag = false)
```

Algorithm 7 uses a two-dimensional flag array in order to ensure Lines 2-3 and 9 of Algorithm 7 will not be executed simultaneously on the same array cell; hence data race-freedom. Consider the case where Algorithm 7 used a single-dimensional flag array. Let u be a frontier vertex of the previous kernel launch (i.e., $FlagArray[u]$ is *true*) and t_1 be the thread associated with u. Moreover, let t_2 be another thread whose DFS search reaches u at depth k. As a result, there is a possibility that thread t_2 assigns *true* to $FlagArray[u]$ in Line 9 of Algorithm 7 exactly at the same time that thread t_1 is reading/writing $FlagArray[u]$ at Line 2 or 3; hence a data race. Since we would like to have no inter-thread synchronization (for efficiency purposes) and yet ensure data race-freedom, we propose a scheme with two flag arrays where in each kernel launch one of them plays the role of the array from which threads read (i.e., $FlagArray[i][u]$) and the other one is the array that holds the frontier vertices (i.e., $FlagArray[i \oplus 1][u]$). Thus, in each iteration of the host where Algorithm 7 is invoked through one of the kernels, $FlagArray[i][u]$ and $FlagArray[i \oplus 1][v]$ cannot point to the same memory cell because i and $i \oplus 1$ cannot be equal in modulo 2.

To increase resource utilization, each thread t, where $0 \leq t < \lceil |V|/2 \rceil$, in the kernel Algorithms 8 and 9 simultaneously performs locality-based relaxation on two vertices $u := startV[2t]$ and $u' := startV[2t+1]$. If vertex u is flagged for relaxation (Line 2 in Algorithm 7), then thread t resets its flag and starts relaxing the neighbors of u that are reachable from u by up to k steps. We invoke Kernel_1(i) repeatedly (in the for-loop in Algorithm 6) in order to propagate the wave of relaxation in the graph for N times without communicating the results with the CPU. After exiting from the for-loop in the host (Algorithm 6), we expect to have updated the distances of majority of vertices. To finalize the relaxation, the repeat-until loop in the host repeatedly invokes Kernel_2(i) until no more updates take place. Kernel_2(i) (Algorithm 9) is similar to Kernel_1(i) (Algorithm 8) except that it communicates the result of locality-based relaxation with the CPU in each iteration via the *Flag* variable.

Algorithm 7 RelaxLocalityAndSetFrontier(u, k, i)

```
1: localFlag := false
2: if FlagArray[i][u] = true then
3:     FlagArray[i][u] := false
4:     Launch an iterative DFS traversal starting at u
5:       Upon visiting any vertex v via another vertex w, do the following:
6:         if (w, v) is already relaxed  then backtrack to w.
7:         else Relax(w, v)
8:         if (v is at depth k from u) then
9:             FlagArray[i ⊕ 1][v] := true      // ⊕ denotes addition modulo 2
10:            localFlag := true
11: return localFlag;
```

Algorithm 8 Device: Kernel_1(i)

```
1: For each thread t assigned to vertices u := startV[2t] and u' := startV[2t + 1]
2:     RelaxLocalityAndSetFrontier(u, k, i)
3:     RelaxLocalityAndSetFrontier(u', k, i)
```

Algorithm 9 Device: Kernel_2(i)

```
1: For each thread t assigned to vertices u := startV[2t] and u' := startV[2t + 1]
2:     Flag := Flag∨ RelaxLocalityAndSetFrontier(u, k, i)
3:     Flag := Flag∨ RelaxLocalityAndSetFrontier(u', k, i)
```

Theorem 1. *The proposed algorithm terminates and correctly calculates the distance of each vertex from the source. (Proof omitted due to space cnstraints.)*

3.3 Data Set

In our experiments, we use real-world road network graphs. Table 1 summarizes these graphs along with the names we use to refer to them throughout the paper. These graphs represent real-world road networks taken from [1] and [2], and they are practical examples of sparse graphs with a low max outdegree, low median outdegree and low standard deviation of outdegrees.

3.4 Experimental Results

In this section, we present our experimental results in addition to comparing them with related work (see Table 2). We conduct our experiments with 100 random sources in each graph and take an average of the time cost over these 100 experiments.

Platform. We use a workstation with 16 GB RAM, Intel Core i7 3.50 GHz processor running Linux Ubuntu and a single NVIDIA GeForce GT 630 GPU with 96 cores. The graphics card has a total of 4095 MB RAM, but 2048 MB is dedicated to video. We implement our algorithm in CUDA version 7.5.

Table 1. Graphs used in our experiments (All graphs have an average outdegree of 2).

Graphs	Name	♯ of Vertices	♯ of Arcs	Maximum Outdegree	Standard Deviation of Outdegree	Median of Outdegree
New York City [1]	New York	264,346	733,846	8	1.24	3
Colorado [1]	Colorado	435,666	1,057,066	8	1.02	2
roadNet-PA [2]	Pennsylvania	1,090,903	3,083,796	20	1.31	3
Northwest USA [1]	Northwest	1,207,945	2,840,208	9	1.00	2
California and Nevada [1]	CalNev	1,890,815	4,657,742	8	1.05	3
roadNet-CA [2]	California	1,971,278	5,533,214	12	1.28	3
Western USA [1]	Western	6,262,104	15,248,146	9	1.02	3

Results. Table 2 compares our algorithm with some related work in terms of space complexity, number of kernel launches, frequency of CPU-GPU communication, the number of atomic statements and speed up over Harish *et al.*'s algorithm. Notice that our approach provides the best speed up while minimizing other factors. The most recent approaches that outperform Harish *et al.*'s algorithm belong to [16, 25, 26] with a speed up of at most 2.6 (see Table 2). Figure 3 illustrates our experimental results in comparison with Harish *et al.*'s. We have run both algorithms on the same platform and same graphs. Observe that in all graphs our algorithm outperforms Harish *et al.*'s algorithm significantly. Specifically, we get a speed up from 3.36 for CalNev to 5.77 for California. Notice that Western (see Figure 3) is the largest sparse graph in our experiments with 6.2 million vertices and more than 15 million arcs. Our algorithm solved SSSP for Western in about 4.9 seconds, whereas Harish *et al.*'s algorithm took 24.7 seconds! Moreover, for the road networks of California and Nevada, our implementation solves SSSP in almost 3.5 seconds on an NVIDIA GeForce GT 630 GPU, whereas (1) Davidson *et al.*'s [8] method takes almost 4 seconds on an NVIDAI GTX 680 GPU; (2) Boost library [24] takes 588 milliseconds; (3) LoneSatr [4] takes 3.9 seconds, and (4) H-BF [5] takes 720 milliseconds on an NVIDIA (Kepler) GeForce GTX 780. Observe that given the weak GPU available to us, our implementation performs well and outperforms some of the aforementioned approaches.

Number of kernel launches. The number of kernel launches in each iteration of the host algorithm has a direct impact on time efficiency; the lower the number

Table 2. Comparison with related work

Summarizing All Related Works					
Methods/Criteria	Space Complexity	♯ of Kernel Launches	CPU-GPU Communication (# per host iteration)	♯ of Atomic Stmts	Speed Up over Harish
Harish *et al.* [12, 13]	4V+2A	2	≥ 1	1	-
Chaibou *et al.* [6]	V^2+3V	2	≥ 1	1	-
Singh *et al.* [26]	3V+2A	1	≥ 1	1	2.5x
Singh *et al.* [25]	4V+2A	2	≥ 1	2	1.9x-2.6x
Busato *et al.* [5]	4V+2A	2	≥ 1	2	-
Ortega *et al.* [20, 21]	5V+2A	3	≥ 1	1	-
Proposed Algorithm	4V+2A	1	< 1	1	3.36x-5.77x

of kernel launches, the better. Observe that our algorithm and that of Singh *et al.* [26] outperform the rest.

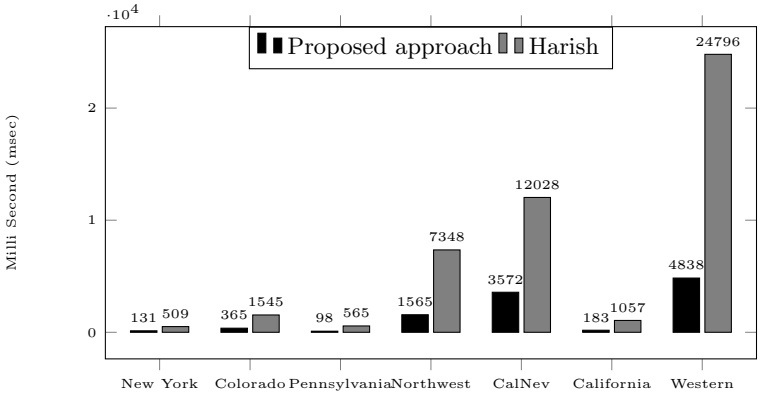

Fig. 3. Time efficiency of the proposed approach vs. Harish *et al.*'s [12, 13].

Number of atomic statements. While the use of atomic statements helps in data race-freedom, they are considered heavy-weight instructions. As such, we would like to minimize the number of atomic statements. In addition to our algorithm and Harish *et al.*'s [12, 13], Singh *et al.* [26], Chaibou *et al.* [6] and Ortega *et al.* [20, 21] present algorithms with just one atomic statement. Chaibou *et al.* [6] evaluate the cost of memory copy between CPU and GPU. Ortega *et al.* [20, 21] propose an algorithm based on Dijkstra's algorithm to find SSSP. Their

method extends Martin *et al.*'s [17] and Crauser *et al.*'s [7]. To increase the degree of parallelism in Dijkstra's algorithm, Martin *et al.* [17] consider all vertices from frontier with minimum distances to do the relaxation simultaneously. Crauser *et al.* [7] improve this method by proposing a threshold. Their idea is based on maximizing the number of relaxations in each iteration while preserving the correctness of Dijkstra's algorithm. Ortega *et al.* [20, 21] implement these two ideas on GPUs. Nasre *et al.* [19] claim that atomic-free algorithms perform more efficiently than the algorithms that use atomic statements. Their results show a small time improvement for SSSP.

Speed up over Harish's. We include a column in Table 2 to illustrate how much speed up our algorithms provide compared with Harish *et al.*'s work. Notice that our algorithm improves time efficiency in comparison to other methods.

3.5 Locality-Based Relaxation

This section analyzes the impact of locality-based relaxation on time efficiency. To validate the proposed hypotheses in Section 3.1, we have conducted a few comparative experiments on graphs NY, CN and WUS in Table 3. We consider two criteria: one is the value of k that determines how far relaxations would go when updating $d[v]$ for some vertex v, and the other one is the impact of thread-vertex affinity. As such, we replace the original weights in the road network graphs of New York City, California-Nevada and Western USA with random values in the interval [1..10]; the actual weights are irrelevant for this experiment. This change enables faster runs of our algorithm on the aforementioned graphs.

Figure 4 illustrates the results of our experiments. Observe that as the value of k is increased from $k = 1$ the time costs decrease until we reach $k = 4$. From $k = 4$ to $k = 5$ we do not observe a significant decrease in time costs since the threads get saturated in terms of their workload. Moreover, determining the best value of k seems to be dependent on a few factors such as (i) the graph being processed; (ii) the algorithm, and (iii) the platform. In the context of our setting, $k = 4$ seems to be the best value. Moreover, we notice that assigning two vertices to one thread increases the workload of each thread and decreases the execution time (see Figure 4), but assigning more than 2 vertices does not result in a significant performance improvement.

Table 3. Revised graphs used in our experiments.

Graphs	Acronym	Description
New York City [1]	NY	Replaced the original arc weights with some random value between 1 and 10 (inclusive).
California and Nevada [1]	CN	Same as above.
Western USA [1]	WUS	Same as above.

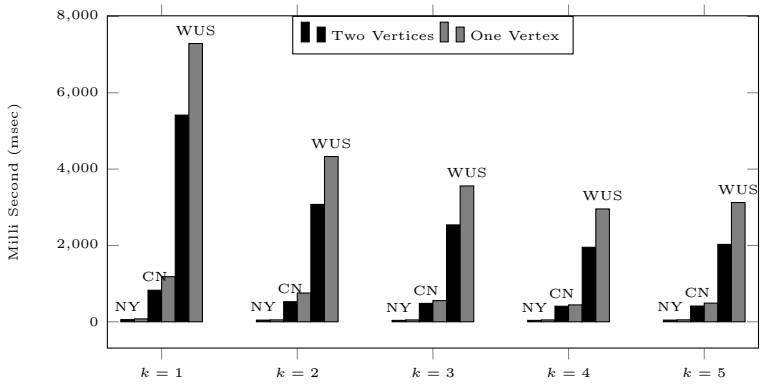

Fig. 4. Impact of locality-based relaxation and association of threads to vertices on execution time.

4 Discussion

In this section, we discuss some ideas that can potentially result in a more efficient GPU implementation that solves SSSP and its variants. In our experience, there are a few factors that have a direct impact on the time/space/work efficiency of a GPU implementation for SSSP. First, minimizing **CPU-GPU communication** can have a significant impact on time efficiency of CUDA programs. For this reason, we design our algorithm in a way that for N iterations of the host there is no communication between the GPU and the CPU. We experimentally observe that this design decision made a significant difference in decreasing the overall execution time. Second, the **data structure** that keeps the frontier vertices, has a noticeable impact on both space and time efficiency. Most existing methods use a queue. The operations performed on queues include enqueue, dequeue and extractMin, which may become costly depending on the graph being processed. A flag array keeps track of the frontier by a bit pattern, where each vertex v has a corresponding bit indicating whether v's distance got updated in the last round. The use of queues may cause another problem where two different threads update the same vertex v at different times and enqueue v, called *vertex duplication* (addressed by Davidson *et al.* [8]). Moreover, using flag arrays allows programmers to devise a well-thought schedule for threads towards avoiding data races; hence decreasing the number of required atomic statements. Third, the **number of kernel launches** and the way we launch them is influential. We observe that having fewer number of kernel launches in each iteration of the host is useful, but on-demand kernel launches do not help; rather it is better to have a fixed number of threads that are loaded with useful work in each launch. Thus, it is important to design algorithms in which all threads perform useful work in each launch (see Section 3.5). We also note that, in the context of our work, replacing atomic operations with busy waiting (as suggested by Nasre *et al.* [19]) does not improve the efficiency of our implementation. Finally, the

scalability of the proposed algorithm is a challenge in that the GPU memory is limited but there is a constant need for solving SSSP in larger graphs.

5 Conclusions and Future Work

This paper presented an efficient GPU-based algorithm for solving the SSSP problem based on a novel idea of *locality-based relaxation*, where we allow a thread to relax all vertices up to k steps away from the current vertex. We also devised a mechanism for systematic scheduling of threads using flag arrays where each bit represents whether a thread should execute in a kernel launch. The proposed scheduling approach enables a communication-efficient method (in the CUDA programming model) that minimizes the number of kernel launches, the number of atomic operations and the frequency of CPU-GPU communication without any need for thread synchronization. The proposed algorithm solves the SSSP problem on large graphs (representing road networks) with up to 6.2 million vertices and 15 million arcs in a few seconds, outperforming existing methods. As for the extensions of this work, we would like to leverage our proposed technique in solving search problems (e.g., DFS, BFS) on large graphs. We also plan to investigate the application of our GPU-based implementation in devising efficient model checking algorithms. Finally, we will study a multi-GPU implementation of our algorithm towards processing even larger graphs.

