Accelerating Shortest Path algorithms on GPU

AARYA PATEL

1 INTRODUCTION

A graph is a data structure consisting of a set of vertices or nodes {V}, and a set of edges {E} connecting them. The edges can have weights associated with them, and the nodes can be containers holding multiple variables. As such, graphs are frequently used to represent real-world information; for example, a data center network could be represented as a graph, with each vertex being a machine, edges between the vertices indicating connections and weights representing the bandwidth, latency, cable length, or any combination of the three.

In graph analysis, it is frequently required to find the shortest path from one vertex to another. With small graphs, this can be easy, but with graphs with more vertices and larger degrees (the number of edges that touch a vertex) analysis can be computationally intensive. For time-critical applications (i.e. load balancing on a network to send a packet to another machine on the least congested path), rapid computation of the shortest path is highly desired. In many cases, this means parallelizing the computation amongst multiple processors.

2 LITERATURE REVIEW

Given a weighted graph $G = (V, E)$ and a source vertex s, single source shortest path problem (SSSP) asks the shortest distance from the source vertex to all vertices in the graph [6]. SSSP is a fundamental part of graph theory and has broad applications in road networks, DNA microarrays [5] and many other problems which can be abstracted as graphs. Many algorithms, such as Dijkstra's algorithm and Bellman Ford algorithm, are proposed to solve SSSP.

2.1 Bellman-Ford algorithm

Bellman-Ford algorithm is proposed by Bellman and Ford in 1950s. In the algorithm, each vertex in the graph has an additional attribute d to record the tentative distance from source vertex to itself. The algorithm also employs a technique called relax, which is used to process edges. If we relax an edge (u, v) whose weight is $w(u, v)$, we update the $v.d$ with $min(v.d, u.d + w(u, v))$. The Bellman-Ford algorithm relaxes each edge for $|V| - 1$ times, so the running time is $O(|V||E|)$.

With the development of modern GPU, many parallel versions are proposed. The main parallelism is embedded in the step which relaxes all edges in the graph. Harish et al. [4] proposes to first visit all vertices in parallel and then to relax outgoing edges. In this way, many threads might read and write $v.d$ for some vertex v concurrently, so atomic operation is necessary to avoid race conditions. However, atomic functions serialize the contentious updates from multiple threads, thus increasing the running time. Instead of relaxing leaving edges, Hajela et al. [3] relaxes all incoming edges after visiting all vertices in parallel. For each vertex v, this method first computes the minimum of $u.d + w(u, v)$ for all v's neighbours u, then only this minimum would be used to update $v.d$. In this case, atomic operation is avoided because there is no concurrent read and write by different threads, but either a loop or a nested kernel is introduced.

2.2 Dijkstra's algorithm

Dijkstra's Algorithm is a single-source, shortest-path (SSSP) algorithm. In essence, it takes a graph and a starting vertex, and finds the shortest path to every other vertex (though it is frequently used to find the shortest path to only one other vertex, it does not require significant additional time to compute the shortest path to every

Author's address: Aarya Patel, aaryap@iiitd.ac.in.

other vertex due to the nature of the algorithm; as such, in many implementations it simply computes all the shortest paths from one vertex to another). Harish et al. [4] implement Dijkstra's algorithm on the GPU using two-stage CUDA kernels where the first CUDA kernel updates the cost of each neighbor if greater than the cost of the current vertex plus the edge weight to that neighbor. The second stage CUDA kernel is required to ensure that there is synchronization between CUDA multiprocessors otherwise it will result in race conditions. This problem could even be solved using CUDA atomics introduced in Version 1.1. A PRAM based paralled DIjkstra's algorithm has been proposed by Crauser et al. [2]. Brodal et al. [1] present a parallel priority-based Dijkstra's algorithm implementation that runs in O(n) time on PRAM.

3 MILESTONES

S. No.	Milestone	Member
	Mid evaluation	
1	Implement Bellman-Ford algorithm on CPU	Aarya
2	Design GPU algorithm for Bellman-Ford algorithm	Aarya
3	Implement Bellman-Ford algorithm on GPU	Aarya
	Final evaluation	
4	Implement Dijkstra's algorithm on CPU	Aarya
5	Design GPU algorithm for Dijkstra's algorithm	Aarya
6	Implement Dijkstra algorithm on GPU	Aarya
7	Profiling and Performance Evaluation of CPU and GPU implementations	Aarya

4 APPROACH

4.1 Graph Representation

The graph $G(V, E)$ can be represented using either an adjacency matrix or an adjacency list. However, for sparse graphs like road networks, the adjacency list is preferred due to its space efficiency. One alternative form of an adjacency list is the Compressed Sparse Row (CSR) representation, which packs the lists of vertices into a single large array. This representation has been found to be suitable for CUDA implementation, as mentioned in references [4]. The CSR representation utilizes four arrays: a vertex array V to store all the vertices, an index array I to store the starting positions of the adjacency lists of edges for each $V[i]$, an edge array E, and a weight array W to store the weights of each edge. The number of edges for each $V[i]$ can be determined by the difference between $I[i+1]$ and $I[i]$.

