PARALLEL GPU BASED SHORTEST PATH IN GRAPH ALGORITHMS

A PREPRINT

Babuji Periasubbramaniam
University of Texas, Austin
`bp26282`

Ryan Denlinger
University of Texas, Austin
`rd28444`

November 16, 2023

ABSTRACT

Finding the shortest paths from a single source to all other vertices is a fundamental method used in a variety of higher-level graph algorithms. In this project, will find the single source shortest path for a given graph with GPU by implementing LLP(Lattice Linear Predicate) algorithm and compare the same with the traditional GPU Bellman-Ford implementation.

***K*eywords** SSSP · LLP · Graph · BellmanFord · GPU · CUDA · Parallel · More

1 Introduction

SSSP stands for Single-Source Shortest Path, which is a fundamental algorithm in graph theory. The primary purpose of the SSSP algorithm is to find the shortest paths from a single source vertex to all other vertices in a weighted graph. The "shortest path" in this context refers to the path with the minimum sum of weights. We assume that all weights are non-negative for the present report. If we view the weights as representing a cost (such as money or time) associated with one segment (edge) of the overall path, then the SSSP problem can be thought of as an optimization problem whereby we seek the minimal cost (and the associated path). It is also possible to think of this problem geometrically, in the case where a finite discretization is known for a smooth two-dimensional manifold (such as the Earth's surface, accounting for the geometry of hills, valleys, etc.). Such a discretization typically arises as the disjoint union of (open) triangles, (open) lines, and points, known as a cell complex, and the edge weights are then just determined by the local metric (assuming the discretization is sufficiently fine). Then the shortest paths emanating from a source vertex simply represent approximations of the geodesics emanating from the corresponding point on the manifold (although this interpretation only applies to geodesics that are not too far from their corresponding discretized paths: namely, the local angle between the true geodesic and its approximating path must not be too large at any point).

2 Use Cases and Applications

Routing Algorithms:

SSSP is widely used in network routing algorithms, where the goal is to find the most efficient paths for data transmission in computer networks, the internet, and telecommunications.

Geographic Information Systems (GIS):

SSSP is used in GIS applications to find the shortest path between locations on maps. This is valuable in route planning, logistics, and navigation systems.

Transportation and Logistics:

SSSP is applied in transportation and logistics for optimizing routes and minimizing travel distances, which is essential for efficient delivery systems.

Robotics and Path Planning:

In robotics, SSSP is used for path planning, allowing robots to navigate through a space while minimizing travel distance or time.

Social Network Analysis:

SSSP can be applied to analyze social networks to find the shortest paths between individuals, understanding relationships and influence in a network.

Graph Algorithms:

SSSP is a building block for many other graph algorithms. It is used in combination with other algorithms for solving complex graph problems.

Traffic Engineering:

SSSP is used in traffic engineering to optimize traffic flow and reduce congestion by finding the most efficient routes for vehicles.

Resource Allocation:

SSSP can be used in resource allocation scenarios, such as determining the most efficient allocation of resources in a network or system.

3 CPU vs GPU

Implementing SSSP on both CPU (Central Processing Unit) and GPU (Graphics Processing Unit) can offer performance advantages based on the characteristics of the algorithm and the hardware.

CPUs are general-purpose processors capable of handling a wide range of tasks; they are well-suited for algorithms with irregular memory access patterns, as they have sophisticated cacheing mechanisms. SSSP is often a memory-bound algorithm, and CPU implementations might face limitations due to memory bandwidth. GPUs are highly parallel processors, capable of handling large amounts of data simultaneously; they are well-suited for algorithms with regular, parallelizable computations, such as those found in graph algorithms. Optimal GPU implementation requires efficient handling of memory access patterns, which may in turn necessitate careful algorithm design to make effective use of GPU memory. GPUs excel at parallel processing (since they may contain thousands of cores), making them suitable for algorithms like SSSP that involve many parallel computations. Memory-bound algorithms may benefit more from CPU implementations if the CPU has better memory access characteristics, or if the algorithmic problem is inherently sequential. Additionally, GPUs may experience overhead due to data transfers between CPU and GPU memory. Efficient management of data transfers is crucial for GPU performance. The specific characteristics of the SSSP algorithm can influence the choice between CPU and GPU implementations. In some cases, a hybrid approach that leverages both CPU and GPU can provide optimal performance. This approach involves dividing the workload between the CPU and GPU, allowing each processor to handle tasks that suit its strengths.

