International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 09 Issue: 05 | May - 2025 SJIF Rating: 8.586 ISSN: 2582-3930

Optimizing the Bellman-Ford Algorithm Using GPU Parallelization

Dr. Raman Cholla , Rahul Ranjan Kumar, P Jayanth Reddy, Ansh Vajpai, Ashutosh Kumar
JAIN (Deemed-to-be University)
Department of Computer Science and Engineering

ABSTRACT

The Bellman-Ford algorithm's temporal complexity of O(VE) renders it ineffective for big and dense networks, which is a serious computational disadvantage. The need for effective algorithms that can handle large-scale graph structures has increased due to the modern digital landscape's rapid data expansion. The Single Source Shortest Path (SSSP) problem, which entails determining the shortest pathways from a single source vertex to every other vertex in a graph, is one of the most basic and extensively researched topics in graph theory. Numerous real-world applications, including biological network modelling, social media analysis, telecommunications routing, and navigation systems, are based on this issue. The Bellman-Ford algorithm stands out among the other algorithms created to address the SSSP problem because it can handle networks with negative weight edges and detect negative weight cycles, two features that more effective algorithms like Dijkstra's do not allow.

Our project investigates using Graphics Processing Units (GPUs) to parallelise the Bellman-Ford algorithm in order to get around this restriction. With thousands of cores, GPUs are made for extremely parallel calculations and provide significant speedups over conventional CPU-based processing for jobs that can be divided into discrete, independent work units. We developed a GPUaccelerated version of the Bellman-Ford algorithm by utilising NVIDIA's parallel computing platform, CUDA (Compute Unified Device Architecture).

Keywords

Computer Unified Device Architecture (CUDA), Graphics Processing Units (GPUs), and the Single Source Shortest Path (SSSP) problem.

1. INTRODUCTION

Effectively resolving graph-based problems has become essential in today's data-driven world for a variety of applications, such as social networks, biological networks, telecommunications infrastructures, and navigation systems. The Single Source Shortest Path (SSSP) problem, which entails determining the shortest pathways from a given source node to every other node in a graph, is one such basic issue in the fields of graph theory and computer science. Real-time applications like GPS-based route finding, dynamic routing in computer networks, and even social media analytics for determining influencers or link channels depend on a speedy and precise solution to this issue.

Bellman-Ford is unique among the traditional algorithms for this problem because it can handle graphs with negative edge weights and identify negative weight cycles, something that more widely used algorithms like Dijkstra's cannot do. When applied to largescale graphs, like those found in road networks or social media graphs, which might have millions of nodes and connections, its time complexity of O(VE), where V is the number of vertices and E is the number of edges, makes it extremely inefficient.

The goal of this project is to employ GPU parallelisation to optimise the Bellman-Ford method in order to alleviate this performance bottleneck. Originally created for graphics rendering, modern graphics processing units (GPUs) have developed into strong instruments for general-purpose computing, particularly for parallelexecutable workloads. GPUs are perfect for speeding up computations like edge relaxations in the Bellman-Ford algorithm because they have thousands of smaller cores that can handle multiple threads, unlike Central Processing Units (CPUs), which have a limited number of cores optimised for sequential execution.

This project implements a GPU-accelerated version of the BellmanFord technique using NVIDIA's parallel computing platform, CUDA (Compute Unified Device Architecture). The project intends to greatly cut down on calculation time and enhance the scalability of the technique by designating distinct GPU threads to perform edge relaxations concurrently. In order to ensure accuracy and performance, the solution also incorporates essential GPU optimisation techniques including memory coalescing, shared memory utilisation, thread synchronisation, and early termination schemes.

The project is divided into two stages: a parallel implementation using C++ and CUDA and a baseline sequential implementation using Python. The USA road network was one of the synthetic and real-world graph datasets used to evaluate both versions. With speedups of up to 185x over the sequential version, the results unequivocally show that GPU parallelisation results in significant performance gains, particularly for big and dense graphs.

In summary, this project not only overcomes the performance constraints of the conventional Bellman-Ford algorithm but also demonstrates how GPU computing can effectively solve large-scale, real-world graph problems. It creates the framework for next developments in scalable social network analysis, real-time routing systems, and other applications that need quick graph traversal and optimisation.

2. Model of CUDA Programming

There are many threads in the CUDA programming architecture, and these threads combine to create a warp. On a multi-core architecture, a warp is a group of threads that operate in parallel. Similar to how a block is the group of threads that execute on a multiprocessor in a specific amount of time, many blocks can operate in a time-shared fashion on a single multiprocessor. These few blocks are gathered into a grid. Every block in a grid and every thread in a block have their own unique ID. A kernel is a section of code that uses the thread ID to execute on each thread and defines the task in a parallel application. Since all of the device memory is readily accessible to every thread in the block, the multiprocessor operates in a SIMD fashion, similar to a GPU, allowing each thread to run the same kernel over various data. In this approach, using shared memory enhances calculation performanc

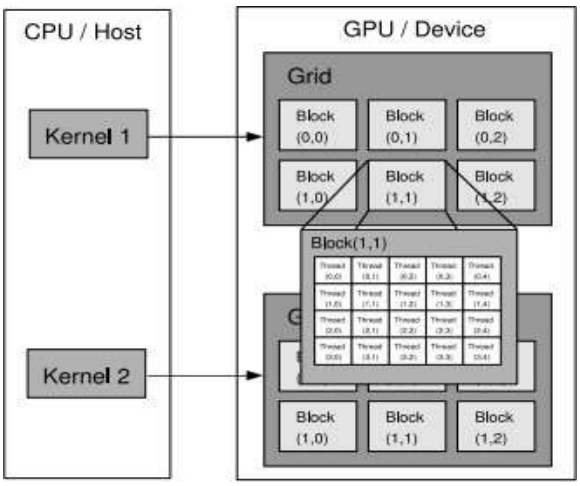


Figure 1: CUDA programming model

3. GRAPH REPRESENTATION ON CUDA

Adjacency matrices are commonly used to represent graphs G (V,E), although they are incredibly inefficient in terms of memory usage for sparse graphs. The adjacency list is a more condensed option that does not store extra zero entries for edges that do not exist. However, the General-Purpose GPU (GPGPU) architecture presents difficulties in effectively modelling adjacency lists on GPUs because each vertex's edge list has a variable size. By enabling the formation of arrays with variable sizes, UDA offers a solution to this restriction and makes it possible to represent graphs in a more space-efficient manner. This method uses a compact adjacency list format to store graphs.

The graph's vertices are kept in a different vertex array (V a), while the adjacency lists are crammed into a single, sizable edge array (E a). In this illustration: The beginning index of each vertex's adjacency list within the edge array (E a) is stored in the vertex array (V a). The adjacency lists are contained in the edge array (E a), where, for every i in V, the edges of vertex i+1 come right after those of vertex i. A direct mapping between the two arrays is created when each entry in $E\ a$ E a refers to a vertex in VaVa. This small adjacency list structure is ideal for large-scale graph analysis on GPUs since it reduces memory overhead and complements CUDA's effective memory management.

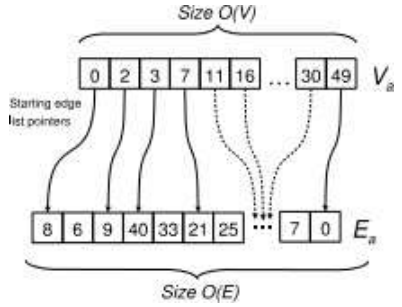


Figure 2: A graph representation where a packed edge list is referenced by a vertex list.

4. BELLMAN- FORD ALGORITHM

A weighted directed graph's shortest paths from a single source vertex to every other vertex are determined via the Bellman Ford algorithm [11], an SSSP discovery algorithm. It employs the relax approach, which substitutes the minimum of the newly computed value and the previous value for the approximate distance to each vertex, which is always larger than or equal to the true distance.
This approach relaxes all of the edges for |V| - 1 times, where |V| is the graph's vertex count. Each node will obtain its shortest distance from the source in this manner node. We can make some changes and parallelise this approach to get faster results because it typically takes a long time. We will demonstrate our parallel Bellman Ford Algorithm implementations in this section. Here, we used CUDA to develop a basic parallel Bellman-Ford algorithm and its improved versions on a GPU.

4.1 Basic Parallel Bellman Ford Algorithm

This algorithm uses CUDA to implement the fundamental Bellman Ford algorithm. The number of threads in this approach is equal to the number of graph edges, which will relax in parallel for |V| iterations. All vertices have their initial costs set to "infinity," with the exception of the start node, whose cost is set to "0." Algorithm1 describes the fundamental BF algorithm.

.

Algorithm 1: BF (G (V, E, W), S)

Create edge_strt_node(Sa), edge_end_node (Ea), edge_weight (Wa) and node_weight (Na) from G (V, E, W)

[1] for each vertex V in parallel do
[2] Invoke INITNODEWEIGHT (Na, S)
[3] end for
[4] for i = 0 toV-1 do
[5] for each edge E in parallel do
[6] Invoke RELAX (Na, Sa, Ea, Wa)
[7] end for
[8] end for

Two kernels are used in the basic Bellman Ford algorithm's implementation.
As demonstrated in Algorithm 2, the node weight of every node except the source node is initialised to "infinity" in the first kernel INITNODEWEIGHT (Ns, S), and it is initialised to "0" for the source node. As demonstrated in Algorithm 3, the second kernel RELAX (Na, Sa, Ea, and Wa) updates the cost of each neighbour if it exceeds the cost of the current vertex plus the edge weight to that neighbour. To prevent concurrent read/write conflicts, this update is carried out atomically using CUDA's atominMin function.

Algorithm 2: INITNODEWEIGHT (Na, S)

[1] id = blockIdx.x*blockDim.x+threadIdx.x
[2] Na[id] = ∞;
[3] if(id == S)
[4] Na[id] = 0;

Algorithm 3: RELAX (Na, Sa, Ea, Wa)

[1] id = blockIdx.x*blockDim.x+threadIdx.x
[2] if Na[Ea[id]]> Na[Sa[id]]+Wa[id]
[3] begin atomic
[4] Na[Ea[id]] ⟵ Na[Sa[id]]+Wa[id]
[5] end atomic
[6] end if

Every iteration of the basic Bellman Ford algorithm attempted to relax every edge. Therefore, if we apply certain criteria to this algorithm, we can lower the computation time that it takes. In light of this, we will introduce the Parallel Bellman Ford algorithm, which uses two flags.

4.2 Parallel Bellman Ford Algorithm using Two Flags

Two flags, F1 and F2, are used in this technique to identify which edges should be relaxed in the following iteration. The number of threads used to implement this technique is equal to the number of edges in the graph, and they should be relaxed |V| times. However, only the edges whose source node weight was changed in the previous iteration will be relaxed in each iteration. Because just a small number of nodes' node weights need to be modified in each iteration, we will be able to drastically cut down on calculation time. Algorithm 4 describes the BF algorithm with two flags.

Algorithm 4: BF_2FLAGS (G (V, E, W), S)

Create edge_strt_node(Sa), edge_end_node (Ea), edge_weight (Wa) , node_weight (Na) from G (V, E, W)

Create Flag (F1) and Flag (F2)

```
[1] for each vertex v in parallel do
[2] Invoke INITNODEWEIGHT (Na, S, F1, F2)
[3] end for
[4] for i=0 toV-1 do
[5] for each edge E in parallel do
[6] Invoke RELAX (Na, Sa, Ea, Wa, F1, F2)
[7] Invoke COPYFLAG (F1, F2)
[8] end for
[9] end for
```

The implementation of the BF algorithm with two flags uses three kernels. As demonstrated in Algorithm 5, the first kernel INITNODEWEIGHT (Na, S, F1, F2) initialises the node weight of each node in parallel, just like in the Basic BF algorithm. Additionally, both flags are initialised to "0" in parallel for every node, with the exception of the source node, for which flag F1 is initialised to 1. The edge relaxation in the second kernel RELAX (Na, Sa, Ea, Wa, F1, F2), as demonstrated in method 6, is carried out in the same way as in the Basic BF method; however, only the edges whose source node was modified in the previous iteration should relax in this case.