4.2 GPU and CUDA

The growing utilization of general-purpose computing on graphics processing units (GPGPU) has grown significantly, providing us with extensive parallel computing capabilities. The GPGPU uses a single instruction, multiple threads (SIMT) execution model where each thread executes identical code. Nvidia has developed a parallel computing platform and APIs called Compute Unified Device Architecture (CUDA) that allows for leveraging GPU parallelism for general-purpose computing while maintaining performance. CUDA is built on industry-standard C++ and comprises a small number of extensions to facilitate heterogeneous programming.

In the context of GPUs, SM (Streaming Multiprocessor) refers to a key component responsible for executing parallel computations. It consists of multiple SPs (Streaming Processors), which are individual processing units capable of executing instructions concurrently. Each SP can execute multiple threads simultaneously, with each

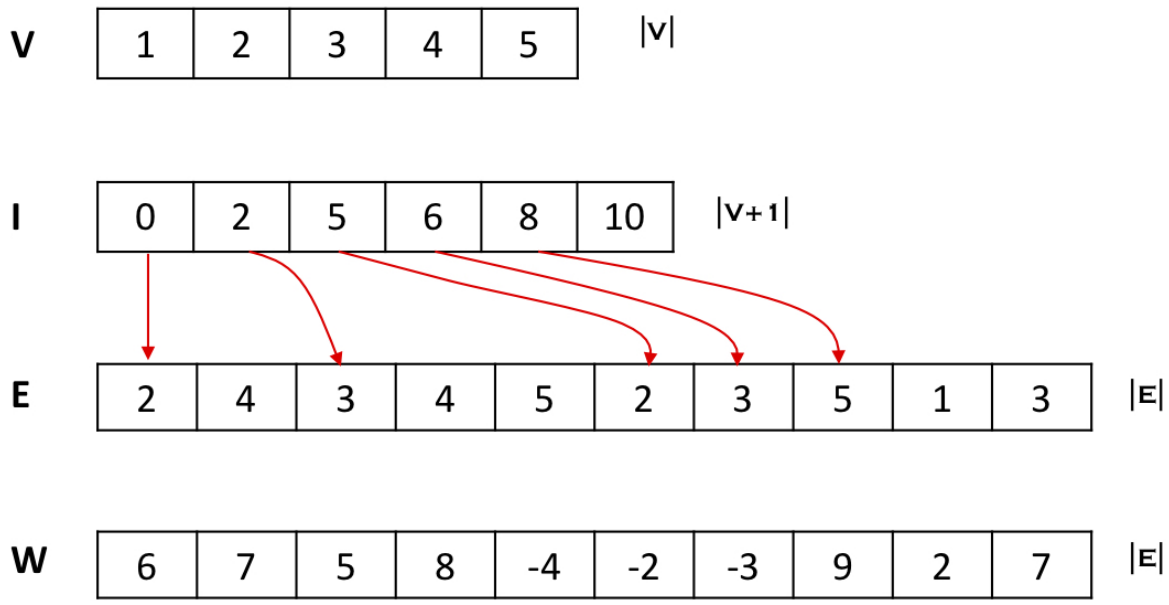

Fig. 1. Compressed Sparse Row (CSR) representation

thread performing a specific task or computation. Threads are lightweight units of execution that are created and managed by the GPU, allowing for concurrent processing of multiple tasks in parallel. The combination of SMs, SPs, and threads in a GPU enables efficient and powerful parallel computing, making GPUs highly suitable for tasks that require massive parallelism, such as deep learning, scientific simulations, and image processing.

4.3 Bellman Ford Sequential Algorithm

The Bellman-Ford algorithm is a shortest path algorithm used to find the shortest path from a source vertex to all other vertices in a directed graph with or without negative-weight edges. The algorithm starts by initializing the distance estimates for all vertices, setting the distance from the source vertex to itself as 0 and the distance to all other vertices as infinity. Then, it iteratively relaxes all edges for a fixed number of iterations, updating the distance estimates based on the current shortest path estimates. After the iterations, the algorithm checks for the presence of negative-weight cycles. If there are no negative-weight cycles, the algorithm outputs the final shortest path estimates, which represent the shortest distances from the source vertex to all other vertices in the graph.