The choice between CPU and GPU implementations for SSSP depends on factors such as the specific algorithm, graph size, memory access patterns, and the efficiency of parallelization. In many cases, GPUs can offer substantial performance gains for graph algorithms due to their parallel processing capabilities. However, careful consideration of algorithm design and memory access patterns is crucial for achieving optimal performance on both CPU and GPU architectures.

4 LLP algorithm for SSSP

The most classical, and standard, algorithm for SSSP is Dijkstra's algorithm, which assumes all weights to be non-negative. (The case of possibly negative weights, assuming no negative cycles, can be handled with a dynamic programming algorithm, known as Bellman-Ford.) Dijkstra's algorithm works by "exploring" the neighboring edges of vertices that have already been explored (traversing away from the source node), until all paths have been fully explored. Once we can say for certain that a certain node's estimate cannot be further improved (by exploring all its incoming edges, and hence all incoming paths to that node, by induction), that node is said to be *fixed*. Since every edge must be considered, for $|V|$ vertices and $|E|$ edges, the time complexity becomes $O\left(|E| + |V| \log |V|\right)$, where a Fibonacci heap is used to track the minimum unfixed node (removal from the heap explains the logarithm). The term $|V| \log |V|$ is quite satisfactory (since it is just slightly worse than linear in the size of the graph). Of course, since the description of a graph must include both its vertices and its edges, the term $|E|$ is actually *linear* in the "size" (in bits) of the graph, so superficially this term seems better than the $|V| \log |V|$ term. The opposite is true, however, because in a directed graph with no repeated edges and no self-loops, there can be $|V|^2 - |V|$ directed edges. Hence, even in a moderately dense graph, the term $|E|$ is potentially much more costly (and the exploration is also more costly than simply "storing" the graph to memory, so the linear comparison to the "size" of the graph is not really relevant).

To better understand the problem, imagine that a company needs to find the shortest paths from a warehouse in Tulsa, Oklahoma, to each location (taken separately from all the others) in the United States. Initially, the company may only be interested in interstate highways, of which there are relatively few. However, as time passes, they will undoubtedly wish to use more detailed maps, which may identify certain shortcuts near the destination. This means including many more edges in the graph, and therefore, more computation time. However, a shortcut in Los Angeles is irrelevant to destinations in New York City, and similarly, a shortcut in New York City is irrelevant to a destination in Los Angeles. This implies that there is an opportunity to parallelize.

In the Lattice Linear Program (LLP) for SSSP (Page 70, Chapter 8, of the textbook), we update all vertex distance estimates (from the source) in parallel. In this algorithm, the *size* of the updates are optimized by solving a minimization problem (which can be done in parallel using reduce). For this algorithm, the predicate *parent* (j, i, G) is defined by

$$\textit{parent}\,(j,i,G) = (i \in \textit{pre}\,(j)) \wedge (G\,[j] \geq G\,[i] + w\,[i,j])$$

where $w\,[i,j]$ is the (directed) weight from vertex i into vertex j, G is the lattice vector (defined over the vertex set), and *pre* (j) is the set of vertices for which there exists a directed edge from that vertex into j. At any time, $G\,[j]$ represents our current estimate of the best path from the source node to vertex j, and G only increases element-wise as the algorithm progresses. We track a set of fixed vertices (for which the correct estimate is now known) by the predicate *fixed* (j, G),

$$\textit{fixed}\,(j,G) = (j = 0) \vee (\exists i \;:\; \textit{parent}\,(j,i,G) \wedge \textit{fixed}\,(i,G))$$

which may be viewed as a recursive definition with base case $j = 0$, representing the source vertex. Forbidden local states are exactly those for which the predicate *fixed* (j, G) fails to hold.

