Link to Github Repo: https://github.com/Erice1221/GPU-Accelerated-SSSP-Algorithms

1 Summary

We implemented and evaluated different approaches for SSSP implementations on GPUs using the Bellman Ford algorithms and Delta-Stepping algorithms. For the Bellman Ford implementations, we created a sequential baseline version, an edge parallel CUDA version, and a frontier parallel CUDA version. For the Delta-Stepping implementation, we created a sequential Dijkstra baseline and a parallel CUDA implementation. We then evaluated the performance of all implementations on a CPU and a NVIDIA RTX 2080 using synthetic and real graphs that vary in size, density, and structure.

2 Background

Bellman Ford is an algorithm that solves the single source shortest path (SSSP) problem on a directed, weighted graph (i.e $G = (V, E)$) by repeatedly relaxing its edges. Essentially, for each edge (u, v, w), the algorithm updates the distance to vertices according to:

$$dist[v] = min(dist[v], dist[u] + w).$$

The algorithm represents the graph as a set of edges and keeps track of the distances (all initially infinity except for the source). The costly part of the algorithm is the nested loop that has to scan all edges and do up to $|V| - 1$ iterations, which gives a worst case time complexity of $O(|V| \cdot |E|)$.

In each iteration, all edge relaxations are independent of each other, except for potentially having competing updates to the same vertex. This clearly has the potential for significant parallelism, where all edges can be processed in parallel if distance updates can be deemed atomic. In addition, locality is good when edges are stored in a contiguous pattern, meaning scanning them is great for SIMD execution on a GPU.

Synchronization is another factor since there can be dependencies between iterations where later relaxations might need updated distances and at each vertex we can have multiple edges updating the same $dist[v]$.

Delta-Stepping is another SSSP algorithm that is more complex but could be more efficient on graphs with non-negative weights. It essentially works as an extension of Dijkstra's algorithm by grouping vertices into buckets based on their current distance estimates. Each bucket represents a range of distances defined by a parameter Δ. The algorithm processes vertices in the order of their buckets, relaxing edges and updating distances accordingly. This would allow for parallel processing of vertices within the same bucket, as they can be relaxed independently. The choice of Δ is crucial, as it affects the number of buckets and the amount of work done in each iteration. A smaller Δ leads to more buckets and potentially finer-grained parallelism, while a larger Δ reduces the number of buckets but may increase the work per bucket.

3 Approach

3.1 Bellman Ford

The sequential baseline implementation is written in C++ and is a more traditional textbook implementation. It does up to $n-1$ iterations over the edge list with a flag to exit early if an iteration doesn't change anything. In addition, all input graphs are read in the format (txt files):

$$\begin{array}{lll} n & m & source \\ u & v & w \\ \ldots & & \end{array}$$

Where n is the number of vertices, m is the number of edges, $source$ is the source vertex.

For the GPU implementations, CUDA/C++ and a NVIDIA RTX 2080 were used. The first algorithm that was implemented for Bellman Ford was the edge parallel implementation. Here, the the general structure mimics the idea of the sequential version but it parallelizes the inner edge loop. Essentially, the graph is copied to the device memory as an array, and each thread processes multiple consecutive edges, where the number of edges per thread is dynamically tuned based on the graph size. Then, each thread computes a potential distance and uses $atomicMin$ on the global distance array. Finally, a per block shared flag is used to reduce contention where one thread per block writes to the global flag if any updates happen in the block. This allows for after each kernel launch, the host to be able to read a global flag and exit early if we have no changes. Initially, an attempt was made to give each thread some fixed number of edges and then compute the indices manually, but this caused bad work distribution and more non contiguous memory accesses as the number of edges changed. The dynamic version improved memory coalescing and reduced kernel launch overhead for smaller graphs.

For the Bellman Ford frontier parallel implementation it essentially first converts the edge list into a CSR style structure by sorting edges by source vertex and computing row pointers (prefix sum), using 2 frontier arrays on the device side. For each iteration, one thread is launched per frontier edge to get better load balancing when vertices have different out degrees. A binary search is then used within each thread to get the frontier vertex for the respective edge index. In each iteration we also have to compute a prefix sum over vertex degrees to find edge offsets for the binary search. The kernels add to the current frontier by relaxing outgoing edges and then building the next frontier using the vertices who saw a reduction in distance. Due to this implementation, there can be a reduction in the work required only when a small subset of vertices are active, but there is also the additional overhead from using the frontiers and extra kernel launches. Important to note that the prefix sum is computed on the CPU, which adds some host device synchronization overhead.

Additionally, a Python script was used to generate different kinds of input graphs that all vary in density, size, structure, and style. Also, a checker script was made to run the graphs on the implementations, verify that the results match between the different implementations, and compute performance metrics.