Consider the following directed graph $G(V, E)$ with vertex set $V = A, B, C, D, E$ and edge set $E = (A, B, 3), (A, C, 5), (B, C, -2), (B, D, 4), (C, D, 1), (C, E, 6), (D, B, 2), (D, E, 4)$. The numbers in parentheses represent the edge weights.

We want to find the shortest path from a source vertex s to all other vertices in the graph. Let's assume $s = A$, so we want to find the shortest path from vertex A to all other vertices.

Step 1: Initialization We initialize the distance from s to all other vertices as ∞, except for the distance from s to itself, which is set to 0. So we have $d(A) = 0$ and $d(B) = d(C) = d(D) = d(E) = \infty$.

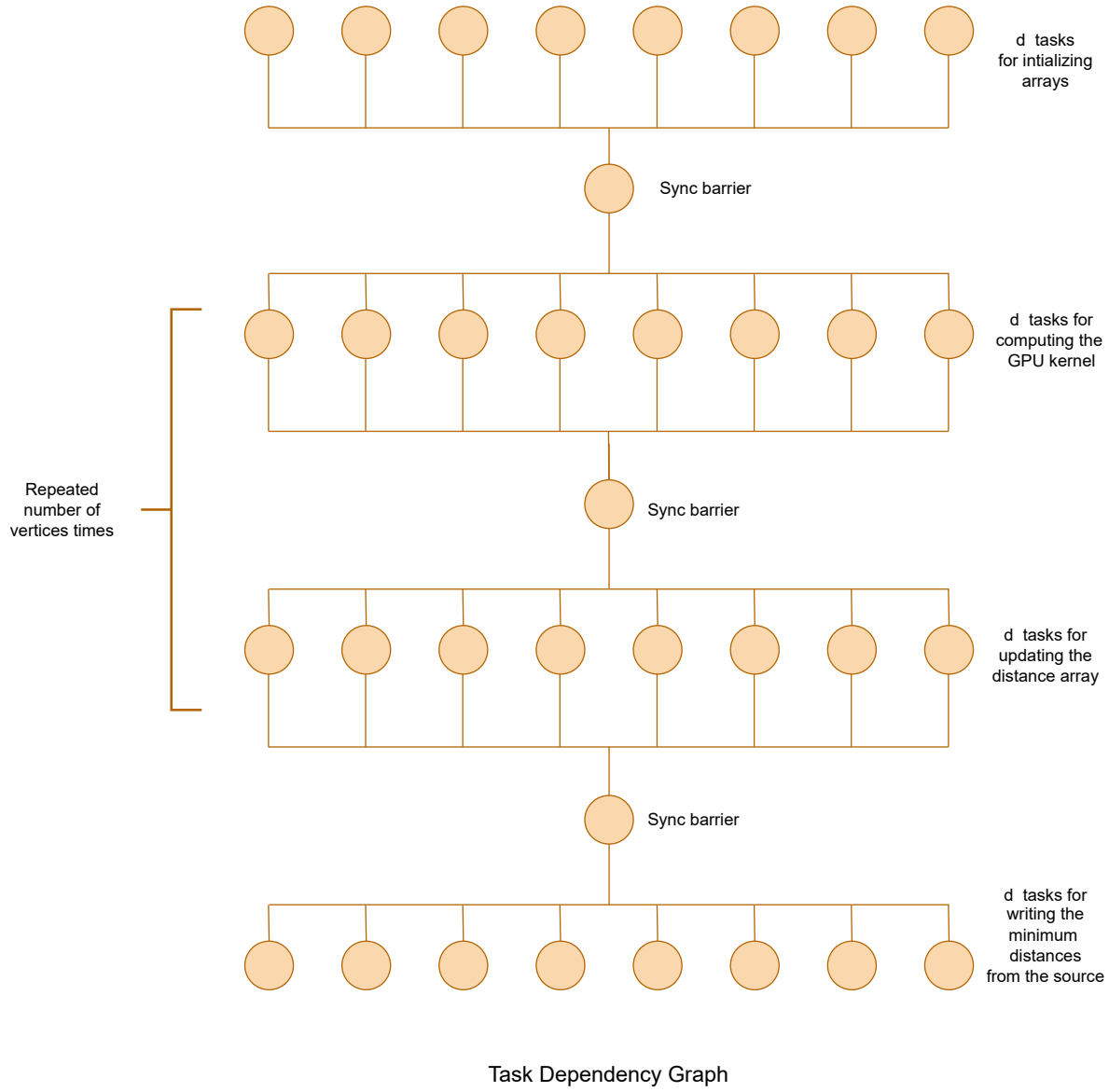

Fig. 2. Task Dependency Graph for finding the shortest path.