In order to guarantee that the LLP progresses forward at a sufficient rate, regardless of the detailed structure of the weights, the following minimization problem is solved:

$$\beta\,[G] = \min\left\{G\,[i] + w\,[i,j] \;:\; (i,j) \in E'\right\}$$

where E' is the set of edges for which the source i is fixed and the target j is not fixed. Assuming that there are edges to explore (i.e. all vertices are reachable), this set is guaranteed to be non-empty as long as there exists a non-fixed vertex. We can check for non-fixed vertices in constant parallel time, just by having each processor set a common flag (initially unset) if it is non-fixed. If there are at most m incoming edges for each vertex, we can minimize $G\,[i] + w\,[i,j]$ across edges in E' leading into j by handling each j on a different processor, in $O\,(m)$ parallel time. Finally, the minimization over all target nodes j can be performed in $O\,(\log |V|)$ time using min-reduce in parallel. Actually, the implementation of min-reduce optimally in CUDA is quite complicated, see slides by Harris [6]. We took a vary naive approach to parallel reduce, using $O\,(\log |V|)$ global synchronization operations (barriers). In the approach outlined by Harris, synchronization occurs at the block level, which is potentially a massive speedup when the block size is chosen carefully, because blocks do not block other blocks. Our lack of optimization of reduce implies that we cannot possibly leverage the full power of parallelism in this LLP. While we have not checked that the reduction is definitely a bottleneck, it is highly suspected. We also believe that our LLP implementation employs more copying between GPU memory and CPU memory (and vice versa) than is strictly required. Our LLP implementation assumes that $|V|$ is a power of 2, so our performance comparisons are subject to this assumption.

Vertices which fulfill the optimization predicate for β are immediately advanced to $\beta\,[G]$, which is a constant parallel time operation, once $\beta\,[G]$ is known. For remaining vertices that are not already fixed, we can advance further, according to the following formula:

$$\alpha\,(G,j) = \max\left\{\beta\,(G)\,, \min\left\{G\,[i] + w\,[i,j] \;:\; i \in \textit{pre}\,(j)\right\}\right\}$$

The overall algorithm is as follows, as long as the vector G is initially defined to be the zero vector and the source vertex is the 0th vertex:

```
while there exists a non-fixed vertex (note: check the non-fixed condition):
    find Eprime, the set of edges with fixed source and non-fixed target
    if Eprime is empty, return "non-fixed nodes not reachable" (error state)
    Let (i*,j*) minimize G[i]+w[i,j] across Eprime, call the min beta(G)
    G[j*] := beta(G)
    for all j != j* which is forbidden (ie., not fixed), in parallel do:
        G[j] := alpha(G,j) as defined in the text
endwhile
return G
```

The work complexity is the same as Dijkstra; the parallel time complexity (assuming $|V|$ cores) is no more than $O(m + \log |S|)$ for each iteration of the outer loop, where $|S| = |E| + |V|$ and there are at most m incoming edges for each vertex.

5 Bellman Ford algorithm for SSSP

Traversing large graphs to compute different information has various use cases in the real-world like social media networks, communication networks, VLSI design and biological network analysis. Bellman-Ford, Dijkstra's and Delta Stepping are widely used Single Source Shortest Path Algorithm (SSSP) algorithms. Dijkstra's algorithm provides a work efficient implementation, whereas Bellman-Ford provides scope for easy parallel implementation. Parallelizing Bellman-Ford on a GPU involves distributing the workload among threads, each handling a subset of vertices. CUDA can be used for efficient GPU parallelization.

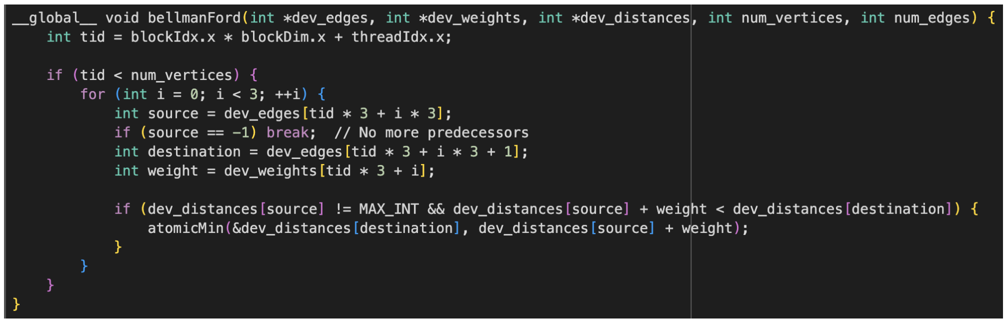
```
function BellmanFord(graph, source):
    initialize(graph, source)
    for i from 1 to |V| - 1:
        for each edge (u, v) in graph.edges:
            relax(u, v, graph)
    return distance

function relax(u, v, graph):
    if distance[u] + weight(u, v) < distance[v]:
        distance[v] = distance[u] + weight(u, v)
        predecessor[v] = u
```

This implementation presents Bellman-Ford SSSP algorithm using Compute Unified Device Architecture (CUDA) on General-purpose computing on graphics processing units (GPGPU) and single-threaded CPU. Later will compare the performance of this algorithm between CPU and GPU.