3.2 Delta Stepping

For the Delta-Stepping implementation, the graphs were stored natively into the CSR style structure to allow for efficient accesses. The sequential Dijkstra would be using C++ STL priority queues to get the next vertices to process. For parallel Delta-Stepping, the implementation would keep the graph and all "buckets" resident on the GPU and driving the algorithm with a single persistent kernel. The graph is stored in CSR form in device memory, and the notion of buckets from the general description is captured by a small set of device queues: a current queue that represents the bucket being processed, and two next queues that hold relaxations of light and heavy edges (often called "near" and "far" work). At any time, the current queue plays the role of the smallest non-empty bucket: all vertices whose tentative distance places them in that bucket are stored there, and their outgoing edges are relaxed in parallel. Light edges (with weight up to the chosen Δ) cause their targets to be placed into the "near" queue (meaning they stay within the same or nearby bucket), while heavy edges send their targets into the "far" queue (meaning they belong to later buckets). Control variables and counters in global memory track which queue is active, how many vertices are in each queue, and which "generation" of deduplication is current, so that bucket transitions can be performed entirely on the device without host intervention.

The mapping to blocks, threads, and warps is designed so that the parallel relaxation inside a bucket matches the conceptual delta-stepping step "process all vertices in the current bucket and relax their edges." A grid of blocks is launched once, and each block is divided into warps of 32 threads. Within each iteration of the persistent kernel's main loop, the vertices in the current queue are distributed across all threads using a global strided loop, so that each thread is assigned one or more frontier vertices to process. For a given vertex, its outgoing edges are scanned from CSR arrays in global memory, and each edge relaxation corresponds to the delta-stepping operation "try to improve the tentative distance of the neighbor and possibly re-bucket it." Warps are used as the basic SIMD/SIMT unit: for low-degree vertices, each thread processes its own neighbor list independently, while for high-degree vertices, an entire warp cooperatively processes the adjacency list of a single frontier vertex so that many light or heavy edges are relaxed in parallel and memory accesses are coalesced. Queue insertions for light and heavy relaxations are also handled at warp granularity: a warp collectively reserves a contiguous block of positions in the appropriate queue via a single atomic operation, and each active lane writes its neighbor index into that block. Across iterations of the while-loop, device-wide barriers synchronize all blocks so that the promotion of "near" or "far" queues to become the next current bucket mirrors the abstract algorithm's progression from one bucket to the next, with blocks, threads, and warps repeatedly re-used to process new buckets until all queues are empty and the final distances have been established.

Conceptually, delta-stepping could be implemented with multiple kernel launches from the host, where each launch processes one bucket and then returns control to the host to set up the next bucket. However, this would incur significant overhead from repeated host-device synchronization and kernel launches, as well as redundant data transfers for queue management. Instead, a persistent kernel approach is implemented in the parallel delta-stepping version, where a single kernel is launched that remains active for the entire duration of the algorithm. This kernel contains a loop that repeatedly processes the current bucket, relaxes edges, and manages queue transitions internally on the device. By keeping all data resident on the GPU and avoiding host intervention between buckets, this approach minimizes overhead and maximizes parallel efficiency, allowing the algorithm to fully leverage the GPU's capabilities for high-throughput computation.

The queue and warp-level optimizations are designed so that work created by relaxations is funneled into a few global queues with as little contention and divergence as possible. Instead of having every thread independently perform an *atomicAdd* to reserve a single slot when it needs to enqueue a vertex, queue insertions are aggregated at the warp level. Within a warp, all lanes that want to push a vertex "vote" together, the total number of participating lanes is computed, and a single *atomicAdd* is issued by a designated leader to reserve a contiguous block of positions in the target queue. The base index returned by that one atomic operation is then broadcast to the warp, and each lane writes its vertex at a unique offset within that block. In this way, many enqueue operations are bundled into one atomic update, reducing contention on the queue counters and keeping the queue memory layout compact and friendly to coalesced writes. Duplicate insertions of the same vertex are further filtered by simple "membership maps," which are not physically cleared each iteration but instead are tagged with a generation number so that stale entries are ignored; this avoids repeatedly zeroing large arrays and keeps queue traffic focused on genuinely new work.