Algorithm 5: INITNODEWEIGHT (Na, S, F1, F2)

```
[1] id = blockIdx.x*blockDim.x+threadIdx.x;
[2] Na[id] = ∞;
[3] F1[id] = 0;
[4] F2[id] = 0;
[5] if(id == S)
[6] Na[id] = 0;
[7] F1[id] = 1;
[8] end if
```

Algorithm 6: RELAX (Na, Sa, Ea, Wa, F1, F2)

```
[1] id = blockIdx.x*blockDim.x+threadIdx.x;
[2] if(F1[Sa[id]] == 1)
[3] if (Na[Ea[id]] > Na[Sa[id]]+Wa[id]) [4]
    begin atomic
[5]     Na[Ea[id]]= Na[Sa[id]]+Wa[id];
[6]     end atomic
[7]     F2[Ea[id]] = 1;
[8]     end if
[9]     end if
```

As seen in Algorithm 7, the third kernel, COPYFLAG (F1, F2), finds the next nodes whose outgoing edges should be relaxed by copying the current flag value from one flag to another.

Algorithm 7: COPYFLAG (F1, F2)

```
[1] id = blockIdx.x*blockDim.x+threadIdx.x;
[2] F1[id] = F2[id];
[3] F2[id] = 0;
```

The results of the BF algorithm with two flags are significantly better than those of the Basic BF Algorithm.

5. PERFORMANCE ANALYSIS

On a wide variety of graph statistical data, including sparse, general, and dense directed networks with 8K to 62K vertices and up to.15M edges, we have assessed the performance of concurrent Bellman Ford algorithms. The execution times of these algorithms will be compared. For big graphs, the Bellman-Ford algorithm is inevitably slow due to its O(VE)O(VE)O(VE) time complexity. GPU parallelisation using NVIDIA's CUDA framework was used to increase execution speed. This method greatly accelerated the edge relaxing procedure by distributing computations across thousands of GPU cores.

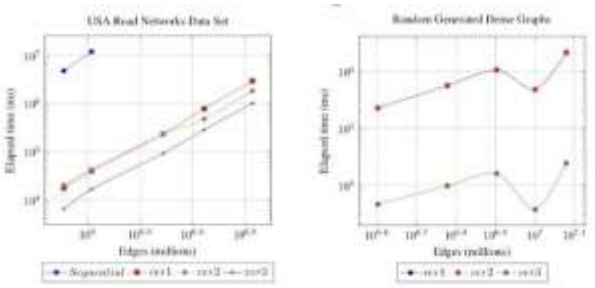
The study also tackled the scaling issues that conventional CPU implementations encounter, which include memory and computational constraints for networks with millions of vertices and edges. The approach was made scalable and effective for practical uses by utilising GPU-based parallelism.
A thorough analysis was carried out to contrast the sequential and parallel implementations.

The investigation, which concentrated on scalability, memory usage, and execution time, showed how much better the GPUaccelerated method is at managing large-scale graphs.
The project had a strong foundation thanks to a thorough literature analysis that looked at current shortest path methods and associated GPU optimisations. To create a baseline for comparison, the Bellman-Ford algorithm's sequential implementation was initially created in its conventional form using Python/C++.

CUDA was subsequently used to parallelise the algorithm, which was then optimised for GPU execution with an emphasis on thread allocation, memory management, and performance tuning. The performance of both systems was evaluated using real-world datasets, such as synthetic random networks and USA Road Network.

The execution times of the sequential and parallel implementations were measured across a range of graph sizes as part of the performance evaluation process, which shed light on the efficiency and speedup improvements made by the parallel approach. To improve performance even more, other optimisation strategies like early termination and shared memory usage were investigated.