The Bellman-Ford algorithm is a single-source shortest path algorithm that finds the shortest paths from a source vertex to all other vertices in a weighted graph. The GPU implementation of the Bellman-Ford algorithm involves parallelizing the computation using CUDA, a parallel computing platform and application programming interface model created by Nvidia.

Here's an explanation of the GPU implementation of the Bellman-Ford algorithm:

CUDA Kernels: The Bellman-Ford algorithm is inherently iterative, and each iteration updates the distance estimates from the source vertex to all other vertices. In CUDA, we represent each vertex as a thread.



```
__global__ void bellmanFord(int *dev_edges, int *dev_weights, int *dev_distances, int num_vertices, int num_edges) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < num_vertices) {
        for (int i = 0; i < 3; ++i) {
            int source = dev_edges[tid * 3 + i * 3];
            if (source == -1) break;  // No more predecessors
            int destination = dev_edges[tid * 3 + i * 3 + 1];
            int weight = dev_weights[tid * 3 + i];

            if (dev_distances[source] != MAX_INT && dev_distances[source] + weight < dev_distances[destination]) {
                atomicMin(&dev_distances[destination], dev_distances[source] + weight);
            }
        }
    }
}
```

Figure 1: Bellman Ford CUDA Kernel is responsible for updating the distances in parallel based on the relaxation step of the Bellman-Ford algorithm.

`__global__` This keyword indicates that the above function is a GPU kernel that can be called from the host (CPU) and will run on the device (GPU).

int *dev_edges, int *dev_weights, int *dev_distances: These are pointers to the device memory, representing the edges, weights, and distances arrays on the GPU.

int num_vertices, int num_edges: These parameters specify the number of vertices and edges in the graph.

int tid = blockIdx.x * blockDim.x + threadIdx.x;: This line calculates a unique thread ID for each thread in the GPU grid. The thread ID is used to determine which vertex the thread is responsible for.

if (tid < num_vertices) ... : This condition ensures that only threads corresponding to valid vertices participate in the computation. Threads beyond the number of vertices do nothing.

The subsequent for loop iterates over all edges in the graph:

int source = dev_edges[i * 3];: Retrieves the source vertex of the current edge.

int destination = dev_edges[i * 3 + 1];: Retrieves the destination vertex of the current edge.

int weight = dev_weights[i];: Retrieves the weight of the current edge.

The following if statement checks if relaxing the edge (updating the distance) would result in a shorter path.

dev_distances[source] != MAX_INT: Ensures that the source vertex has a valid distance (not infinity) before attempting to relax the edge.

dev_distances[source] + weight < dev_distances[destination]: Checks if updating the distance along the current edge results in a shorter path to the destination vertex.

atomicMin(&dev_distances[destination], dev_distances[source] + weight);: If the condition is true, it atomically updates the distance of the destination vertex to the new, shorter distance. atomicMin is used to avoid race conditions when multiple threads try to update the same destination vertex simultaneously.

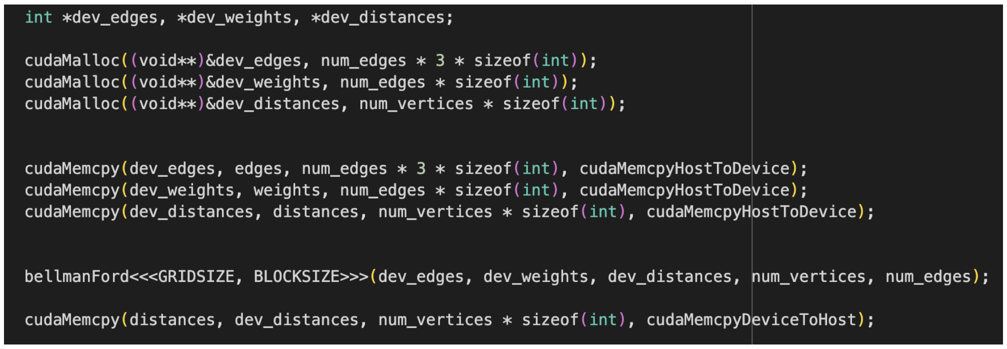

```
int *dev_edges, *dev_weights, *dev_distances;

cudaMalloc((void**)&dev_edges, num_edges * 3 * sizeof(int));
cudaMalloc((void**)&dev_weights, num_edges * sizeof(int));
cudaMalloc((void**)&dev_distances, num_vertices * sizeof(int));


cudaMemcpy(dev_edges, edges, num_edges * 3 * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_weights, weights, num_edges * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_distances, distances, num_vertices * sizeof(int), cudaMemcpyHostToDevice);


bellmanFord<<<GRIDSIZE, BLOCKSIZE>>>(dev_edges, dev_weights, dev_distances, num_vertices, num_edges);

cudaMemcpy(distances, dev_distances, num_vertices * sizeof(int), cudaMemcpyDeviceToHost);
```