At the warp level, adjacency processing is organized to keep the SIMD hardware busy and to mitigate the irregularity of real-world graphs. Vertices with small degree are processed independently by the threads that own them, which keeps control flow simple when there is not much parallel work per vertex. When a vertex is identified as "heavy" (having a large adjacency list), its neighbors are instead processed cooperatively by an entire warp. One lane in the warp supplies the start and end indices of the adjacency segment and its current distance value, and these are broadcast to all lanes. Each lane then walks over a distinct subset of the edges from that vertex in a strided fashion, so that many relaxations for a single high-degree vertex are carried out in parallel with coalesced loads from the edge array. Combined with the warp-aggregated queue appends, this cooperative heavy-vertex mode ensures that both the creation of new work (enqueueing) and the consumption of existing work (edge relaxation) are carried out in a warp-aware manner, improving throughput and reducing the overheads that would otherwise arise from per-thread atomics and highly divergent control flow.

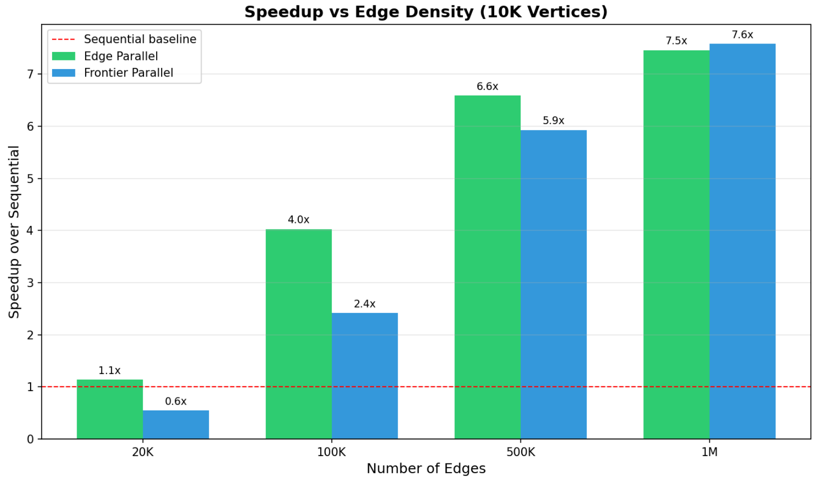
4 Results for Bellman Ford

Performance for the Bellman Ford implementations was measured using wall clock time for the computation runtime of each one. Its important to note that since performance was the main criteria for evaluating the implementations only graphs with positive weight edges were used, allowing the implementations not to check for negative cycles. The main metric is speedup ($\frac{T_{CPU}}{T_{GPU}}$). Then for the input graphs 3 different styles of synthetic graphs were generated and used:

- **Density scaling** with 10K vertices and 20K, 100K, 500K, and 1M edges.
- **Size scaling** with 1K, 5K, 10K, 50K, 100K, and 500K vertices with 10 edges per vertex.
- **Structure** having 50K vertex graphs that have random, clustered, grid, chain, road like (sparse), and star structures.

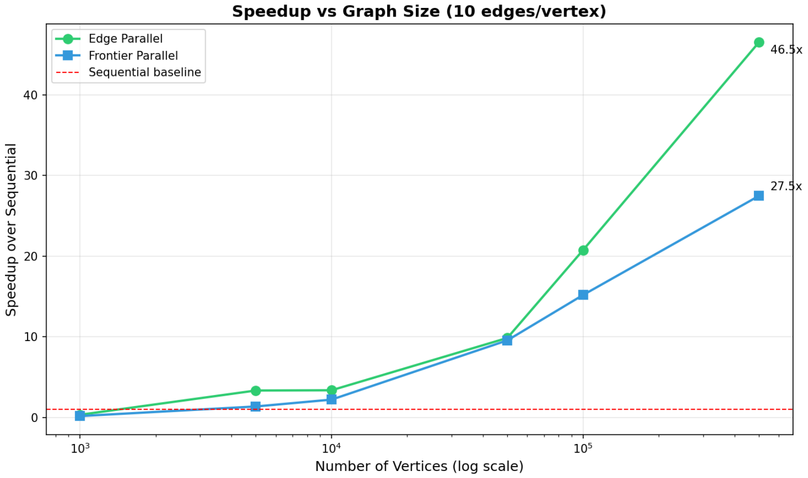
For each input graph, the checker script confirmed that all the implementations gave identical distances/results.