Step 2: Relaxation We perform relaxation on all edges $|V| - 1$ times, where $|V|$ is the number of vertices in the graph. In each iteration, we update the distance estimates for all vertices based on the current shortest path estimates.

Iteration 1:

- For edge (A, B): $d(A) + w(A, B) = 0 + 3 < d(B)$, so we update $d(B) = 3$.

- For edge (A, C): $d(A) + w(A, C) = 0 + 5 < d(C)$, so we update $d(C) = 5$.

Iteration 2:

- For edge (B, C): $d(B) + w(B, C) = 3 - 2 < d(C)$, so we update $d(C) = 1$.
- For edge (B, D): $d(B) + w(B, D) = 3 + 4 < d(D)$, so we update $d(D) = 7$.

Iteration 3:

- For edge (C, D): $d(C) + w(C, D) = 1 + 1 < d(D)$, so we update $d(D) = 2$.
- For edge (C, E): $d(C) + w(C, E) = 1 + 6 < d(E)$, so we update $d(E) = 7$.

Step 3: Output The final shortest path estimates are: $d(A) = 0$, $d(B) = 3$, $d(C) = 1$, $d(D) = 2$, and $d(E) = 7$. These values represent the shortest distances from vertex A to all other vertices in the graph.

Algorithm 1 Bellman-Ford Algorithm

Input: Graph $G(V, E)$ with source vertex s and edge weights $w(e)$
Output: Shortest distance $d(v)$ from s to all vertices $v \in V$

```
for each vertex v ∈ V do
    d(v) ← ∞;
end for
d(s) ← 0;
for i ← 1 to |V| − 1 do
    for each edge (u, v) ∈ E do
        if d(u) + w(u, v) < d(v) then
            d(v) ← d(u) + w(u, v);
        end if
    end for
end for
    return d;
```

4.4 Bellman-Ford Parallel CUDA Implementation

In this first approach, we implement the Bellman-Ford algorithm on CUDA. We use a monolithic kernel where each vertex in the graph is assigned to a separate thread. Each thread is responsible for relaxing all the outgoing edges of the vertex it was assigned to, using its thread ID to identify the vertex. The number of GPU blocks is determined during runtime based on the number of vertices in the input graph. However, this approach assumes that we have enough threads to cover all the vertices in the input graph and that each thread can handle the relaxation of edges for its corresponding vertex. The algorithm for the approach is presented as follows.

Algorithm 2 Parallel Bellman-Ford Algorithm

Create vertex array V, edge array E, edge offset O and weight array W from $G(V, E, W)$
Create cost array C and Updating cost array U of size V
Initialize cost array C and Updating cost array U to ∞
$C[0] \leftarrow \infty, U[0] \leftarrow \infty$

for $n \leftarrow 1$ to $|V|$ **do**
    Invoke CUDA_BELLMAN_KERNEL(V, E, O, W, C, U)
    Invoke CUDA_UPDATEDIST_KERNEL(V, E, O, W, C, U)
end for

Algorithm 3 CUDA_BELLMAN_KERNEL(V, E, O, W, C, U)

$tid \leftarrow getThreadID$

for $j \leftarrow O[tid]$ to $O[tid + 1]$ **do**
    $w \leftarrow W[j]$
    $du \leftarrow C[tid]$
    $updated\_dist \leftarrow w + du$
    if $du! = INT\_MAX$ **then**
        $atomicMin(\&U[E[j]], updated\_dist)$;
    end if
end for

Algorithm 4 CUDA_UPDATEDIST_KERNEL(V, E, O, W, C, U)

$tid \leftarrow getThreadID$

if $U[tid] < C[tid]$ **then**
    $C[tid] \leftarrow U[tid]$
end if$U[tid] \leftarrow C[tid]$

4.5 Optimized Bellman-Ford CUDA Implementation

We optimize the parallel implementation of the Bellman-Ford algorithm on CUDA in our second approach. Within this, we use the grid stride approach in the CUDA kernel that ensures that no thread is idle and that each thread performs an equal amount of work. The stride is calculated using $blockDim.x * gridDim.x$, which is equal to the total number of threads in the grid. This grid stride loop approach offers scalability and thread reuse. This helps maximize the utilization of GPU resources and enhances the overall performance of the algorithm. The algorithm for the approach is presented below.