Figure 2: These arrays are allocated and copied to the device memory using CUDA memory management functions (cudaMalloc, cudaMemcpy). The bellmanFord kernel is launched with a grid and block configuration to execute the algorithm in parallel for all vertices.

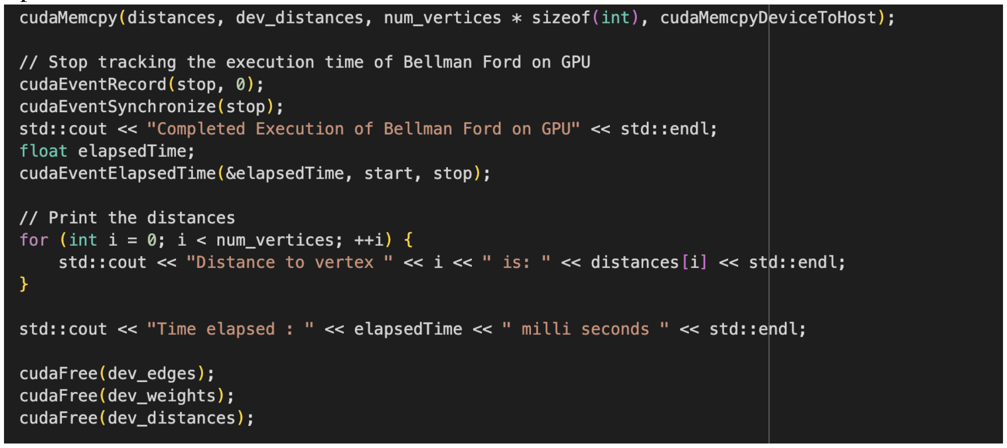

```
cudaMemcpy(distances, dev_distances, num_vertices * sizeof(int), cudaMemcpyDeviceToHost);

// Stop tracking the execution time of Bellman Ford on GPU
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
std::cout << "Completed Execution of Bellman Ford on GPU" << std::endl;
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);

// Print the distances
for (int i = 0; i < num_vertices; ++i) {
    std::cout << "Distance to vertex " << i << " is: " << distances[i] << std::endl;
}

std::cout << "Time elapsed : " << elapsedTime << " milli seconds " << std::endl;

cudaFree(dev_edges);
cudaFree(dev_weights);
cudaFree(dev_distances);
```

Figure 3: After the kernel execution, the results (updated distances) are copied back to the host from the device memory. This allows the host (CPU) to access the final results.