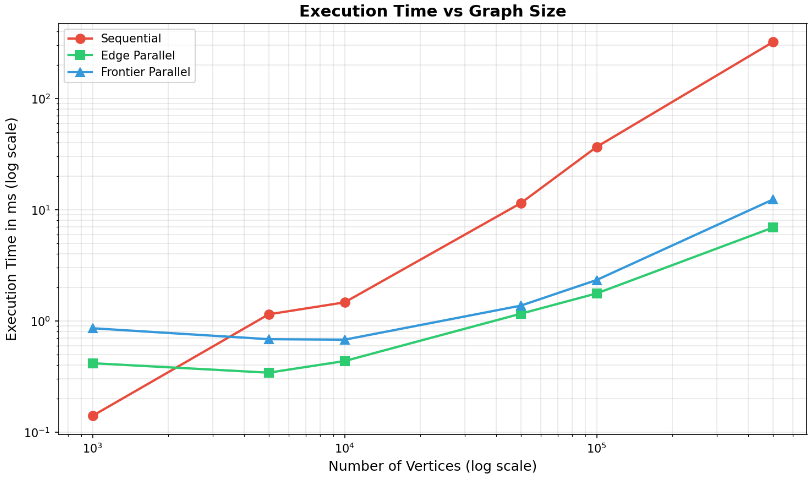
4.1 Effect of Edge density (10K vertices)



With the number of vertices fixed at 10K, and with an edge count of 20K, meaning 2 edges per vertex and increasing the edges to 1M, meaning 100 edges per vertex significantly improved GPU speedup. The edge parallel version goes from 1.1x speedup on the sparse graph to 4x, 6.6x, and 7.5x as the density of the graph grows. For the frontier implementation speedup went from 0.6x to 2.4x, 5.9x, and 7.6x. Implementation wise, having sparse graphs causes little parallel work per iteration while still having the kernel launch and synchronization overhead. Additionally, frontier management overhead plays a significant factor when the frontier is nearly the whole graph. In general, as the density increases each kernel has to process a lot more edges (creating more work for threads), and allowing the parallel implementations to outperform the sequential version.

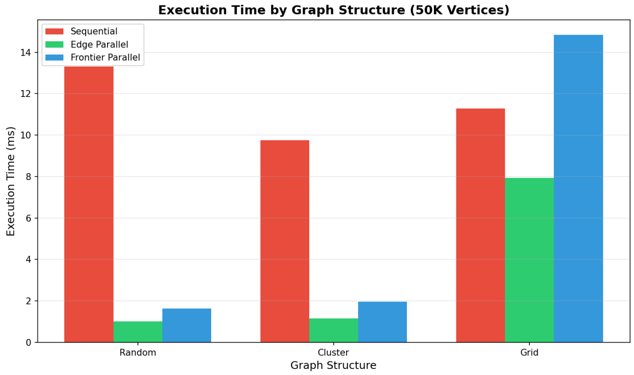
4.2 Effect of Graph size (10 edges per vertex)

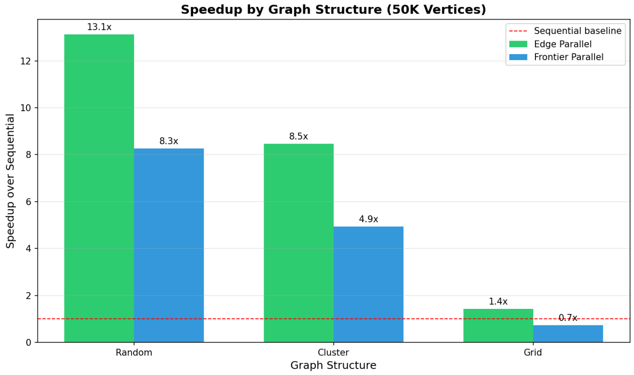




With a fixed 10 edges per vertex, scaling from 1K to 500K vertices shows the expected realistic outcome where small graphs dont perform well for parallel performance but large graphs do. At 1K vertices, the edge speedup was 0.33x and the frontier speedup was 0.17x (much slower than the sequential). This makes sense though because the work per iteration is so small, there is no meaningful parallel benefit due to overheads. At 10K vertices, both parallel implementations are faster than the sequential version. At 100k, edge parallel gets to 21x speedup and frontier gets 15x. Then at 500k speedup goes to 46x and 27x respectively. The plot of the execution time (log scale), shows the sequential grows somewhat linear with the number of vertices. The GPU implementations increase slower which is a sign that as the graph size gets bigger, the computation dominates all of the overheads (parallelism is being fully utilized).