Algorithm 5 Optimized Parallel Bellman-Ford Algorithm

Create vertex array V, edge array E, edge offset O and weight array W from $G(V, E, W)$
Create cost array C and Updating cost array U of size V
Initialize cost array C and Updating cost array U to ∞
$C[0] \leftarrow \infty, U[0] \leftarrow \infty$

for $n \leftarrow 1$ to $|V|$ **do**
    Invoke CUDA_BELLMAN_KERNEL_GRID_STRIDES(V, E, O, W, C, U)
    Invoke CUDA_UPDATEDIST_KERNEL_GRID_STRIDES(V, E, O, W, C, U)
end for

Algorithm 6 CUDA_BELLMAN_KERNEL_GRID_STRIDES(V, E, O, W, C, U)

$tid \leftarrow getThreadID$
$stride \leftarrow getStrides$

for $index \leftarrow tid$ to $|V|$ by $stride$ **do**
    for $j \leftarrow O[tid]$ to $O[tid + 1]$ **do**
        $w \leftarrow W[j]$
        $du \leftarrow C[tid]$
        $updated\_dist \leftarrow w + du$
        if $du! = INT\_MAX$ **then**
            $atomicMin(\&U[E[j]], updated\_dist)$;
        end if
    end for
end for

Algorithm 7 CUDA_UPDATEDIST_KERNEL_GRID_STRIDES(V, E, O, W, C, U)

$tid \leftarrow getThreadID$
$stride \leftarrow getStrides$

for $index \leftarrow tid$ to $|V|$ by $stride$ **do**
    if $U[tid] < C[tid]$ **then**
        $C[tid] \leftarrow U[tid]$
    end if
    $U[tid] \leftarrow C[tid]$
end for

4.6 Dijkstra's Algorithm

Dijkstra's algorithm is a graph traversal algorithm that finds the shortest path between two nodes in a weighted graph. It works by starting from the source vertex and gradually exploring its neighbors, keeping track of the distance from the source to each neighbor. This process is repeated for each unexplored neighbor, selecting the one with the smallest distance as the next vertex to explore. As each vertex is explored, the algorithm updates

the distances to its neighbors, based on the sum of the current distance to the vertex and the weight of the edge connecting them. The algorithm continues to explore vertices until the destination vertex is reached or all reachable vertices have been explored. Dijkstra's algorithm is widely used in many applications, including network routing and geographic mapping. The algorithm for Dijkstra's is presented as follows:

Algorithm 8 Dijkstra's Algorithm

Input: Graph $G(V, E)$ with source vertex s and edge weights $w(e)$
Output: Shortest distance $d(v)$ from s to all vertices $v \in V$

```
for each vertex v ∈ V do
    d(v) ← ∞;
end for
d(s) ← 0;
Initialize set S of visited vertices as empty;
Initialize priority queue Q;
Insert s into Q with priority 0;

while Q is not empty do
    Extract vertex v with minimum distance from Q
    Add v to S;
    for each neighbor u of v do
        if d(v) + w(v,u) < d(u) then
            d(u) ← d(v) + w(v,u);
            insert u into Q with priority equal to d(v) + w(v,u);
        end if
    end for
end while
     return d;
```

4.7 Dijkstra's Parallel CUDA Implementation

In this first approach, we implement Dijkstra's algorithm on CUDA. We use a monolithic kernel where each vertex in the graph is assigned to a separate thread. Each thread is responsible for relaxing all the outgoing edges of the vertex it was assigned to, using its thread ID to identify the vertex.

In our implementation of the algorithm, we utilize several arrays to represent the graph and the current state of the algorithm. The vertex array V and the edge array E store the vertices and edges of the graph, respectively. We also use a boolean mask M of size $|V|$ to keep track of which vertices have already been processed by the algorithm, and a weight array W of size $|E|$to store the weights of each edge.

In each iteration of the algorithm, each vertex checks whether it is in the mask M. If it is, the vertex fetches its current cost from the cost array C and its neighbor's weights from the weight array W. The cost of each neighbor is then updated if it is greater than the cost of the current vertex plus the edge weight to that neighbor. The new cost is not immediately reflected in the cost array but instead is updated in an alternate array U.

At the end of the kernel execution, a second kernel is launched to compare the cost array C with the updating cost array U. If the cost in U is less than the cost in C, the cost in C is updated with the value from U, and the corresponding vertex is added to the mask M. The updating cost array reflects the cost array after each kernel execution for consistency.

This approach allows for parallel processing of the vertices and ensures that the algorithm correctly computes the shortest path from the source vertex to all other vertices in the graph.

Algorithm 9 Parallel Dijkstra's Algorithm

Create vertex array V, edge array E, edge offset O and weight array W from $G(V, E, W)$
Create mask array M, cost array C and Updating cost array U of size V
Initialize mask array M to $false$, cost array C and Updating cost array U to ∞
$M[0] \leftarrow true, C[0] \leftarrow \infty, U[0] \leftarrow \infty$

while M is not empty **do**
  for each Vertex V in parallel **do**
    Invoke CUDA_DIJKSTRAS_KERNEL(V, E, O, W, M, C, U) on the grid
    Invoke CUDA_UPDATEDIST_KERNEL(V, E, O, W, M, C, U) on the grid
  end for
end while

Algorithm 10 CUDA_DIJKSTRAS_KERNEL(V, E, O, W, M, C, U)

$tid \leftarrow getThreadID$
if $M[tid]$ **then**
  $M[tid] \leftarrow false$
  for $j \leftarrow O[tid]$ to $O[tid + 1]$ **do**
    $w \leftarrow W[j]$
    $du \leftarrow C[tid]$
    $updated\_dist \leftarrow w + du$
    $atomicMin(\&U[E[j]], updated\_dist)$;
  end for
end if

Algorithm 11 CUDA_UPDATEDIST_KERNEL(V, E, O, W, M, C, U)

$tid \leftarrow getThreadID$

if $U[tid] < C[tid]$ **then**
  $C[tid] \leftarrow U[tid]$
  $M[tid] \leftarrow true$
end if$U[tid] \leftarrow C[tid]$

4.8 Optimized Dijkstra's CUDA Implementation

We optimize the parallel implementation of Dijkstra's algorithm on CUDA in our second approach. Within this, we use the grid stride approach in the CUDA kernel that ensures that no thread is idle and that each thread performs an equal amount of work. The stride is calculated using $blockDim.x * gridDim.x$, which is equal to the total number of threads in the grid. This grid stride loop approach offers scalability and thread reuse. This

helps maximize the utilization of GPU resources and enhances the overall performance of the algorithm. The algorithm for the approach is presented below.

Algorithm 12 Optimized Parallel Dijkstra's Algorithm

Create vertex array V, edge array E, edge offset O and weight array W from $G(V, E, W)$
Create cost array C and Updating cost array U of size V
Initialize mask array M to $false$, cost array C and Updating cost array U to ∞
$M[0] \leftarrow true, C[0] \leftarrow \infty, U[0] \leftarrow \infty$

while M is not empty **do**
  for each Vertex V in parallel **do**
    Invoke CUDA_DIJKSTRAS_KERNEL_GRID_STRIDES(V, E, O, W, M, C, U)
    Invoke CUDA_UPDATEDIST_KERNEL_GRID_STRIDES(V, E, O, W, M, C, U)
  end for
end while

Algorithm 13 CUDA_DIJKSTRAS_KERNEL_GRID_STRIDES(V, E, O, W, M, C, U)

$tid \leftarrow getThreadID$
$stride \leftarrow getStrides$

for $index \leftarrow tid$ to $|V|$ by $stride$ **do**
  if $M[tid]$ **then**
    $M[tid] \leftarrow false$
    for $j \leftarrow O[tid]$ to $O[tid+1]$ **do**
      $w \leftarrow W[j]$
      $du \leftarrow C[tid]$
      $updated\_dist \leftarrow w + du$
      $atomicMin(\&U[E[j]], updated\_dist)$;
    end for
  end if
end for

Algorithm 14 CUDA_UPDATEDIST_KERNEL_GRID_STRIDES(V, E, O, W, C, U)

$tid \leftarrow getThreadID$
$stride \leftarrow getStrides$

for $index \leftarrow tid$ to $|V|$ by $stride$ **do**
  if $U[tid] < C[tid]$ **then**
    $C[tid] \leftarrow U[tid]$
    $M[tid] \leftarrow true$
  end if
  $U[tid] \leftarrow C[tid]$
end for

Number of Vertices	CPU time (in ms)	GPU time (in ms)	GPU with Grid Stride time (in ms)
1000	30.13	4.06	4.05
10000	2691.81	41.84	41.92
100000	279466.56	1330.69	1306.15
200000	1471912.75	6260.26	6191.79
400000	4774133.50	19044.46	18373.78
600000	10862620	38713.35	39020.69
800000	18911958	62427.45	61076.81
1000000	338496098.13	1056117.12	1041526.45

Table 1. Time taken by CPU, GPU and GPU optimized code by Bellman-Ford Algorithm

5 RESULTS

In this section, we show the speedups achieved by our GPU and optimized GPU kernels over the CPU implementation.

Table 1 shows the time taken by CPU, GPU and GPU with grid strides to compute the single source shortest path using the Bellman-Ford algorithm. It can be easily observed that GPU implementations outperform the sequential CPU implementation by a big margin. The number of vertices varies from 1K to 1M and has a max out-degree of 2 per vertex. While it can be observed that optimized GPU implementation ie. GPU with grid strides performs better than normal GPU implementation on all the number of vertices. This shows that our optimized GPU approach scales well as the size of the graph is increased.

Similarly, table 2 displays the time taken by CPU, GPU and GPU with grid strides to compute the single source shortest path using Dijkstra's algorithm. Here as well, we infer that the GPU implementations outperform the sequential CPU implementation by a big margin. The number of vertices varies from 1K to 10M and has a max out-degree of 2 per vertex. Moreover, it can be observed that in most cases, the GPU with grid-stride approach outperforms vanilla GPU implementation.

Figure 3 displays the speedups we get on our GPU and GPU with the grid stride optimized method using the Bellman-Ford algorithm. It can be empirically evaluated that the speedup increases as we increase the number of vertices. Moreover, the speedup achieved by our optimized GPU kernel is slightly more than the vanilla GPU implementation for all the vertices in the graph. The number of vertices ranges from 1K to 1M.

Figure 4 displays the speedups we get on our GPU and GPU with the grid stride optimized method using Dijkstra's algorithm. We observe that speedup sharply increases from 1K to 200K vertices to close to 50. Then it decreases by half for 400K vertices and then finally linearly increases from 400K to 10M nodes in the graph.

We conclude that our optimized GPU implementation performs the best overall for any number of vertices and scales efficiently as the number of vertices are increased.

Table 3 shows the time achieved by other open-source implementations of the Bellman-Ford algorithm on GitHub. It is easily observed that our approach beats the open-source implementation on all the metrics by a fair amount. This is primarily due to their implementation using an adjacency matrix to represent the graph which isn't efficient on GPUs as it consumes a lot of memory. On the other hand, our approach uses a much more compact representation for graphs.

REFERENCES

[1] G.S. Brodal, J.L. Traff, and C.D. Zaroliagis. 1997. A parallel priority data structure with applications. In *Proceedings 11th International Parallel Processing Symposium*. 689–693. https://doi.org/10.1109/IPPS.1997.580979

Number of Vertices	CPU time (in ms)	GPU time (in ms)	GPU with Grid Stride time (in ms)
1000	0.4616	0.396	0.370
10000	4.637	0.825	0.818
100000	56.515	3.206	3.251
200000	198.383	3.935	4.120
400000	285.907	12.066	11.881
600000	459.667	18.303	18.236
800000	620.702	22.239	22.079
1000000	783.978	26.165	26.127
5000000	4637.879	126.779	125.528
10000000	10441.888	249.553	250.763

Table 2. Time taken by CPU, GPU, and GPU optimized code by Dijkstra's Algorithm

Method	CPU time (in ms)	GPU time (in ms)	Speedup
qiansunn-github	104.83	49.21	2.13
Our approach	**30.13**	**4.056**	**7.43**

Table 3. Comparison of CPU time, GPU time and Speedup with other open-source implementations of Bellman-Ford algorithm for 1K vertices

Method	GPU time (in ms)
jfmartinez-gihub	2029.304
Our approach	**0.370144**

Table 4. Comparison of CPU time, GPU time and Speedup with other open-source implementations of Dijkstra's algorithm for 1K vertices

[2] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. 1998. 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 23*. Springer, 722–731.

[3] Gaurav Hajela and Manish Pandey. 2014. Article: Parallel Implementations for Solving Shortest Path Problem using Bellman-Ford. *International Journal of Computer Applications* 95, 15 (June 2014), 1–6. Full text available.

[4] Pawan Harish and P. J. Narayanan. 2007. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *High Performance Computing – HiPC 2007*, Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 197–208.

[5] Kwonmoo Lee, Ju Han Kim, Tae Su Chung, Byoung-Sun Moon, Hoseung Lee, and I.S. Kohane. 2001. Evolution strategy applied to global optimization of clusters in gene expression data of DNA microarrays. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, Vol. 2. 845–850 vol. 2. https://doi.org/10.1109/CEC.2001.934278

[6] F. Benjamin Zhan and Charles E. Noon. 1998. Shortest Path Algorithms: An Evaluation Using Real Road Networks. *Transportation Science* 32, 1 (1998), 65–73. https://doi.org/10.1287/trsc.32.1.65

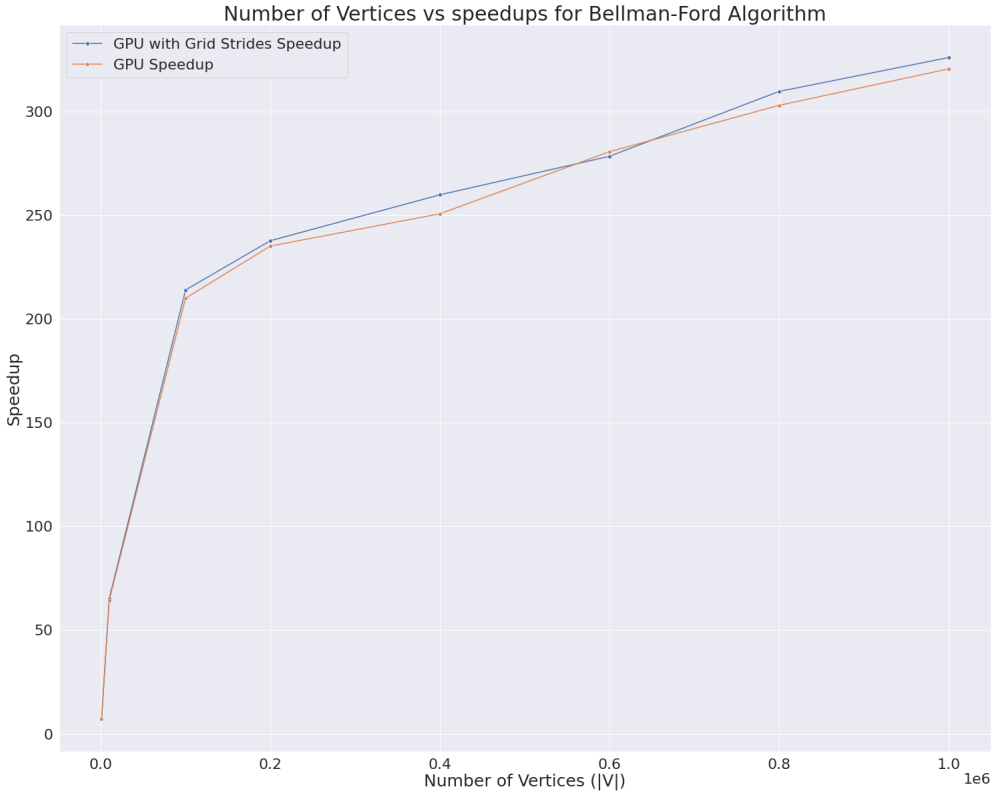

Fig. 3. Speedups achieved by GPU and GPU with grid strides over different numbers of vertices by Bellman-Ford algorithm.

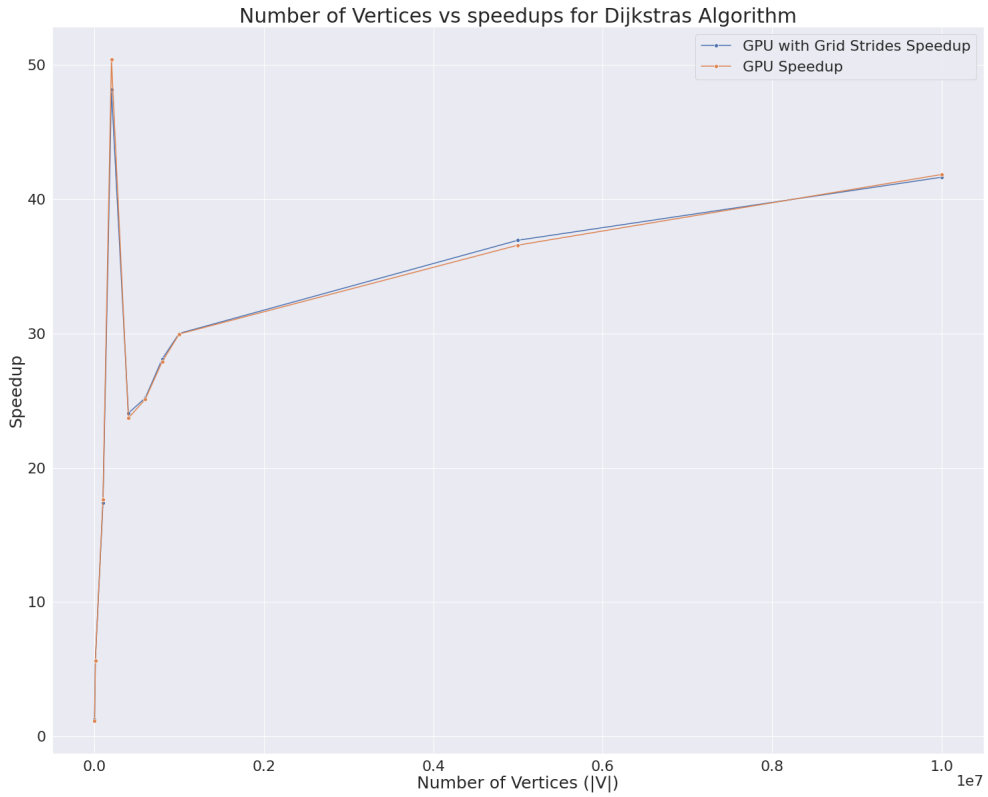

Fig. 4. Speedups achieved by GPU and GPU with grid strides over different numbers of vertices by Dijkstra's algorithm.