Performance Measurement – Bellman-Ford

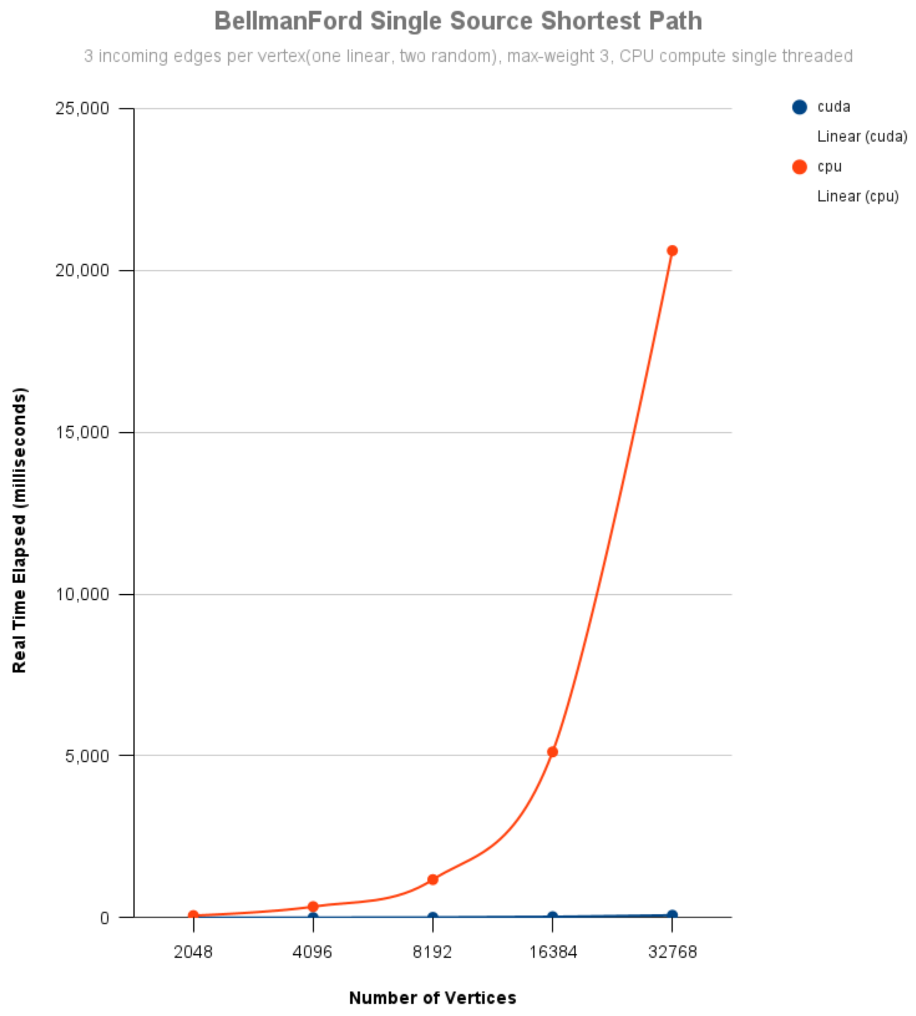


Figure 4: Comparison of GPU with 2560 CUDA cores, vs CPU with a single thread on a single core. Both computes use the same BellmanFord algorithm. Performance on CUDA (with 2560 CUDA cores) was compared to the same bellmanford implementation running on a single thread on a single core (2GHz Intel(R) Xeon(R)). In the CUDA case, the number of CUDA cores is smaller than the number of vertices apart from the smallest problem, so the scaling with problem size cannot be better than linear.

Same choices of random seed for graphs were measured in both GPU and CPU cases. The only effective difference between the two data series is whether the compute was run on GPU or single-threaded CPU. Observed huge speed difference with GPU when compared max case of 32768 vertices. This GPU implementation harnesses the parallel processing capabilities of the GPU to accelerate the Bellman-Ford algorithm, especially beneficial for large graphs with many vertices.

The time complexity of the Bellman-Ford algorithm is O(V * E), where V is the number of vertices and E is the number of edges. In our CUDA implementation, the outer loop in the bellmanFord kernel runs for each vertex (V), and the inner loop runs for each edge (E). Therefore, the time complexity of the CUDA implementation remains O(V * E).

It's important to note that the Bellman-Ford algorithm has a time complexity that depends on both the number of vertices and edges in the graph. In the worst case, the algorithm must relax all edges for each vertex, resulting in O(V * E) time complexity.

In our CUDA implementation, the work is divided among the CUDA threads, and the algorithm is parallelized to take advantage of GPU parallel processing capabilities. The specific execution time will depend on factors such as the number of CUDA threads, the GPU architecture, and the characteristics of the input graph.

Keep in mind that while parallelization can significantly speed up certain algorithms, the fundamental time complexity remains a key factor in understanding the algorithm's efficiency as the input size grows. Our CUDA Implementation of BellmanFord is available at the following URL: `https://colab.research.google.com/drive/1BPOP7NVWG6QjYpjTmt8sVJogtvKD4oyd?usp=sharing`.