This approach produced a structured performance analysis that showed how the Bellman-Ford algorithm's scalability and execution speed were enhanced when it was optimised for GPU execution.



USA Road Networks Dataset Analysis

On the USA road network dataset, the effectiveness of four algorithms—Sequential, ver1, ver2, and ver3—was assessed as the number of edges rose. The Sequential Algorithm's poor scalability for larger datasets was evident from the sharp rise in elapsed time that accompanied the increase in edge count. The ver1 Algorithm, on the other hand, outperformed the sequential version, showing higher scalability with a less abrupt increase in elapsed time.

The ver2 Algorithm proved to be the most effective of the parallel algorithms. It was the most scalable and efficient method for this dataset, continuously maintaining the lowest elapsed times across different edge counts. Although it outperformed ver1, the ver3 Algorithm fell short of ver2's efficiency levels.

Examination of Dense Graphs Produced at Random

The performance of three parallel methods, ver1, ver2, and ver3, was examined for dense graphs with different numbers of edges. Although the duration of the ver1 Algorithm varied, it generally showed an upward trend as the edge count rose. Likewise, the duration of the ver2 Algorithm fluctuated, peaking at about 10 7 10 7 edges, suggesting possible inefficiencies at particular edge counts.

However, the consistency and effectiveness of the ver3 Algorithm made it stand out. The most appropriate option for dense graphs, it maintained noticeably shorter elapsed times than ver1 and ver2. Its resilience and effectiveness in managing densely connected networks are demonstrated by its consistent performance across all edge counts.

6. CONCLUSION

This research demonstrates the revolutionary effect of GPU parallelisation on the Bellman-Ford method by utilising CUDA to obtain a notable speedup of increase to 100x. By leveraging the enormous processing capacity of contemporary GPUs, the parallelised method efficiently divides the task among thousands of cores. This leads to significantly shorter execution times and faster edge relaxation, especially for huge graphs with millions of edges. Coalesced memory access, thread synchronisation, and shared memory usage are important optimisations that help reduce memory latency and increase computational performance. These improvements guarantee peak performance, even for intricate and huge graph datasets. Future research avenues include expanding the system to provide real-time processing for dynamic graphs and using dynamic load balancing to optimise thread allocation and minimise idle GPU threads. This will make it possible for applications like network monitoring and navigation systems to update the shortest paths in real time.

Overall, the findings highlight GPU parallelization's promise for large-scale graph processing applications where scalability and performance are essential for practical implementation, like social network analysis, transportation systems, and network routing.

7. REFERENCES

This section lists the research papers, books, and online resources that were used as references for theoretical and technical insights during the project.

1. Bellman, R. (1958). 'On a Routing Problem.' *Quarterly of Applied Mathematics*, 16(1), 87-90.

2. *Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009).*

3. Delling, D., Goldberg, A. V., Nowatzyk, A., & Werneck, R. F. (2011). 'PHAST: HardwareAccelerated Shortest Path Trees.' *International Symposium on Experimental Algorithms*, Springer. 4. Harish, P., & Narayanan, P. J. (2007). 'Accelerating Large Graph Algorithms on the GPU Using CUDA.' *High-Performance Computing Conference*.

5. Khronos Group. (2022). 'CUDA Programming Guide.' *NVIDIA Official Documentation*.

6. Lindholm, E., Nickolls, J., Oberman, S., & Montrym, J. (2008). 'NVIDIA Tesla: A Unified Graphics and Computing Architecture.' *IEEE Micro*, 28(2), 39-55.

7. NetworkX Developers. (2021). 'NetworkX: High Productivity Software for Complex Networks.' *Python Software Foundation*.

8. Sengupta, S., Harris, M., Zhang, Y., & Owens, J. D. (2007). 'Scan Primitives for GPU Computing.' *ACM Transactions on Graphics*.

9. Thakur, D., & Patel, D. (2020). 'Parallelization of Graph Algorithms Using CUDA.' *International Journal of Computer Science and Network Security*.

10. DIMACS. (2006). 'Shortest Path Implementation Challenge Datasets.'