References

1. 9th DIMACS implementation challenge-shortest paths, http://www.dis.uniroma1.it/challenge9/download.shtml.
2. Stanford network analysis project, http://snap.stanford.edu/.
3. R. Bellman. On a routing problem. *Quarterly of applied mathematics*, pages 87–90, 1958.
4. M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on GPUs. In *Workload Characterization (IISWC), IEEE International Symposium on*, pages 141–151. IEEE, 2012.
5. F. Busato and N. Bombieri. An efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures. *IEEE Transactions on Parallel and Distributed Systems*, 27(8):2222–2233, 2016.
6. A. Chaibou and O. Sie. Improving global performance on GPU for algorithms with main loop containing a reduction operation: Case of Dijkstra's algorithm. *Journal of Computer and Communications*, 3(08):41, 2015.
7. A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. In *International Symposium on Mathematical Foundations of Computer Science*, pages 722–731. Springer, 1998.
8. A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 349–359, 2014.
9. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

10. R. Farber. *CUDA application design and development*. Elsevier, 2011.
11. L. R. Ford Jr. Network flow theory. Technical report, DTIC Document, 1956.
12. P. Harish and P. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *High performance computing–HiPC 2007*, pages 197–208. 2007.
13. P. Harish, V. Vineet, and P. Narayanan. Large graph algorithms for massively multithreaded architectures. *International Institute of Information Technology Hyderabad, Tech. Rep. IIIT/TR/2009/74*, 2009.
14. F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. CuSha: Vertex-centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, pages 239–252, 2014.
15. S. Kumar, A. Misra, and R. S. Tomar. A modified parallel approach to single source shortest path problem for massively dense graphs using CUDA. In *Computer and Communication Technology (ICCCT), 2nd International Conference on*, pages 635–639. IEEE, 2011.
16. D. Li and M. Becchi. Deploying graph algorithms on GPUs: An adaptive solution. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 1013–1024. IEEE, 2013.
17. P. J. Martín, R. Torres, and A. Gavilanes. CUDA solutions for the SSSP problem. In *International Conference on Computational Science*, pages 904–913, 2009.
18. U. Meyer and P. Sanders. Delta-Stepping: A Parallel Single Source Shortest Path Algorithm. In *Proceedings of the 6th Annual European Symposium on Algorithms*, ESA '98, pages 393–404, 1998.
19. R. Nasre, M. Burtscher, and K. Pingali. Atomic-free irregular computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 96–107. ACM, 2013.
20. H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos. Comprehensive evaluation of a new GPU-based approach to the shortest path problem. *International Journal of Parallel Programming*, 43(5):918–938, 2015.
21. H. Ortega-Arranz, Y. Torres, D. Llanos, and A. Gonzalez-Escribano. A new GPU-based approach to the shortest path problem. In *High performance computing and simulation (HPCS)*, pages 505–511. IEEE, 2013.
22. N. A. Sherwani. *Algorithms for VLSI physical design automation*. Springer Science & Business Media, 2012.
23. S. Shirinivas, S. Vetrivel, and N. Elango. Applications of graph theory in computer science an overview. *International Journal of Engineering Science and Technology*, 2(9):4610–4621, 2010.
24. J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual, Portable Documents*. Pearson Education, 2001.
25. D. P. Singh and N. Khare. Modified Dijkstra's algorithm for dense graphs on GPU using CUDA. *Indian Journal of Science and Technology*, 9(33), 2016.
26. D. P. Singh, N. Khare, and A. Rasool. Efficient parallel implementation of single source shortest path algorithm on GPU using CUDA. *International Journal of Applied Engineering Research*, 11(4):2560–2567, 2016.
27. C. Sommer. Shortest-path queries in static networks. *ACM Computing Surveys (CSUR)*, 46(4):45, 2014.
28. Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A High-performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 11:1–11:12, 2016.
29. J. Zhong and B. He. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2014.