Performance Measurement – LLP

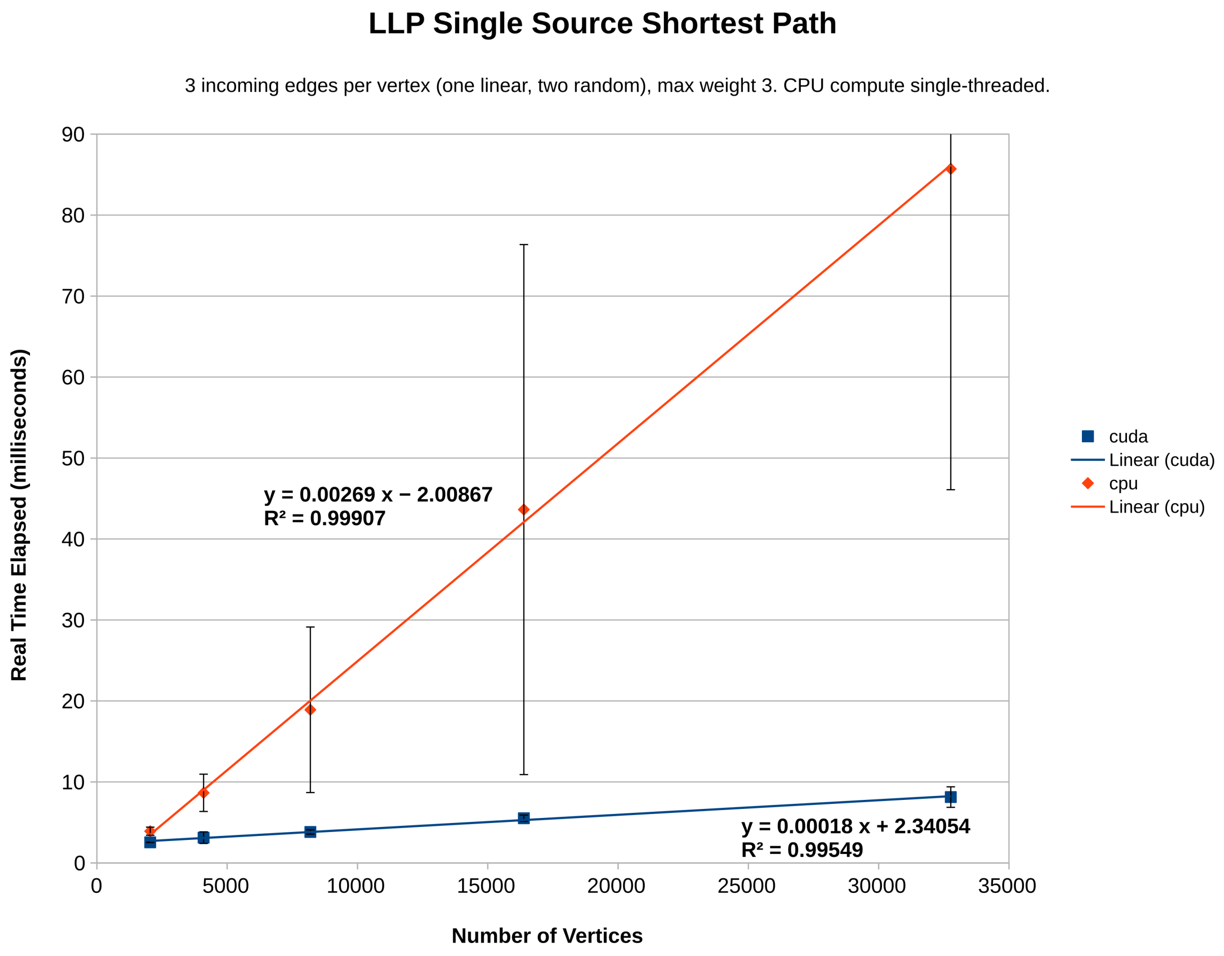


Figure 5: Comparison of GPU with 2560 CUDA cores, vs CPU with a single thread on a single core. Both computes use the same LLP algorithm (we are not using a min-heap in the CPU case, so that time is not fully optimized – but the GPU version is also not fully optimized for the reasons discussed in the text). A linear non-self-intersecting directed path through all vertices (emanating from the source) is constructed to ensure reachability, then two more randomly-selected edges are added per vertex; all weights are random positive integers not greater than 3. Each data point is the average of 3 runs using different random seeds, and error bars represent a 99% confidence interval assuming Gaussian statistics. In the CUDA case, the error bars are smaller than the data point mark except for the case of 32768 vertices.

