DEGREE PROJECT IN COMPUTER ENGINEERING,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2017

Experimental Evaluation of GPU Solutions to the Single Source Shortest Path Problem

DAN HEMGREN, ERNST WIDERBERG

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

Experimental Evaluation of GPU Solutions to the Single Source Shortest Path Problem

DAN HEMGREN, ERNST WIDERBERG

Degree Project in Computer Science, DD143X
Date: June 5, 2017
Supervisor: Mårten Björkman
Examiner: Örjan Ekeberg
Swedish title: Experimentell utvärdering av GPU-lösningar för SSSP-problemet
School of Computer Science and Communication

Abstract

Originally designed for computer graphics, the modern graphics processing unit (GPU) has now become a potential powerhouse for parallel calculations through the use of platforms such as CUDA and OpenCL. This paper presents an experimental evaluation of GPU- and CPU-parallel as well as CPU-sequential implementations for solving the single source shortest path problem. The algorithms tested are versions of Dijkstra's algorithm, the Bellman-Ford algorithm, and Delta-Stepping, from the Boost Graph Library, Parallel Boost Graph Library, Gunrock and LoneStarGPU. Testing the implementations on large graphs modeled on road networks over USA, results indicate that the sequential Boost implementation of Dijkstra's algorithm gives the best performance if memory transfer time to and from GPU is taken into account. The paper also presents a discussion on possible reasons for this result.

Sammanfattning

Grafikkort, vars ursprungliga syfte är beräkningar inom datorgrafik, kan idag användas för generella beräkningar via plattformar såsom CUDA och OpenCL. Denna rapport presenterar en undersökning av GPU- och CPU-parallella samt CPU-sekventiella implementationer ämnade att lösa SSSP-problemet. De testade algoritmerna är versioner av Dijkstras algoritm, Bellman-Ford-algoritmen, samt Delta-Stepping, från Boost Graph Library, Parallel Boost Graph Library, Gunrock, och LoneStarGPU. Då implementationerna testas på stora grafer modellerade på vägnätverk över USA indikerar resultaten att den sekventiella implementationen av Dijkstras algoritm når snabbast resultat om överföringstid till och från GPU beaktas. Rapporten presenterar även en diskussion kring möjliga anledningar för de funna resultaten.

Contents

1 Introduction **1**
1.1 Problem statement . . . 2

2 Background **3**
2.1 CUDA . . . 3
2.2 MPI . . . 4
2.3 The SSSP Problem . . . 4
2.4 Foundational Algorithms . . . 4
2.4.1 Dijkstra's Algorithm . . . 5
2.4.2 Bellman-Ford . . . 6
2.4.3 Delta-Stepping . . . 8
2.5 Implementations . . . 10
2.5.1 Boost . . . 10
2.5.2 Gunrock . . . 10
2.5.3 LoneStarGPU . . . 11

3 Method **12**
3.1 Tested Algorithms . . . 12
3.2 Test Data . . . 13
3.3 Procedure . . . 13

4 Results **14**

5 Discussion **22**

6 Conclusion **24**

Bibliography **25**

Chapter 1

Introduction

General purpose computing on GPUs is an important field of study due to the high degree of parallel execution performance enabled by the GPU architecture. Originally intended only for graphics processing, GPUs with their hundreds of cores are especially suited for tasks that employ data level parallelism – performing essentially the same operation on many data items simultaneously. This high capacity for parallelization has led to tools and techniques being developed that enable the use of GPUs for general purpose computing tasks normally handled by the CPU. Many algorithms have, given the right implementation details, been shown to run faster on many-core GPUs than on multi-core CPUs.

Graph problems are typically fairly sequential in nature, and thus hard to parallelize. One problem which arises in many practical situations is **the shortest path problem**, one simple instance of which is the task of finding the shortest path between two points on a map. A more general version of the same problem is known as the **single source shortest path problem** (SSSP), which asks the distance from some source vertex to each other vertex in a graph. Perhaps unintuitively, calculating the distance to *all* vertices is not computationally harder than calculating the distance to any single specific vertex.

1.1 Problem statement

In this paper, experimental results from solving SSSP for large road networks using GPU parallel algorithms on consumer hardware are compared and evaluated against CPU versions of three foundational algorithms – Dijkstra’s algorithm, Bellman-Ford, and Delta-Stepping.

The aim of these experiments is to determine whether any of two tested GPU solutions, Gunrock and LoneStarGPU, can outperform single- and multi-core CPU solutions for SSSP on road network graphs using consumer hardware.

Chapter 2

Background

This section presents the algorithms and implementations studied throughout the experiment along with brief explanations of the single source-shortest path problem and the CUDA platform used for general purpose GPU computing. As each implementation uses a derived form of either Dijkstras algorithm, the Bellman-Ford algorithm or Delta-Stepping, we consider these to be *foundational* algorithms and explain them in detail.

2.1 CUDA

CUDA is a platform for parallel computing and an application programming interface (API) created by NVIDIA for use with CUDA-enabled graphics processing units. The API exposes low-level GPU instructions to higher level programming languages enabling general purpose GPU programs written in C, C++ or Fortran using the API to be run on supported hardware [15].

The CUDA programming model separates the computing system into two entities. The CPU is considered the *host*, and is responsible for running all code not exhibiting data parallelism. The GPU is the *device*, to which the CPU should delegate all SPMD (single program, multiple data) instructions via *kernel functions*. These kernel functions are then executed in parallel over the data set using a large amount of CUDA cores. The host and device do not share memory space; any data a kernel function will operate on needs to be transfered to the device memory space beforehand and subsequently retrieved once the computations are done. [10]

Executing a CUDA kernel function spawns a two-level hierarchy of threads called a *grid* to be run on the device. A grid consists of a two-dimensional array of *blocks*, and each block is a three-dimensional array of threads. Grid dimensions are configured when calling the kernel function. Threads are executed per block on the device without an explicit execution order. As a result of this, no thread synchronization is allowed between blocks to ensure scalable performance with additional CUDA cores. [10]

2.2 MPI

Message Passing Interface (MPI) is a standard for writing programs that run in parallel in distributed memory systems [20]. It is designed for use in high performance computing, and the API supports C and Fortran with C++ support depreceated as of MPI version 2.2 [19]. Many libraries have implemented the standard such as Open MPI, MPICH and LAM/MPI. Boost.MPI was used for the multi-core CPU experiments in this paper. In short, MPI consists of routines for point-to-point communication between processes and abstractions to enable and enhance concurrency in programs [20].

2.3 The SSSP Problem

The single source shortest path problem is defined as follows: Given a directed graph with weighted edges and some source vertex s, what is the distance from s to each vertex in the graph?

2.4 Foundational Algorithms

In this section well-studied algorithms relevant to the study are presented. Many of the algorithms share certain concepts, which we will begin by providing brief explanations of.

Tentative distance: All algorithms maintain for each vertex v in the graph a so called *tentative distance* $dt(v)$, an approximation of the distance from the source vertex s to v. At any point during the algorithm's execution, v:s tentative distance is guaranteed to be equal to or greater than its true distance. It is eventually lowered until it matches the vertex's *true distance* $d(v)$, the distance of the shortest path from s to v. When a vertex's tentative distance matches its true shortest distance it is said to be **settled**. If not the vertex is **unsettled**.

Edge relaxation: An edge $E = (u, v, w)$ from u to v with weight w is relaxed by checking if $dt(v)$ can be lowered by the use of E. v can be reached by appending E to the shortest path leading to u, and the distance of this new path will be $d(u) + w$, where w is the weight of E. To relax E is thus simply to compare $d(v)$ to $d(u) + w$, and assign the lowest of the two as the new tentative distance for v.

Vertex relaxation: To relax a *vertex* u is defined as relaxing all outgoing edges from u.

2.4.1 Dijkstra's Algorithm

Dijkstra's algorithm is a foundational sequential algorithm for solving SSSP. It is theoretically optimal with regards to work amount, as each edge is relaxed only once. [12]

Initially, the tentative distance of the root vertex s is set to 0, and the distance to all other vertices is set to ∞. All vertices are then put in a set Q. The vertex s in Q with the lowest tentative distance is removed from Q and relaxed. This is repeated until Q is empty.

Thanks to the order of vertex processing (lowest tentative distance first), Dijkstra's algorithm needs only relax each edge once. In other words, each time an edge $E = (u, v, w)$ is relaxed, it is guaranteed that the tentative distance of u is equal to its true distance. To see why this is the case, look at the set of all vertices N separated by exactly one edge from the root vertex s. Let n be the vertex in N with the lowest tentative distance. At this point there might be ingoing edges to n which we have not yet considered. However, each of the potential paths to n crossing one of these not yet considered ingoing edges would have to first pass from s to a vertex in $N \setminus \{n\}$. We know that the weight of each edge going from s to a vertex in $N \setminus \{n\}$ is greater than the weight of the edge $s \to n$, so no other path to n could be shorter than $s \to n$.

Parallelization

Although Dijkstra's algorithm is by nature fairly sequential there are some steps which can be performed concurrently. Specifically, as observed by Crauser et al.[6], there is no need to relax only one vertex (the lowest tentative distance vertex) at a time. Taking a page from Martín et al. [13] we will refer to the group of all vertices which are possible to relax in parallel (in a given step of the algorithm) as the **frontier**, and a vertex in this group as a **frontier vertex**.

One method of finding such frontier vertices, based on the weight of outgoing edges, is as follows. Let $l(v) = dt(v) + mow(v)$, where $mow(v)$ is the lowest weight of any outgoing edge from v. Note that $l(v)$ is the lowest possible distance for any shortest path passing v. Now let $L = min(l(v))$ for all v in Q, and note that L is the lowest possible distance for any shortest path passing any vertex in Q. As no new path shorter than L will be discovered at any later point, all vertices in Q with $dt \leq L$ may be declared settled at once, and relaxed concurrently.

A similar method which instead considers ingoing vertices can be used to find additional frontier vertices. These two methods used in tandem form the basis of Crauser et al.'s parallel version of Dijkstra's algorithm. In our testing we have used a multi-core CPU implementation from the Parallel Boost Graph Library. [18]

2.4.2 Bellman-Ford

The Bellman-Ford algorithm is quite similar to Dijkstra's. It also maintains a tentative distance to each vertex, and starts by setting all except the root vertex's to ∞. However, Bellman-Ford is less rigid about the order of its graph exploration. Unlike in Dijkstra's algorithm, edges are relaxed in an arbitrary order, and hence the property that each edge needs to be relaxed only once is lost. After $E = (u, v, w)$ is relaxed, it might be that a shorter path to u is found, which means that E will need to be relaxed again.

To guarantee that each vertex's true distance has been found, each edge has to be relaxed $|V| - 1$ times, where $|V|$ is the number of vertices in the graph. The reason for this is that the longest possible shortest path l in any graph will be composed of $|V| - 1$ edges, and the tentative distance of at least one of the edges in l will be lowered to its true distance in every edge-relaxing iteration.

For example, consider the path in figure 2.1, where the weight of each edge is 1.

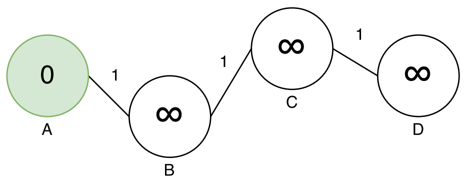

Figure 2.1: Example of a graph at the start of a Bellman-Ford execution.

In the worst possible case, the order of edge relaxation in every iteration will be d, c, b, a. This will lead the following progression of tentative distances illustrated in figure 2.2.

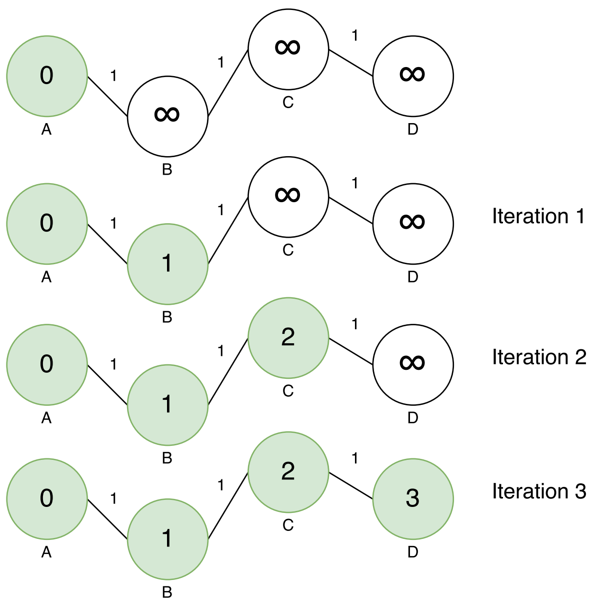

Figure 2.2: Three iterations of the Bellman-Ford algorithm.

The Bellman-Ford algorithm clearly does more work than Dijkstra's, but it has two important benefits – it is easier to parallelize and it can handle graphs with *negative weight-cycles*. A negative weight-cycle is a cycle in which the weight of all included edges is negative. In a graph with this sort of cycle the task of finding a shortest path between any two vertices is impossible, as any path in the graph can be made infinitely shorter by adding loops around the negative weight-cycle. As a result, of course, both Dijkstra's and Bellman-Ford will fail to solve SSSP given such a graph, but the difference is that Bellman-Ford is guaranteed to terminate while Dijkstra's algorithm will loop indefinitely. Dijkstra's algorithm only stops once no vertex with lowest tentative distance remians to be processed, and in a negative weight-cycle, such vertices will be continually discovered in perpetuity. Bellman-Ford on the other hand will stop once every edge has been relaxed $|V| - 1$ times. In addition, the Bellman-Ford algorithm finishes by examining all edges once more to ensure the that the input graph did not contain any negative weight-cycles.

Parallelization

Parallelization of Bellman-Ford is trivial, as each edge can be relaxed simultaneously with no race conditions. One solution is thus to let as many threads as there are edges attempt to relax one edge each, and repeat the process as long as at least one edge was relaxed in the previous iteration. This is the method employed by the LoneStarGPU library's Bellman-Ford implementation [11], which we use for testing.

2.4.3 Delta-Stepping

As mentioned in the previous section, one of the main issues with the Bellman-Ford algorithm is premature edge relaxations. There is nothing keeping an unsettled vertex from being relaxed, which of course is wasted work, as the vertex will later become settled and have to be relaxed again. Dijkstra on the other hand imposes a very strict order of edge relaxation which allows it to guarantee work optimality – each edge is relaxed only once.

Meyer and Sanders Delta-Stepping algorithm [14] can be described as a combination of Dijkstra's algorithm and Bellman-Ford, requiring careful tuning of a parameter Δ which dictates the balance between the two. It uses some of the same parallelism as Bellman-Ford, while enforcing a stricter Dijkstra-style policy on the order of edge relaxations to minimize total work. It accomplishes this by sorting vertices into different "buckets" based on tentative distance, and repeatedly performing a sort of parallel Bellman-Ford on vertices within the lowest tentative distance-bucket. All outgoing edges from vertices within the selected bucket are relaxed in parallel and the affected vertices (those whose tentative distances have changed) are assigned to new buckets.

Delta-Stepping can be summarized in three steps:

1. Select the bucket with lowest index k.

2. Perform Bellman-Ford on all vertices in bucket k until there are no more outgoing edges from vertices in bucket k to relax.
3. Select a new bucket with lowest possible index greater than k and repeat.

A key observation is that while vertices in bucket k is being processed, it is impossible for a vertex to be moved to a bucket with index lower than k. This is why it is sufficient for k to do a single single sweep from 0 up to the highest indexed bucket. However, it is possible for a vertex to be moved to the currently active bucket k while k is being processed. In this case it is simply immediately included in the following relaxation iterations for bucket k.

The algorithm takes a parameter Δ, which determines the span of tentative distances within each bucket (the "size" of each bucket). It is this parameter which sets the balance between minimizing work and maximizing parallelism, and it must be chosen with care.

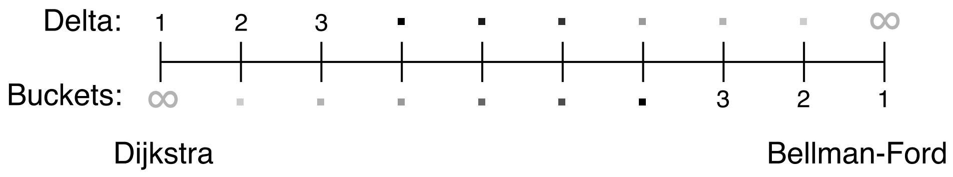

Figure 2.3: Δ:s influence on bucket count.

As illustrated in figure 2.3, a large delta value corresponds to a low number of buckets. In the extreme all vertices would be assigned to the same bucket, and only one iteration of Bellman-Ford would be carried out. With such a delta value the overall behaviour of Delta-Stepping would thus be essentially the same as that of the Bellman-Ford algorithm. With a very low delta, the number of buckets begins to equal the number of vertices and the Delta-Stepping algorithm becomes very similar to a sequential Dijkstra.

Parallelization

As mentioned above, the parallelizable part of Delta-Stepping is relaxing vertices inside a single bucket. Choosing which bucket to process (the lowest indexed one) must be done sequentially.

Davidson et al. highlights several characteristics inherent to Delta-Stepping that prevents efficient GPU-parallel implementations of the algorithm [7]. The bucket mechanism implies the need for dynamic arrays that can quickly be resized, which is unfit for the current GPU programming model. Furthermore, moving vertices between buckets likely requires atomics and communication between threads, resulting in a loss of concurrency.

However, it is quite possible to implement an efficient CPU-parallel version of Delta-Stepping. We have tested such a version from the Parallel Boost Graph Library. [18]

2.5 Implementations

This section introduces the specific algorithm implementations used in our testing.

2.5.1 Boost

Boost is a large and widely used collection of C++ libraries. Two subsets of Boost are the Boost Graph Library (BGL) and the Parallel Boost Graph Library (PBGL), containing a number of sequential and CPU-parallel implementations of SSSP algorithms. We have selected four algorithms for testing, namely

1. a sequential implementation of Dijktra's algorithm [3],
2. a sequential implementation of the Bellman-Ford algorithm [1],
3. a CPU-parallel implementation of Dijkstra's algorithm, based on the work of Crauser et al. as decribed in section 2.4.1 [16], and
4. a CPU-parallel implementation of Delta-Stepping [17].

2.5.2 Gunrock

Gunrock is a CUDA library which provides a level of abstraction on top of CUDA in the shape of a set of programming primitives for implementing graph processing algorithms. Its aim is to simplify the implementation of parallel graph algorithms for GPUs while maintaining performance comparable to that of hand-written CUDA code.

Central to the Gunrock programming model is the previously described concept of a *frontier*. Generalized to be applicable to problems outside of SSSP, the frontier is defined as a subset of either vertices or edges that are "actively participating" in a given step of the algorithm. In the case of SSSP, this is a set of vertices as described in section 2.3.1.1. Writing a Gunrock program consists of writing functions of two different types: *operators* and *functors*. Operators are functions that definine how the frontier is updated as the algorithm progresses. Functors define what calculation is to be performed in parallel over the frontier vertices at every step of the algorithm. As this applies to SSSP, the operation of relaxing a vertex is a functor.

For the sake of performance, Gunrock employs adaptations of a number of work-saving methods suggested by Davidson et al. [7]. The goal of these methods are to find ways of reducing work amount between iterations without significantly adding overhead. [21]

In a 2017 paper by the group from the University of California, Davis which developed Gunrock, a Gunrock implementation of SSSP outperforms the sequential Dijkstra implementation from the Boost Graph Library when memory transfer time is disregarded. [22] In our testing, we have used an example SSSP implementation included in the Gunrock package.

2.5.3 LoneStarGPU

LoneStarGPU is a collection of general purpose GPU applications implemented using the CUDA platform by the ISS group at the University of Texas, in collaboration with Texas State University. Included in the collection is an implementation of the Bellman-Ford algorithm [11], which was referred to as state of the art in 2014 by Davidson et al. [7] This Bellman-Ford implementation is included in our testing.

Chapter 3

Method

Here we present the algorithms evaluated and how they were evaluated. This includes choice of algorithms, input data and how running time is calculated. The evaluation is performed from a consumer hardware perspective, thus all algorithms are tested on a system running Ubuntu 16.04.2 with a NVIDIA 980 Ti GPU and an Intel i7 4770k processor.

3.1 Tested Algorithms

The CPU implementations were picked to each represent a sequential or parallel version of the fundamental algorithms. Note that Delta-Stepping does not have a sequential implementation as it is inherently parallel, nor does it have a GPU implementation as it is deemed unfit for current GPU architectures (as mentioned in section 2.4.3).

Table 3.1: Algorithms tested

Algorithm	Implementation	Abbreviation
Dijkstra, sequential	Boost Graph Library [2]	CPU-Seq D
Dijkstra, CPU parallel [6]	Parallel Boost Graph Library [18]	CPU-Par D
Modified Dijkstra, GPU parallel	Gunrock [22]	GPU-Par D
Bellman-Ford, sequential	Boost Graph Library	CPU-Seq BF
Bellman-Ford, GPU parallel	LoneStarGPU [11]	GPU-Par BF
Delta-Stepping, CPU parallel	Parallel Boost Graph Library	GPU-Par DS

3.2 Test Data

The algorithms were tested on a set of weighted and undirected road network graphs modeled on US cities used in the 9th DIMACS Implementation Challenge [5], a 2005 competition in solving various shortest path problems. They vary in size from approximately 250 000 to 24 000 000 vertices, with each graph containing about twice as many edges as vertices. While the graphs are modeled on actual geographic data, there are errors such as gaps in highways and bridges. [5]

Each graph is a *primal model* of the corresponding geographic location, meaning the graph consists of roads (as edges) with distances (as weights) between junctions (as vertices). As a result of this, the tested graphs do not exhibit scale invariance. [9]

3.3 Procedure

Each algorithm is tested 10 times to produce an average running time for each input graph. The running time is tracked by C++ standard time library Chrono. Start and stop points were inserted manually in the code for each implementation, surrounding actual SSSP execution, as well as, for the GPU versions, GPU memory allocation and transfer of data to and from GPU. This is necessary to enable a fair comparison of CPU and GPU methods, as the data transfer to and from GPU memory can have a large impact on total running time. [8] Scanning input graphs was not included as this is a necessary step for all versions, CPU and GPU. It was therefore excluded to eliminate possible efficiency differences in scanning related to different input formats.

Chapter 4

Results

Figure 4.1 presents a summary of the results of our experiments. The y-axis represents the mean running time of the algorithm and the x-axis represents the number of vertices in the input graph, with each data point representing a graph, denoted by its abbreviation from table **??**.

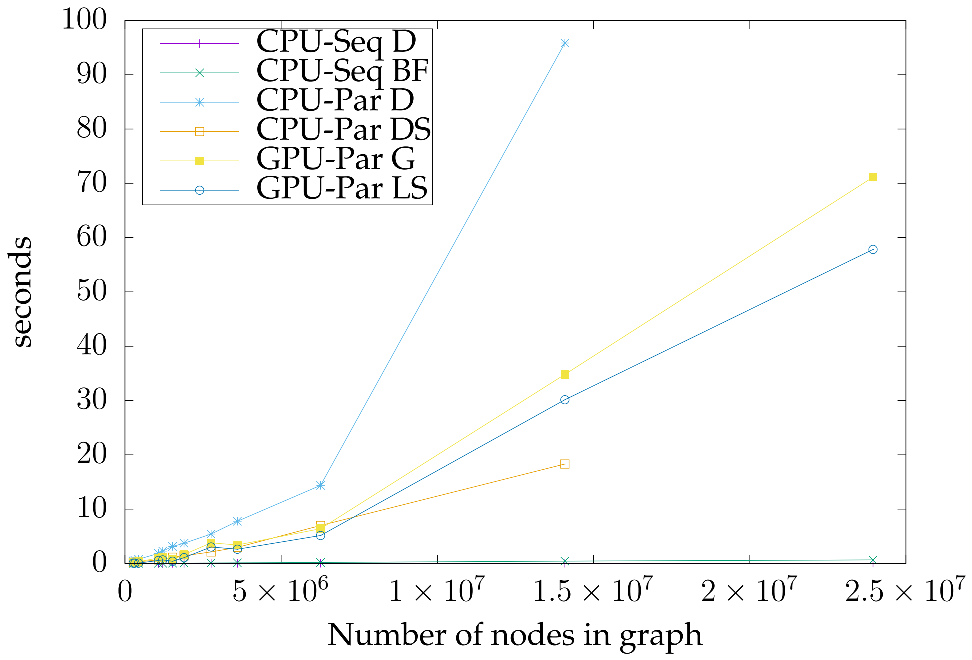

Figure 4.1: Average running time of SSSP algorithms over 10 executions.

In our testing, the sequential implementation of Dijkstra's algorithm proved fastest across all graphs. Both sequential algorithms outperformed all parallel versions. Comparing the two GPU-parallel solutions, we observe that LoneStarGPU was faster than Gunrock on all tested graphs. Of the two tested CPU-parallel implementations, Delta-Stepping performed best, in accordance with what is stated to be true in the general case by the authors of the Parallel Boost Graph Library. [18] With the exception of two of the larger graphs, Central USA and Great Lakes, LoneStarGPU performed better than Delta-Stepping.

Presented below are the results for each graph individually. Note that the number of edges specified refers to undirected edges (two directed edges in opposite directions connecting the same vertex pair are counted as one). Each bar represents the mean running time of the specified algorithm on the graph.

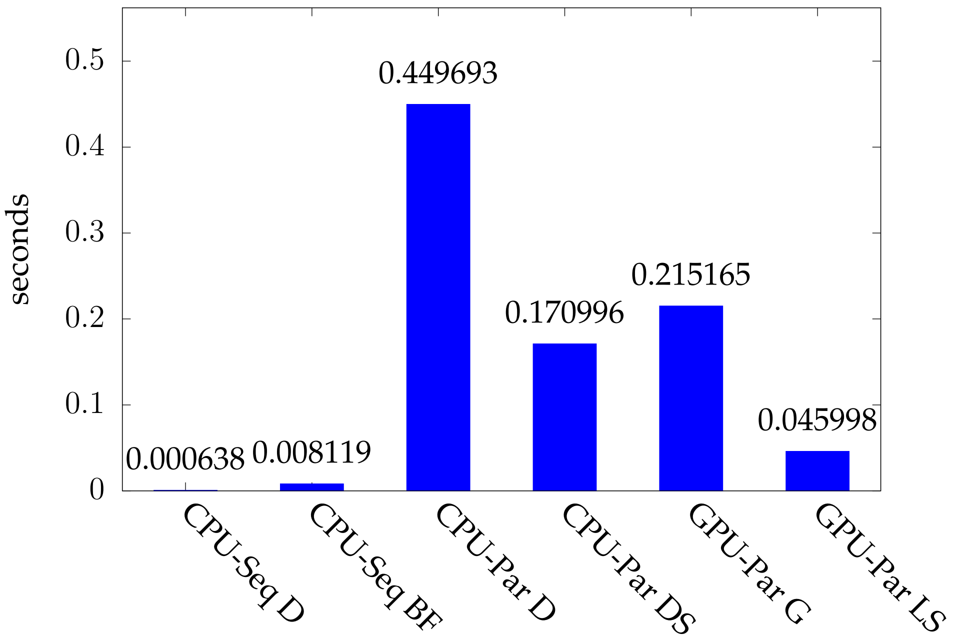

Figure 4.2: New York City (264346 vertices, 365050 edges)

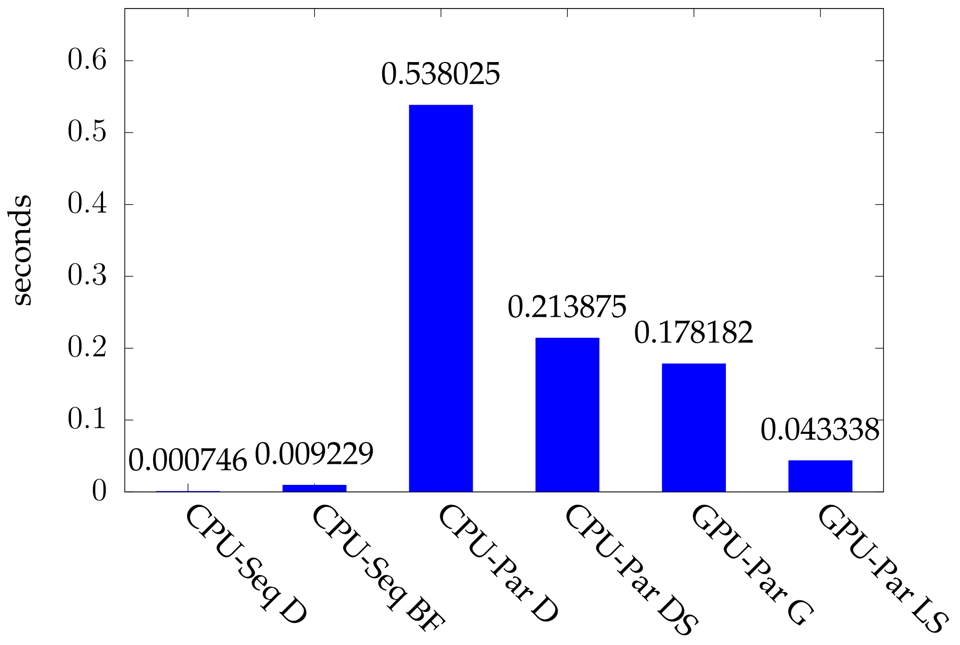

Figure 4.3: San Francisco Bay Area (321270 vertices, 397415 edges)

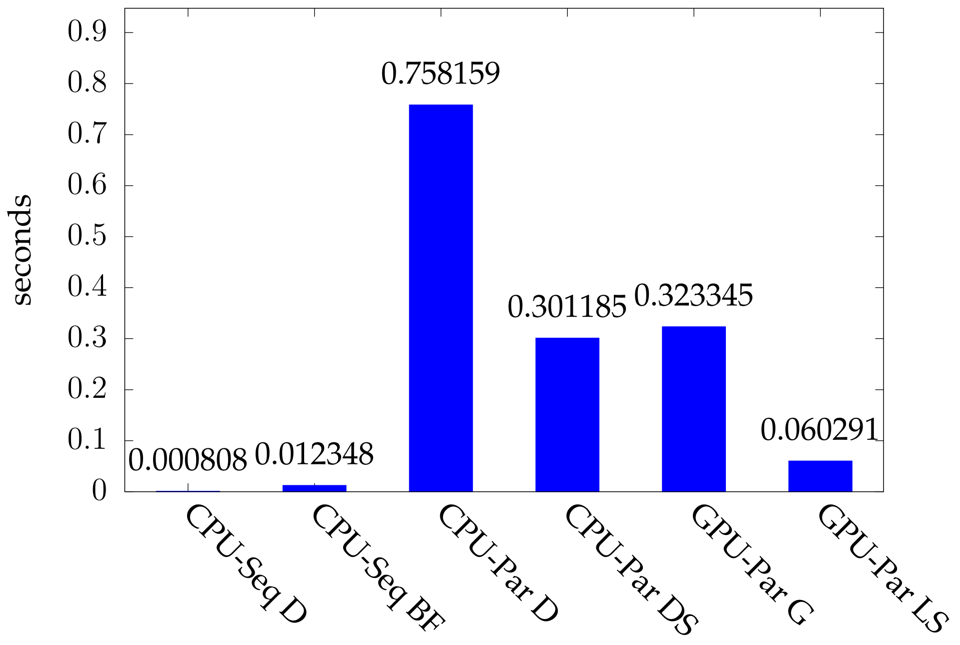

Figure 4.4: Colorado (435666 vertices, 521200 edges)

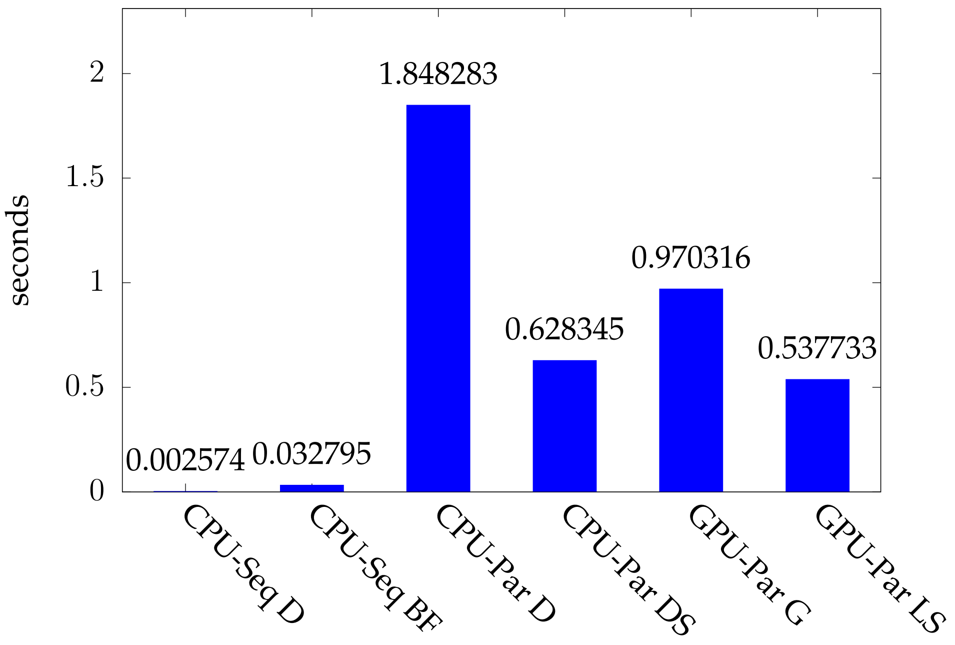

Figure 4.5: Florida (1070376 vertices, 1343951 edges)

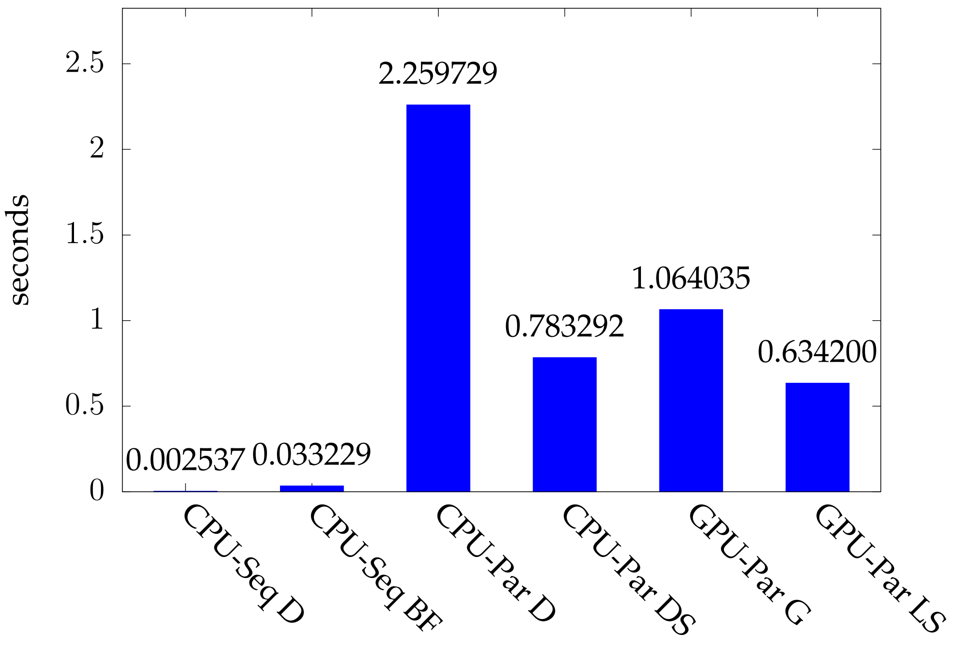

Figure 4.6: Northwest USA (1207945 vertices, 1410387 edges)

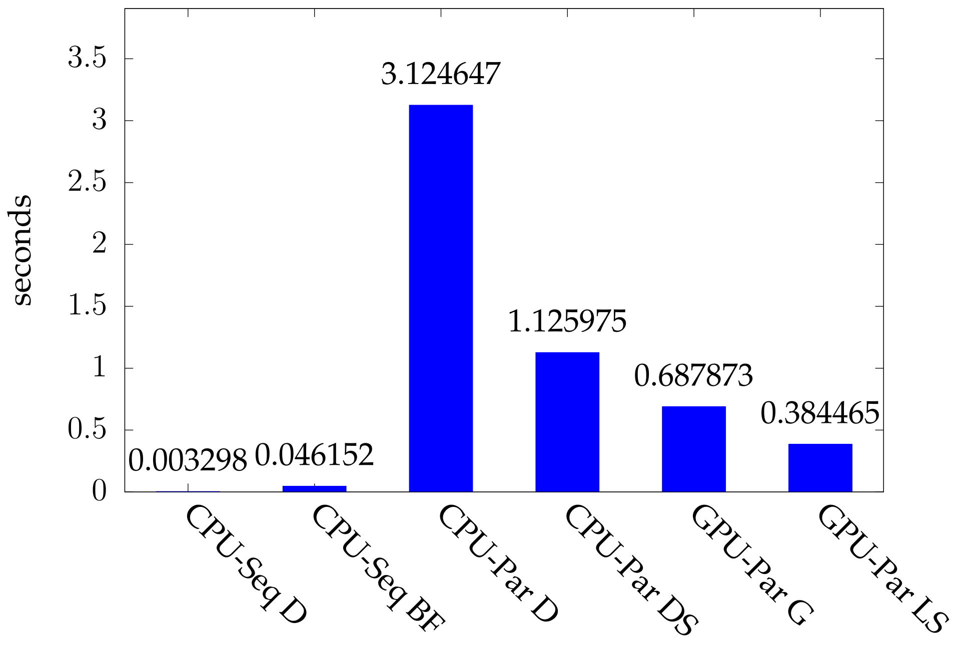

Figure 4.7: Northeast USA (1524453 vertices, 1934010 edges)

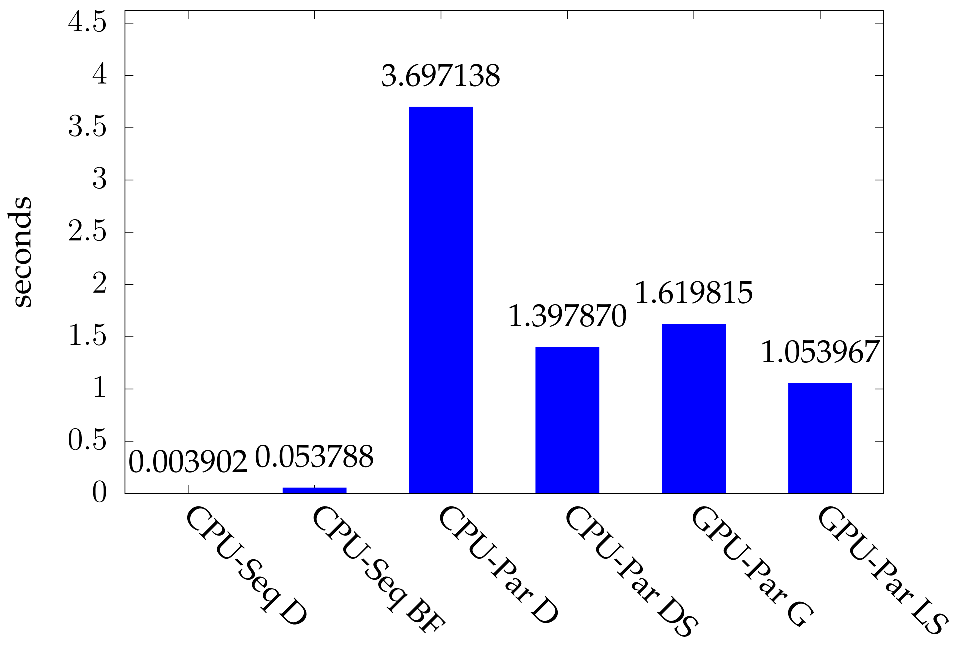

Figure 4.8: California and Nevada (1890815 vertices, 2315222 edges)

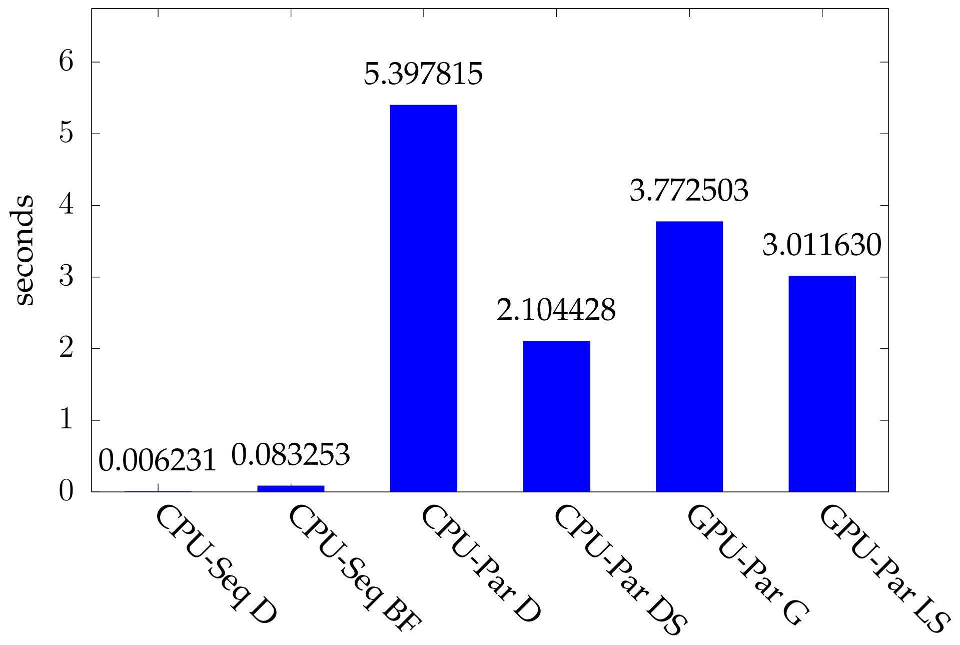

Figure 4.9: Great Lakes (2758119 vertices, 3397404 edges)

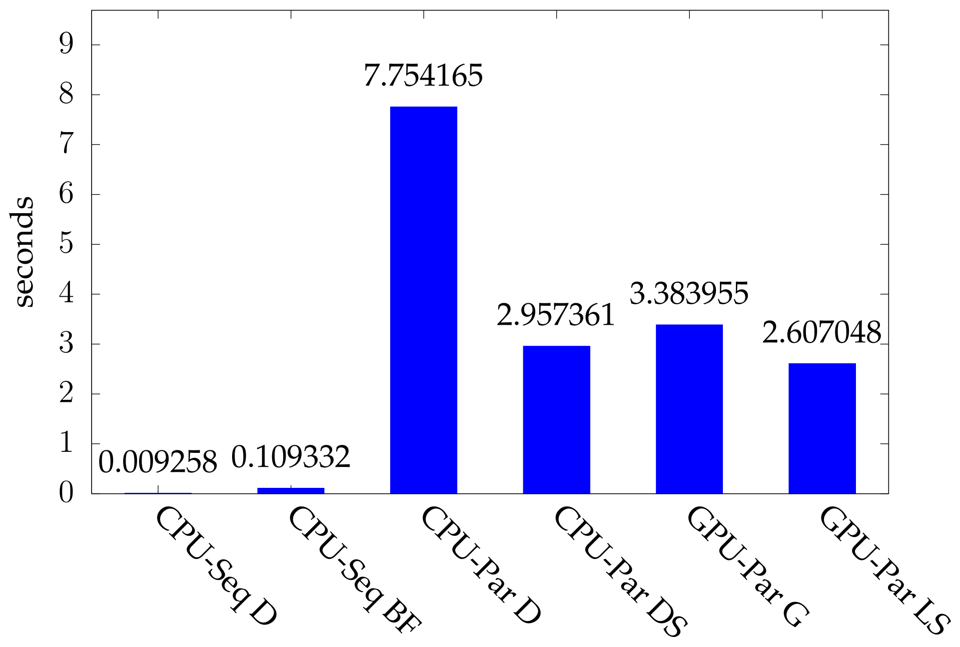

Figure 4.10: Eastern USA (3598623 vertices, 4354029 edges)

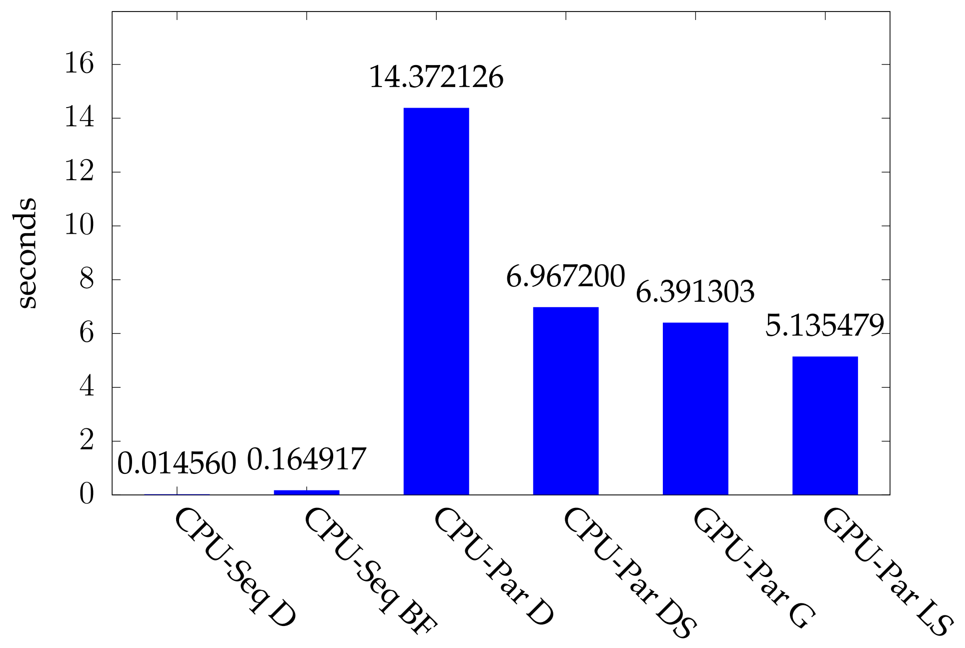

Figure 4.11: Western USA (6262104 vertices, 7559642 edges)

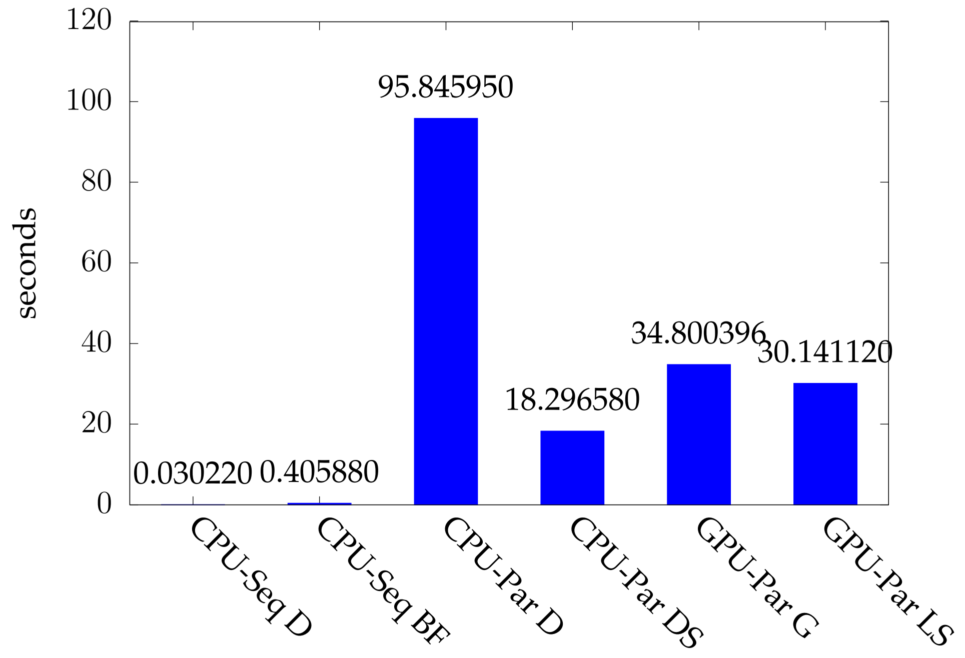

Figure 4.12: Central USA (14081816 vertices, 16933413 edges)

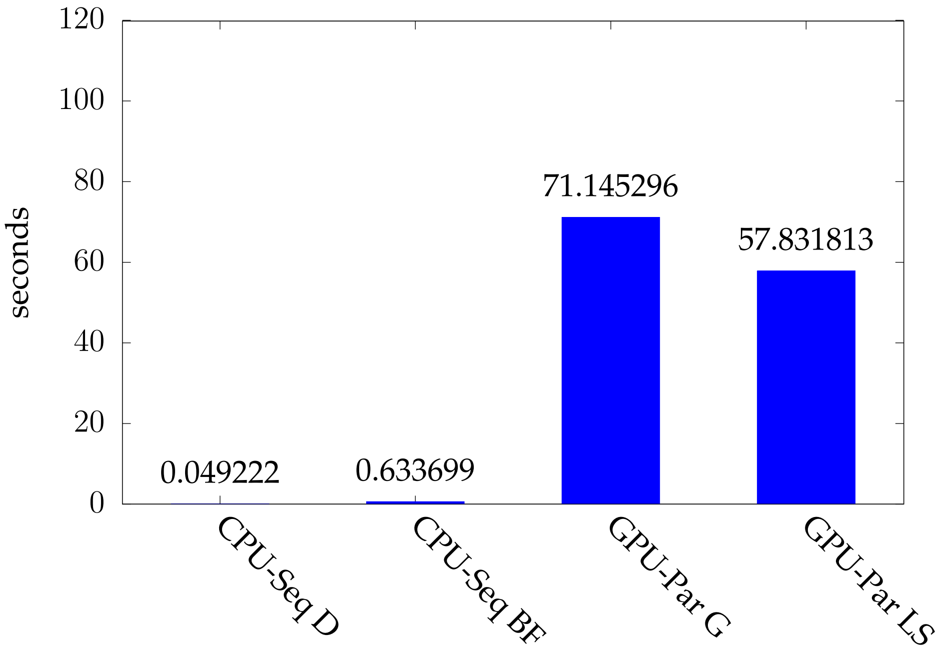

Figure 4.13: Full USA (23947347 vertices, 28854312 edges)

For the largest graph, based on the entire United States, the two CPU-parallel implementations both ran for several hours without reaching completion. They are therefore omitted from figure 4.13.

Chapter 5

Discussion

It should come as no surprise that the sequential Dijkstra's algorithm was faster than the sequential Bellman-Ford algorithm. As explained in the background section, Bellman-Ford performs significantly more work than Dijkstra's algorithm, which is work optimal.

That the sequential Dijkstra's algorithm performed better than both parallel GPU methods can likely be attributed to the large time cost of copying data between GPU and CPU. This also explains the discrepancy between our results, and the results observed by Wang et al., where memory transfer time was not included in timing. [22]

There are a number of factors which could account for the poor performance of the CPU-parallel algorithms from the Parallel Boost Graph Library (Delta-Stepping and Crauser et al.'s parallel Dijkstra). One such factor is that the Boost version of Delta-Stepping lists a number of unimplemented possible optimizations. [4] Furthermore, as explained in section 2.4.3, in order to get the best performance out of Delta-Stepping the parameter Δ (bucket size) should be chosen carefully. In our testing, for each graph we have simply used the graphs maximum edge weight divided by the maximum degree of any of its vertices. This is the default value in the Boost implementation, but the algorithm could very possibly run faster if an individual Δ was determined experimentally for each graph.

The poor performance of Boost's CPU-parallel Dijkstra is partly explained in the Boost documentation, where it is stated that a large amount of work overhead is introduced into the implementation through the use of three separate priority queues. [3]

Also, Boost.MPI, which is used by both CPU-parallel algorithm implementations, has been shown to be a sub-optimal implementation of the MPI standard. [19] Using a different implementation of the message passing interface could thus possibly produce better performance.

In all tested implementations, the actual code tested was derived from SSSP examples supplied by the creators of said implementations. We attempted to modify this example code as little as possible, while still being able to time the methods in a comparable manner. A possible source of errors is unfortunately our own fairly superficial understanding of the inner workings of each implementation, due to research time constraints. This could impact the accuracy of our results in two ways. Firstly, it is possible that the start and stop points of our timing was placed incorrectly, leading to skewed results. Secondly, we were unable to precisely judge whether the examples tested were truly representative of the performance capability of each implementation.

Chapter 6

Conclusion

Our results indicate that data transfer speeds between CPU and GPU constitute a limiting factor in the viability of using GPU methods to solve SSSP on consumer hardware. The performance distribution did not vary significantly across differently sized graphs. In our testing environment, none of the two GPU solutions could solve SSSP more efficiently than the traditional sequential Dijkstra's algorithm.

It was also found that, while the CPU-parallel Delta-Stepping algorithm outperformed both GPU-parallel algorithms on some graphs, none of the CPU-parallel algorithms tested could outperform Dijkstra's algorithm on any of the tested graphs.

Bibliography

[1] *Bellman Ford Shortest Paths - 1.63.0*. URL: `http://www.boost.org/doc/libs/1_63_0/libs/graph/doc/bellman_ford_shortest.html` (visited on 05/28/2017).

[2] *Boost Graph Library*. URL: `http://www.boost.org/doc/libs/1_63_0/libs/graph/doc/index.html` (visited on 05/07/2017).

[3] *Boost Graph Library: Dijkstra's Shortest Paths (No Color Map) - 1.63.0*. URL: `http://www.boost.org/doc/libs/1_63_0/libs/graph/doc/dijkstra_shortest_paths_no_color_map.html` (visited on 05/28/2017).

[4] *boost/graph/distributed/delta_stepping_shortest_paths.hpp - 1.63.0*. URL: `http://www.boost.org/doc/libs/1_63_0/boost/graph/distributed/delta_stepping_shortest_paths.hpp` (visited on 05/24/2017).

[5] Center for Discrete Mathematics Theoretical Computer Science. *9th DIMACS Implementation Challenge - Shortest Paths*. URL: `http://www.diag.uniroma1.it/challenge9/` (visited on 05/10/2017).

[6] A. Crauser et al. "A parallelization of Dijkstra's shortest path algorithm". In: *Mathematical Foundations of Computer Science 1998: 23rd International Symposium, MFCS'98 Brno, Czech Republic, August 24–28, 1998 Proceedings*. Ed. by Luboš Brim, Jozef Gruska, and Jiří Zlatuška. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 722–731. ISBN: 978-3-540-68532-6. DOI: `10.1007/BFb0055823`. URL: `http://dx.doi.org/10.1007/BFb0055823`.

[7] Davidson, Andrew and Baxter, Sean and Garland, Michael and Owens, John D. "Work-efficient parallel GPU methods for single-source shortest paths". In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE. 2014, pp. 349–359.

[8] Chris Gregg and Kim Hazelwood. "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer". In: *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*. IEEE. 2011, pp. 134–144.

[9] Vamsi Kalapala et al. "Scale invariance in road networks". In: *Phys. Rev. E* 73 (2 Feb. 2006), p. 026130. DOI: `10.1103/PhysRevE.73.026130`. URL: `https://link.aps.org/doi/10.1103/PhysRevE.73.026130`.

[10] D. Kirk and W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Applications of GPU Computing Series. Morgan Kaufmann Publishers, 2010. ISBN: 9780123814722. URL: `https://books.google.se/books?id=x8oNlQEACAAJ`.

[11] *LoneStarGPU Collection, Single Source Shortest Path*. URL: `http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu/sssp` (visited on 05/07/2017).

[12] Saeed Maleki et al. "DSMR: A Parallel Algorithm for Single-Source Shortest Path Problem". In: *Proceedings of the 2016 International Conference on Supercomputing*. ACM. 2016, p. 32.

[13] Pedro J. Martín, Roberto Torres, and Antonio Gavilanes. "CUDA Solutions for the SSSP Problem". In: *Computational Science – ICCS 2009: 9th International Conference Baton Rouge, LA, USA, May 25-27, 2009 Proceedings, Part I*. Ed. by Gabrielle Allen et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 904–913. ISBN: 978-3-642-01970-8. DOI: `10.1007/978-3-642-01970-8_91`. URL: `http://dx.doi.org/10.1007/978-3-642-01970-8_91`.

[14] Ulrich Meyer and Peter Sanders. "Δ-stepping: a parallelizable shortest path algorithm". In: *Journal of Algorithms* 49.1 (2003), pp. 114–152.

[15] NVIDIA. *Cuda Parallel Computing Platform*. URL: `http://www.nvidia.com/object/cuda_home_new.html` (visited on 03/27/2017).

[16] *Parallel BGL Dijkstra's Single-Source Shortest Paths - 1.63.0, Crauser et al.'s algorithm*. URL: `http://www.boost.org/doc/libs/1_63_0/libs/graph_parallel/doc/html/dijkstra_shortest_paths.html#crauser-et-al-s-algorithm` (visited on 05/28/2017).

[17] *Parallel BGL Dijkstra's Single-Source Shortest Paths - 1.63.0, Delta-Stepping algorithm*. URL: `http://www.boost.org/doc/libs/1_63_0/libs/graph_parallel/doc/html/dijkstra_shortest_paths.html#delta-stepping-algorithm` (visited on 05/28/2017).

[18] *Parallel Boost Graph Library*. URL: `http://www.boost.org/doc/libs/1_63_0/libs/graph_parallel/doc/html/index.html` (visited on 05/07/2017).

[19] S. Pellegrini, R. Prodan, and T. Fahringer. "A Lightweight C++ Interface to MPI". In: *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. Feb. 2012, pp. 3–10. DOI: `10.1109/PDP.2012.42`.

[20] David W Walker and Jack J Dongarra. "MPI: a standard message passing interface". In: *Supercomputer* 12 (1996), pp. 56–68.

[21] Yangzihao Wang et al. "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*. ACM. 2016, p. 11.

[22] Yangzihao Wang et al. "Gunrock: GPU Graph Analytics". In: *arXiv preprint arXiv:1701.01170* (2017).