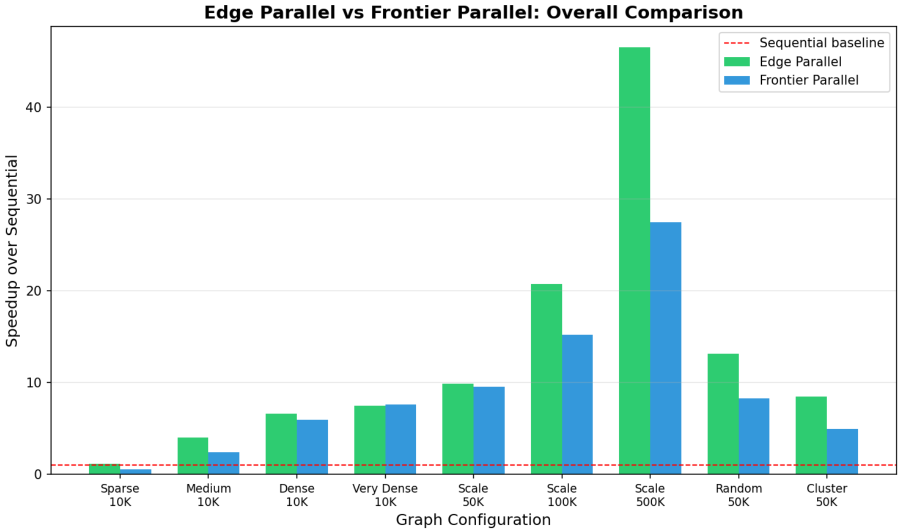
4.3 Effect of graph structure (50k vertices)





With 50K vertex graphs using different structures, the performance is highly dependent on the amount of parallel work per iteration. The random graph (with 500k edges), the edge parallel version gets around 13x speedup and frontier 8.3x. For the clustered graph, speedups are approximately 8.5x and 4.9x. These types of graphs have similar properties where they have many edges and small diameters. Meaning, the shortest path distances can be found in few iterations, where each iteration has large frontiers. In both parallel implementations, the GPU has enough work to get meaningful performance, but the edge version especially outperformed the frontier version. For the grid structure graphs (2D grid), speedups drop significantly, the edge parallel has a speedup of 1.4x and the frontier has 0.73x speedup. The grid structure has low degree and in general longer shortest paths, meaning more iterations with less edges in each iterations. This results in the overhead dominating where the frontier does not get smaller in proportion to this. In an additional extreme case, with a chain (successive edges) shows the limits of GPU parallelism where the sequential version finishes in approx 0.1 ms using in place relaxations and early exit. Both GPU versions are much slower because they still launch full kernel going over all edges and additionally have overheads from kernel launch and synchronization with almost no meaningful parallelism.

4.4 General Comparison



In general, the edge parallel implementation of Bellman Ford generally beats the frontier parallel implementation in terms of performance with these synthetic graphs. The maximum speedups found were 46x for edge parallel and 27x for frontier parallel over the sequential baseline version. The frontier parallelism implementation struggles to outperform the edge one due to the fact that on these generated graphs, frontiers end up including most vertices. This leads the algorithm to do almost as much work as the edge version while dealing with the additional overhead from having frontiers and extra kernels.

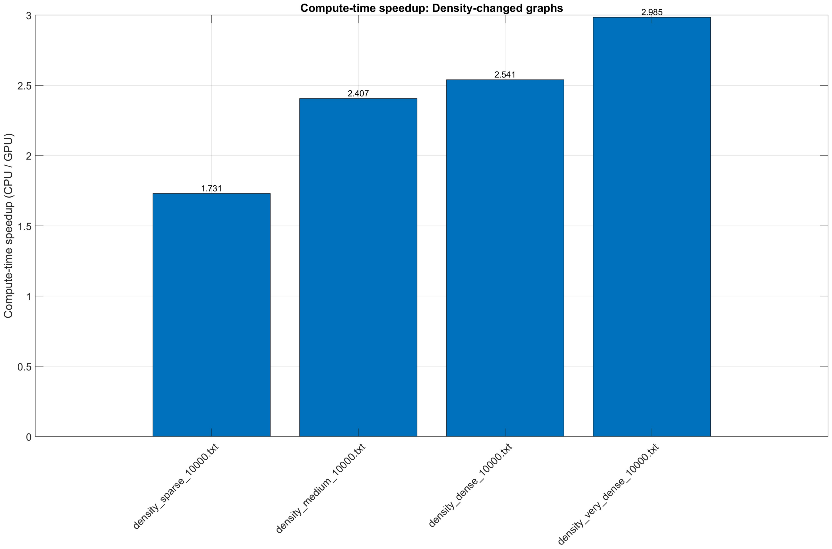
The main limiting factors of the Bellman Ford approaches is first having long diameters that result in many global iterations. Second, small graphs where things like kernel launches and synchronization dominate. Third, atomic contention from high degree vertices, which can be seen in dense graphs (amortized by fully utilized parallelism).

from these results, its clear that the GPU is great for Bellman Ford when used on large, relatively dense graphs. The parallel versions gives meaningful speedups when the graphs are large enough, while the sequential version outperforms for tiny structured graphs where the algorithm finds the shortest path in few easy passes.

5 Results for Delta-Stepping

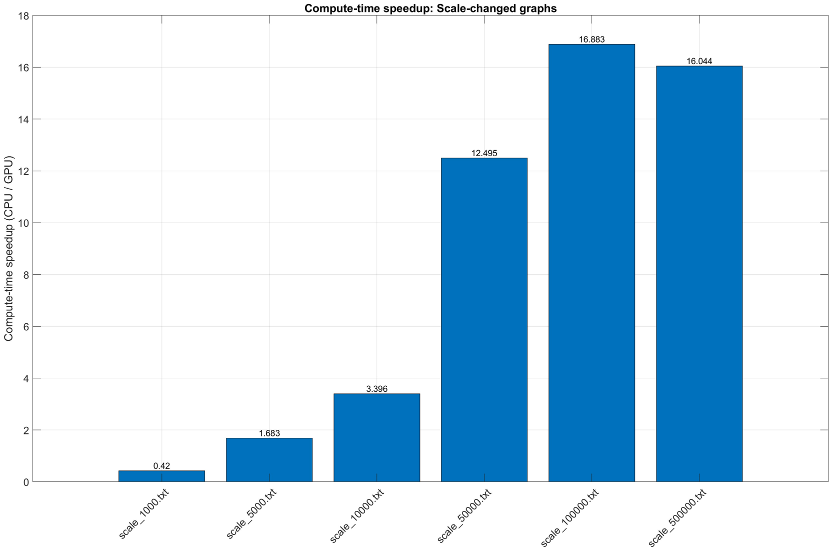
We used a similar methods for evaluating the Delta-Stepping implementations as with Bellman Ford. Performance was measured using wall clock time for the computation runtime of each one, and speedup ($\frac{T_{CPU}}{T_{GPU}}$) was the main metric. The same types of synthetic graphs were generated and used, varying in density, size, and structure. The checker script was also adapted to verify that the results from both implementations matched. We selected Δ values based on preliminary experiments to optimize performance for the parallel implementation. Also, additional real-world datasets from the SNAP repository and DIMACS collection were used to evaluate potential practical performance.

5.1 Effect of Edge density



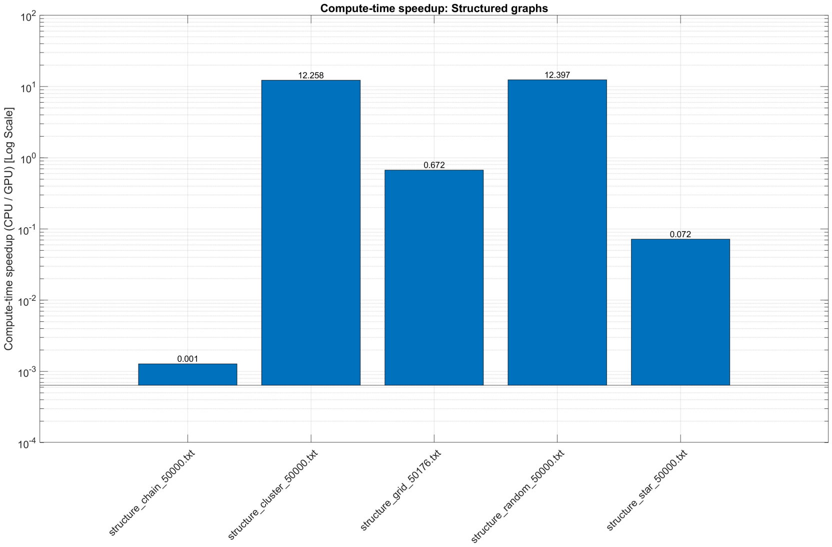
When using the same density scaling graphs as with Bellman Ford (10K vertices and 20K, 100K, 500K, and 1M edges), we observed that as the edge density increased, the parallel Delta-Stepping implementation achieved higher speedups compared to the sequential Dijkstra baseline. As density increases, the frontier contains more edges per active vertex, which lets the heavy-edge path (warp-cooperative processing) and the light-edge loops expose more parallel work per bucket. The fact that the speedup tops out around about 3x rather than growing unbounded with density, suggests you're starting to run into global memory / atomic throughput limits or the cost of the global barrier and queue-management logic, rather than raw ALU throughput.

5.2 Effect of Graph size



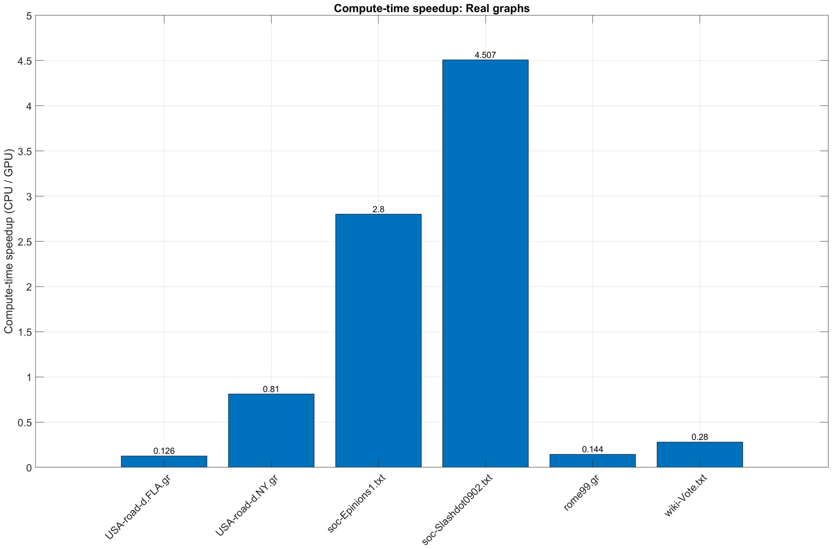
Using the same size scaling graphs as with Bellman Ford (1K, 5K, 10K, 50K, 100K, and 500K vertices with 10 edges per vertex), we found that while the small 1K graph did not benefit from parallelism (speedup ¡ 1x), as the graph size increased, the parallel Delta-Stepping implementation achieved progressively higher speedups. At 10K vertices, we observed a speedup of around 3.3x, which increased to approximately 16.8x at 100K vertices, which was kind of peaked as at 500K vertices the speedup was still around 16x. This indicates that the delta-stepping algorithm scales almost linearly with the number of edges, while the overhead of each global iteration (global barrier, reading counters, switching queues) doesn't grow much. Once the graph is big enough, those overheads are amortized and you get strong GPU wins in the pure SSSP phase.

5.3 Effect of graph structure



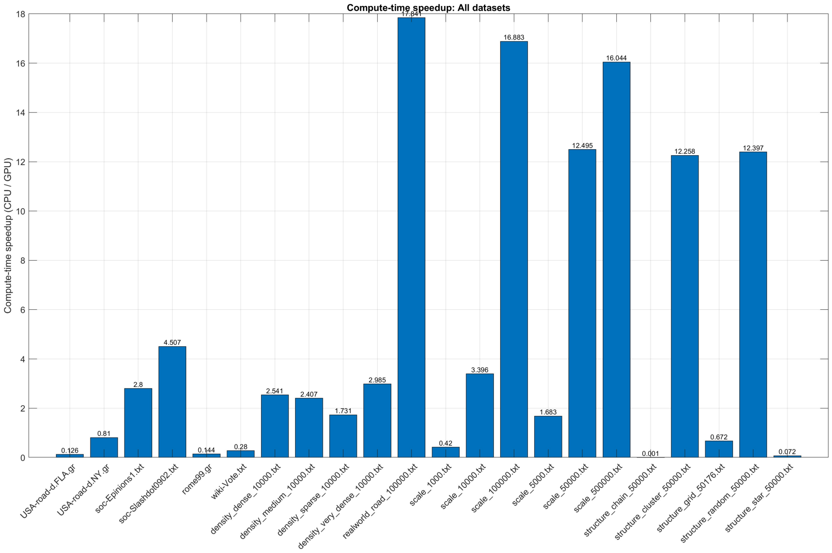
Using the same structure scaling graphs as with Bellman Ford (50K vertex graphs that have random, clustered, grid, chain, road like (sparse), and star structures), we observed that the parallel Delta-Stepping implementation's performance is highly dependent on the graph structure. For random and clustered graphs, which have relatively high average degrees and small diameters, we observed speedups of around 12x compute speedup, since there are more edges that could be relaxed in parallel within each bucket. In contrast, for grid structures, the speedup dropped to around 0.67x, indicating that the overheads of the parallel implementation outweighed the benefits due to the low degree per vertex and longer shortest paths. The chain and star structures shown the worst speedups (0.001x and 0.07x respectively), as they have extremely limited parallelism potential, since the frontier often has one or a handful of vertices, so nearly all threads in a warp are idle while still paying for $global_{s}ync$, atomic queue pushes, and the global while-loop. This is very likely severe SIMT underutilization plus synchronization overhead, not bandwidth.

5.4 Real-world datasets

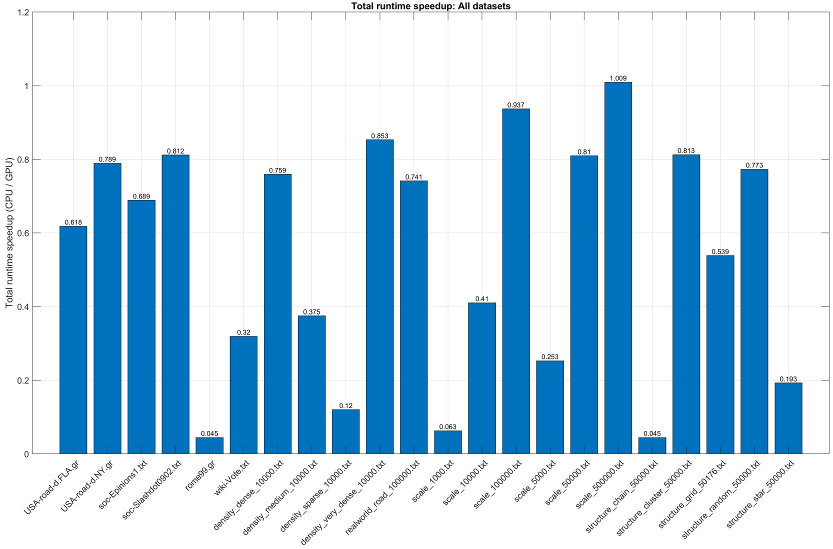


Using real-world datasets from the SNAP repository and DIMACS collection, we found that the parallel Delta-Stepping implementation only achieves noticeable speedup on social networks. $soc-Epinions1$ and $soc-Slashdot0902$ see around 2.8x and 4.5x compute speedup. In contrast, the road networks and tiny graphs are worse: $USA - road - d.FLA$ is about 0.13x, $rome99$ about 0.14x, and $wiki - Vote$ about 0.28x. This lines up well with what delta-stepping likes: wide, irregular frontiers with lots of independent edges (social graphs) vs long, skinny, low-degree graphs (roads, small graphs) that offer little intra-bucket parallelism, so most warps idle while the persistent kernel still pays for global barriers and atomics.

5.5 General Comparison



Overall, the parallel Delta-Stepping implementation would work just as expected from the algorthmic design. For Large, irregular graphs, it shows strong GPU advantages in the SSSP kernel. On the other hand, graphs that are tiny, long-diameter, or structurally "thin" cluster below 1x speedup, and sometimes even lower, due to overheads and SIMT under utilization. The lack of speedup on some graphs is therefore best explained by algorithmic dependence and poor parallelism, plus the fixed per-iteration overhead of the persistent-kernel loop and global barriers.



When considering the total speedup including data transfer times, we could realized that the GPU initialization for the Delta-Stepping implementation is relatively expensive compared to the computation time, indicating that potential speedups could be further improved by optimizing data transfer and kernel launch overheads. This is especially true for smaller graphs where the computation time is already low, making the initialization overhead a significant portion of the total runtime. For larger graphs, while the computation time dominates, there is still room for improvement in reducing data transfer times to further enhance overall performance.

6 References

The Delta-Stepping algorithm is based on the original paper by Meyer and Sanders [1], and further optimizations were inspired by the work of Wang et al. [2] and Davidson et al. [3]. The real-world datasets were sourced from the SNAP repository [4] and the DIMACS collection [5]. The Bellman Ford algorithm was designed based on the CMU 15-210 textbook explanation [6].

7 List of Work by Each Member

The project was completed with two team mebers simultaneously working on different algorithms and implementations. Eric worked on the Bellman Ford implementations, including both the edge parallel and frontier parallel versions, as well as the synthetic graph generation and verification scripts. Daniel focused on the Delta-Stepping implementations, including both the sequential Dijkstra baseline and the parallel CUDA version, along with the evaluation on real-world datasets. Both members collaborated on the overall project design, performance evaluation, and report writing. Marks should be split based on the performance of each implementation, and the reporting aspects should be shared equally.

References

[1] U. Meyer and P. Sanders, “Delta-stepping: A parallelizable shortest path algorithm,” *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003, ISSN: 0196-6774. DOI: `https://doi.org/10.1016/S0196-6774(03)00076-2`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0196677403000762`.

[2] K. Wang, D. Fussell, and C. Lin, “A fast work-efficient sssp algorithm for gpus,” in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 133–146.

[3] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, “Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths,” in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 349–359. DOI: `10.1109/IPDPS.2014.45`.

[4] *Stanford Large Network Dataset Collection.* Accessed: Nov. 17, 2025. [Online]. Available: `https://snap.stanford.edu/data/`.

[5] *9th DIMACS Implementation Challenge: Shortest Paths.* Accessed: Nov. 17, 2025. [Online]. Available: `https://www.diag.uniroma1.it/challenge9/download.shtml`.

[6] *Bellman Ford's Algorithm*, Carnegie Mellon University, 15–210. Accessed: Nov. 18, 2025. [Online]. Available: `https://www.cs.cmu.edu/~15210/algobook/shortest-paths/bellmanford.pdf`.