The LLP performance on CUDA (with 2560 CUDA cores) was compared to the same LLP implementation running on a single thread on a single core (2GHz Intel(R) Xeon(R)). In the CUDA case, the number of CUDA cores is smaller than the number of vertices apart from the smallest problem, so the scaling with problem size cannot be better than linear. A min heap **was not used** in the CPU case; rather, the minimization was solved with min reduce, just as in the CUDA case, except run sequentially; moreover, the same choices of random seed were measured in both

GPU and CPU cases. Therefore, the only effective difference between the two data series is whether the compute was run on GPU or single-threaded CPU. Surprisingly, despite a heavy synchronization cost (e.g. due to highly sub-optimal min-reduce, compared to the optimized code of Harris [6]), the GPU observes a massive speedup, up to a factor of 10 on average in the case of 32768 vertices. The number of incoming edges was capped at a constant of 3 for each vertex (one deterministic to ensure reachability, and two more chosen at random, for each vertex), so the graphs are inherently quite sparse; it is expected that the runtime could be much larger for a dense graph. Our CUDA implementation of LLP SSSP is available at the following URL: `https://colab.research.google.com/drive/1sOLDXcmIbvbwBEpeCxTuey4nHD_h5U1N?usp=sharing`

6 GPU Environment Details

Device, Driver, and Compiler Used via Google Colab:

Device #0: Tesla T4

Compute Capability: 7.5

Total Memory: 14 GB

nvcc: NVIDIA (R) Cuda compiler driver

Copyright (c) 2005-2022 NVIDIA Corporation

Built on Wed_Sep_21_10:33:58_PDT_2022

Cuda compilation tools, release 11.8, V11.8.89

Build cuda_11.8.r11.8/compiler.31833905_0

7 CPU Environment Details

Device, Driver, and Compiler Used via Google Colab:

Intel(R) Xeon(R) 2.00GHz

2 cores

13 GB available RAM

8 Performance Comparison

Comparing the performance of LLP with Bellman-Ford algorithm involves running both algorithms on comparable input data of comparable complexity, and measuring various metrics such as execution time, memory usage, and correctness. Remember that the choice of the "best" algorithm depends on various factors, including the specific characteristics of the input data, the problem constraints, and the available resources. Consider the trade-offs between time complexity, space complexity, and practical considerations for our specific use case.

9 Conclusion

Implementing algorithms in parallel, especially on GPUs, can lead to significant speedups for certain types of problems. LLP(Lattice Linear Predicate) and Bellman-Ford algorithm are valuable tools for finding shortest paths in graphs. Parallelizing the algorithm on GPUs can significantly improve its performance, especially for large and dense graphs. Efficient parallelization requires careful consideration of GPU architecture, thread configuration, and graph characteristics. Benchmarking and profiling are essential to evaluate the benefits of parallelization and identify scenarios where it outperforms sequential implementations.

In conclusion, the decision to parallelize the SSSP algorithm depends on the specific requirements of the problem, the characteristics of the input graph, and the available computational resources. It's essential to balance the benefits of parallelization with the associated complexities and overhead. The main advantage of Bellman-Ford is that it works for graphs with negative weight edges. Moreover, it is highly parallelizable for running on GPU with a relatively simple implementation. However, the sequential time complexity is O(V * E), making it less efficient for dense graphs.

The LLP Shortest Path, by contrast, leverages the efficient properties of Dijkstra, but can be much faster than Dijkstra on GPU since it can run in parallel. Like Dijkstra, the present implementation only works for positive weights. In order to leverage the full power of the LLP SSSP algorithm, an efficient minimization routine is required (such as min-reduce), which can be challenging on CUDA due to synchronization and memory layout issues.

The choice between LLP and BellmanFord algorithm on GPU depends on the specific characteristics of the problem and the input graph. GPU parallelization provides an avenue to harness the computational power of GPUs, but the efficiency of parallelization depends on the algorithm and the nature of the graph. Benchmarking and profiling are essential steps in determining the most suitable algorithm for a given use case and graph type.

References

[1] New Approach of Bellman Ford Algorithm on GPU using Compute Unified Design Architecture (CUDA) - Agarwal, Pankhari, Dutta, Maitreyee

[2] Shortest Paths Algorithms: Theory And Experimental Evaluation. Boris Cherkassky, Andrew V. Goldberg and Tomasz Radzik

[3] Fast classification of handwritten on-line arabic characters. Accelerating large graph algorithms on the GPU using CUDA - Pawan Harish and P. J. Narayanan

[4] https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/

[5] https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/

[6] Mark Harris, *Optimizing Parallel Reduction in CUDA*, https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf