WarpNet: Parallel BFS and DFS on CUDA GPUs

Tanvin Kalra
tkalra@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Abstract

Graph traversal algorithms such as Breadth-First Search (BFS) and Depth-First Search (DFS) are foundational in computer science, with applications in route planning, social network analysis, and more. This paper investigates GPU-accelerated implementations of BFS and DFS using CUDA. We present parallel BFS using level-synchronous frontier expansion, bitmap optimizations, and warp-centric designs. For DFS, we target Directed Acyclic Graphs (DAGs) and leverage topological sorting to enable parallelism. Benchmarks on real and synthetic datasets demonstrate where these optimizations shine or falter, emphasizing the role of graph structure in parallel efficiency.

CCS Concepts

• **Computing methodologies** → *Concurrent algorithms*; **Parallel algorithms**.

Keywords

Breadth First Search, Depth First Search, Parallel Algorithms, CUDA, Frontier expansion

ACM Reference Format:
Tanvin Kalra. 2025. WarpNet: Parallel BFS and DFS on CUDA GPUs. In *Proceedings of (Symposium on Parallelism in Algorithms and Architectures).* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

Breadth-First Search (BFS) and Depth-First Search (DFS) are fundamental graph traversal algorithms with broad applications in social network analysis, route planning, web crawling, and many other domains [7, 10]. BFS systematically explores graph layers outward from a source, making it essential for shortest-path and connectivity algorithms [1], while DFS dives deep along one path before backtracking, which is useful for tasks like cycle detection, topological sorting, and planning [7].

Given their ubiquity and importance in real-world applications, accelerating BFS and DFS can greatly benefit systems that handle large-scale graphs (e.g., road networks for route optimization, social graphs, etc.). Indeed, BFS is a core primitive in many graph analytics benchmarks [6], and DFS underlies procedures such as topological sort and planarity testing [7].

However, these traversals pose different challenges for parallelization. BFS naturally exposes frontier parallelism since all neighbors of a given layer can be visited concurrently. In contrast, DFS's inherently sequential nature (exploring one path at a time) makes it difficult to parallelize on conventional architectures [3]. GPUs offer massive parallelism, but harnessing it for graph traversals requires careful algorithm design to address irregular memory access patterns and load imbalance.

We focus on BFS and DFS because they represent two extremes of graph traversal: BFS is often highly parallelizable, whereas DFS is generally sequential, and exploring their optimization on GPUs provides insights applicable to a range of graph algorithms.

In this work, we target CUDA-enabled GPUs to accelerate BFS and DFS. We choose BFS and DFS as exemplars to demonstrate optimization techniques like frontier bitmaps and warp-centric processing for irregular workloads. By optimizing BFS, we aim to improve performance in scenarios such as shortest path computations on road networks or social graphs. By attempting to parallelize DFS on Directed Acyclic Graphs (DAGs), we explore methods to handle inherently sequential tasks in parallel. The following sections detail our approaches, implementations, and empirical results.

1 Breadth-First Search (BFS)

1.1 BFS Algorithm Review

BFS traverses a graph level by level, starting from a source vertex. All immediate neighbors of the source are visited first, then all unvisited neighbors of those neighbors, and so on. A queue data structure is traditionally used to manage the *frontier* (the current layer of vertices being expanded). Each vertex is marked with a distance (or level) from the source, initialized to infinity (unvisited), except for the source which is set to 0. In each iteration, the algorithm dequeues a vertex, explores all its adjacent vertices, and enqueues any that are discovered (marking their distance as the current vertex's distance + 1). This continues until the queue is empty.

Listing 1: Sequential BFS pseudocode

```
procedure BFS(Graph G, Vertex s):
    for each vertex v in G:
        v.distance := ∞
    s.distance := 0
    Queue Q := [s]
    while Q is not empty:
        u := Q.pop_front()
        for each neighbor v of u:
            if v.distance == ∞:
                v.distance := u.distance + 1
                Q.push_back(v)
```

1.2 Graph Representation (CSR)

For GPU traversal, I store the graph in Compressed Sparse Row (CSR) format, which is efficient for parallel neighbor expansion [4]. An array nodes of length $|V| + 1$ stores the prefix sum of neighbor counts (so that neighbors of vertex i are in `edges[nodes[i] ... nodes[i+1]-1]` in the edges array). This compact adjacency list representation allows coalesced memory accesses when threads load neighbors for expansion. We also maintain an array `distance` of size $|V|$ for distances (or levels), initialized to `INF` (a sentinel for unvisited), and a boolean `visited` array (or bitmask) to mark discovered vertices.

1.3 Implemented BFS Strategies

In this work, I implemented and evaluated three different GPU-based Breadth-First Search (BFS) strategies to analyze how various parallelization techniques perform across graph structures. These include a level-synchronous queue-based kernel, a bitmap-based frontier kernel, and a warp-centric optimized kernel.

1.3.1 Level-Synchronous Queue-Based BFS. The first implementation follows a classic level-synchronous BFS approach using explicit queues to manage the current and next frontiers. In each iteration (or level), GPU threads are assigned to vertices in the current frontier. Each thread explores the neighbors of its assigned vertex and adds unvisited neighbors to the next frontier using atomic operations.

This approach is straightforward to implement and performs reasonably well when frontiers are not sparse. However, it suffers from several limitations: atomic contention during concurrent frontier updates, uncoalesced memory access patterns, and imbalance caused by uneven vertex degrees [9].

1.3.2 Bitmap-Based Frontier BFS. To reduce contention and simplify synchronization, I implemented a BFS variant using bitmaps to represent the current and next frontiers. In this model, each vertex is assigned a bit in a bitmap array. Threads scan vertices in parallel and check whether their assigned vertex is active in the current frontier. This is achieved using the following conditional check:

Listing 2: Bitmap frontier check in BFS

```
if (!(frontier_bitmap[v / 32] &
                    (1U << (v % 32))))
    return;
```

If the bit is set, the thread processes all neighbors and marks any newly discovered vertices in the next frontier using atomic bitwise operations. This method eliminates the need for explicit dynamic queues and improves scalability, especially for denser frontiers. However, for graphs with sparse frontiers, launching a thread per vertex may result in underutilization [5].

1.3.3 Warp-Centric Bitmap BFS. To address warp divergence and inefficient memory access in irregular graphs, we implemented a warp-centric variant of the bitmap-based BFS. Instead of each thread independently processing a vertex, threads in a warp collaborate to process the neighbor list of a single high-degree vertex.

The key loop of this warp-cooperative expansion looks like:

Listing 3: Warp-centric neighbor expansion

```
for (int i = start + lane_id; i < end; i += 32)
{
    int neighbor = col_indices[i];
    if (atomicCAS(&distances[neighbor],
            INT_MAX, level) == INT_MAX)
            {
        atomicOr(
        &next_frontier_bitmap[neighbor / 32],
        1U << (neighbor % 32));
            }
}
```

Here, ‘lane_id‘ refers to the thread's index within the warp. This loop ensures that neighbor expansion is performed in a coalesced and SIMD-friendly manner, significantly improving memory throughput and reducing warp divergence [4].

1.3.4 Comparison and Trade-offs. Each strategy presents unique trade-offs:

- **Queue-based BFS** is intuitive and efficient on moderately sized frontiers but suffers from atomic update overhead.
- **Bitmap-based BFS** reduces atomic contention and synchronization complexity but may be inefficient for sparse frontiers due to excess thread launches.
- **Warp-centric BFS** provides the best SIMT efficiency and memory coalescing, particularly for graphs with high-degree vertices.

Ultimately, the choice of strategy depends on the graph structure and the characteristics of the workload.

1.4 Implementation Challenges in BFS

During development and tuning of our BFS kernels, two key implementation-level challenges emerged that directly influenced performance:

Block Size and Occupancy. The choice of CUDA thread block size (`BLOCK_SIZE`) plays a crucial role in overall kernel performance. A larger block size (e.g., 512 threads) increases the number of warps available per block, allowing a Streaming Multiprocessor (SM) to better hide memory latency. However, it can also increase register pressure and shared memory usage, particularly when using warp-cooperative methods, potentially limiting the number of resident blocks per SM and thus reducing occupancy.

Conversely, smaller block sizes (e.g., 128 or 256 threads) consume fewer resources per block, which allows more blocks to run concurrently on each SM, improving overall occupancy. In our experiments, we selected a block size of 256 threads as a balance between active warps and resource utilization. This configuration worked well for our memory-bound BFS kernels, but we note that the optimal size is hardware-dependent and may vary across architectures. A brief tuning sweep was performed to identify this optimal value on the RTX 2070.

Distance Initialization Overhead. Proper initialization of the `distance[]` array is critical to ensure correctness and reproducibility in BFS benchmarking. Setting each entry to `INT_MAX` (0x7FFFFFFF) cannot be achieved using `cudaMemset`, which only sets uniform byte values. Similarly, performing the initialization on the CPU becomes a significant bottleneck for graphs with tens of millions of vertices.

To address this, I implemented a dedicated GPU kernel `set_int_max` that launches thousands of threads to initialize the distance array in parallel. This dramatically reduces setup time and ensures consistent starting conditions for each run. This overhead is incurred only once at the beginning of each iteration and enables us to focus on measuring the actual BFS traversal time, not memory setup cost. A similar approach could be applied to initialize visited flags when needed.

These implementation choices, though seemingly low-level, are essential to achieving high performance and accurate benchmarking results in GPU-based BFS.

2 Benchmarking BFS Performance

2.1 Experimental Setup

All BFS benchmarking experiments were conducted on the ARC cluster using an NVIDIA RTX 2070 GPU. The benchmarking code included a warmup phase, correctness validation, and dynamic iteration selection similar to the benchmarking methodology used in Homework 2. The number of BFS iterations was adjusted based on the time taken for one run to ensure consistent and fair comparisons across datasets.

2.2 Dataset Description

I evaluated my BFS implementations on a suite of real-world graphs from the DIMACS10 benchmark competition, specifically from the *OpenStreetMap (OSM)* and *road network* datasets. These graphs represent large-scale, sparse networks with low average degrees and large diameters—challenging cases for many GPU traversal strategies.

The datasets and their structural properties are shown in Table 1. Data was obtained from the Sparse Matrix Collection hosted at Texas A&M University [2].

Table 1: DIMACS10 OSM and road datasets used for BFS benchmarking

Dataset	Nodes	Edges
luxembourg_osm	114,599	119,666
belgium_osm	1,441,295	1,549,970
great-britain_osm	7,733,822	8,156,517
germany_osm	11,548,845	12,369,181
asia_osm	11,950,757	12,711,603
europe_osm	50,912,018	54,054,660
road_central	14,081,816	16,933,413
road_usa	23,947,347	28,854,312

2.3 BFS Runtime Results

We evaluated four BFS implementations:

(1) **Sequential BFS:** CPU-based single-threaded reference implementation.
(2) **Frontier (CUDA):** Naive queue-based level-synchronous BFS with frontier array on GPU.
(3) **Bitmap Frontier:** Bitmask-based GPU BFS to reduce atomic contention.
(4) **Warp-Centric:** GPU kernel optimized to exploit warp-level cooperation for neighbor exploration.

2.4 Analysis and Discussion

Ideally, each successive BFS implementation (naive $\rightarrow$ bitmap $\rightarrow$ warp-centric) would outperform its predecessor by reducing synchronization overhead and improving memory access patterns. However, our benchmarking revealed that this was not always the case, especially on sparse graphs like road networks and OSM maps.

Bitmap vs. Queue-based Frontier: On sparse graphs, the number of active vertices in each level (the frontier) is small compared to the total vertex count. As a result, launching threads for every node and checking bitmap membership adds overhead, especially when few nodes are active. The queue-based CUDA implementation, which processes only active vertices, is more efficient in these scenarios.

Warp-Centric Strategy: This approach is particularly designed to benefit graphs with high variance in vertex degree, such as scale-free networks or social graphs. However, road networks and OSM graphs tend to have uniform, low-degree distributions (average degree $\approx$ 2–3). This results in underutilized warps and minimal gains from intra-warp collaboration. In some cases, the overhead of warp-wide coordination outweighs any benefits.

General Observation: All GPU implementations significantly outperformed the CPU sequential baseline (often by 10x–100x). However, the expected performance hierarchy between naive, bitmap, and warp-centric implementations did not always hold due to the sparsity and structure of the datasets.

2.5 Takeaways

- GPU acceleration of BFS offers massive gains on real-world road and map graphs.
- Bitmap frontiers are effective when frontiers are dense; otherwise, they introduce unnecessary scanning overhead.
- Warp-centric BFS is best suited to graphs with irregular degree distributions.
- Real-world sparse graphs like road networks challenge these optimizations due to their uniformity and low degree.

3 Depth-First Search (DFS)

3.1 DFS Algorithm and Challenges

Depth-First Search (DFS) explores a graph by going as deep as possible along each branch before backtracking. In a sequential recursive implementation, DFS uses a stack (implicitly through recursion or explicitly in an iterative version) to remember the current traversal path. Unlike BFS, DFS does not explore vertices level by level but instead may follow one long path before considering alternatives.

Listing 4: Sequential DFS (recursive pseudocode)

```
procedure DFS(Graph G, Vertex u):
    visited[u] := true
```

Table 2: BFS runtime (in milliseconds) across implementations and datasets

Dataset	Sequential BFS (CPU)	Frontier (CUDA)	Bitmap	Warp-Centric
luxembourg_osm	3.1598	0.0324	0.0273	0.0572
belgium_osm	93.5074	0.0508	0.1258	0.3913
great-britain_osm	582.5160	0.1404	0.5339	2.0515
germany_osm	966.2730	0.1957	0.7051	2.9091
asia_osm	693.0681	0.2041	0.7171	3.0877
europe_osm	4545.5850	0.7700	2.1939	14.8139
road_central	1350.9464	0.3783	0.8607	3.5220
road_usa	1876.9289	0.2370	1.1148	6.7423

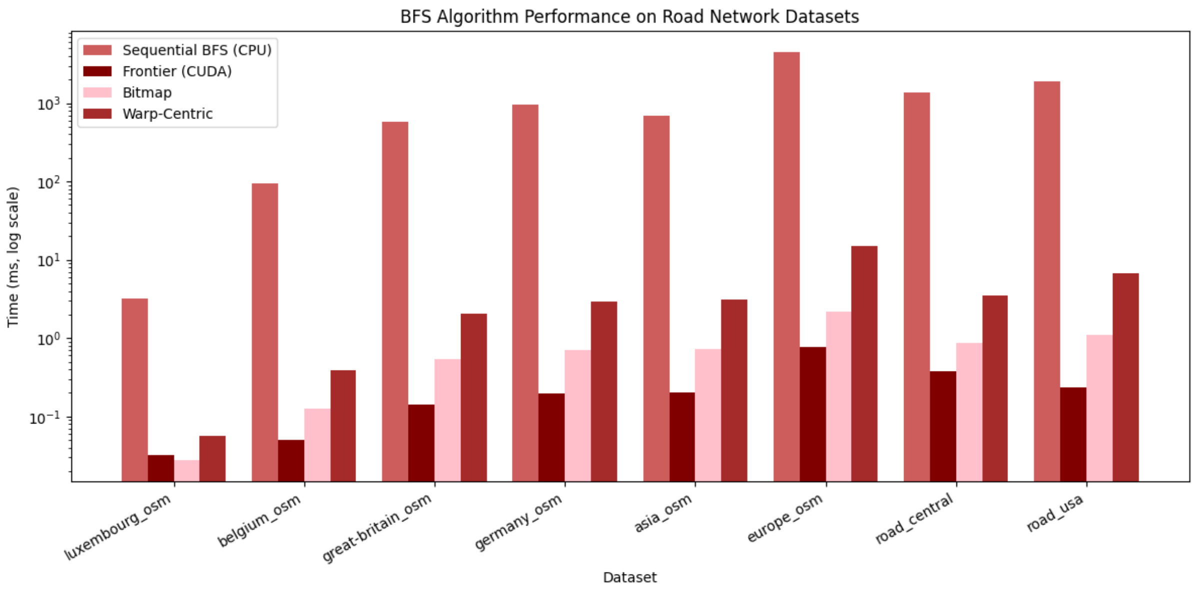

Figure 1: Comparison of BFS runtime (log scale) across datasets and implementations. GPU implementations consistently outperform the sequential CPU baseline. Bitmap and warp-centric methods vary in benefit depending on graph structure and sparsity.

```
    for each neighbor v of u:
        if not visited[v]:
            DFS(G, v)
```

In the above pseudocode, a new recursive call is made immediately upon discovering an unvisited neighbor, establishing a depth-first traversal pattern. This results in an inherent dependency chain: one path must finish (or at least pause) before another can be pursued. Consequently, parallelizing DFS in general graphs is non-trivial.

For example, in graphs with cycles, uncontrolled parallel DFS may revisit nodes concurrently, requiring complex synchronization to ensure correctness and prevent race conditions. This challenge arises from the need to preserve DFS's structural integrity while avoiding duplicated work or inconsistent state updates.

However, in specific graph classes such as Directed Acyclic Graphs (DAGs) or forests (acyclic undirected graphs), there exists more opportunity to exploit parallelism. In a DAG, each node's descendants are independent as long as all of their predecessors have been processed. This property lends itself to a topological sort-based parallel DFS strategy.

If a partial order of node visitation is known or can be computed, then DFS subtrees rooted at different independent nodes can be traversed concurrently. In our implementation, we leverage this by using a parallel topological sort to identify such traversal opportunities and enable concurrency in DFS execution on GPUs.

3.2 Parallel DFS on DAGs via Topological Sorting

A Directed Acyclic Graph (DAG) can be topologically sorted into a linear order such that every directed edge (u, v) satisfies u appears before v in the order. While a topological sort does not reflect a true DFS traversal, it respects dependencies and provides a valid order in which nodes can be processed. This forms the basis of our parallel approach to DFS on DAGs.

Our algorithm is based on a parallel version of Kahn's algorithm, which processes the graph in rounds and is structurally similar to BFS. The key insight is that a node in a DAG can be safely visited (in DFS-like order) once all of its parent dependencies have been satisfied. That is, if a node's in-degree (number of incoming edges from unvisited nodes) becomes zero, it is eligible to be visited.

We begin by initializing an in-degree array for all nodes in the graph. Nodes with zero in-degree are identified as sources and placed in the initial frontier. The algorithm proceeds in rounds:

- Each thread processes one node u in the current frontier.
- The node is marked as visited, and its topological position is recorded using an atomic counter.
- For each outgoing edge (u, v), the thread atomically decrements the in-degree of v.
- If the in-degree of v becomes zero, v is added to the next frontier.

This procedure continues until all nodes have been visited, thereby generating a topological sort of the DAG. While this approach does not produce a DFS tree or strict discovery/finish times, it aligns with the structure of DFS traversal in acyclic graphs and can be extended for such metadata if needed.

Kernel Operation: The core of our implementation is encapsulated in the `topological_step` kernel, where each thread executes the following:

Listing 5: Parallel topological traversal step (simplified pseudocode)

```
int u = frontier_curr[tid];
int pos = atomicAdd(topo_counter, 1);
topo_order[pos] = u;

for (int i = row_offsets[u];
    i < row_offsets[u + 1];
    i++) {
    int v = col_indices[i];
    int old_val = atomicSub(&in_degree[v], 1);
    if (old_val == 1) {
        int next_pos = atomicAdd(
        next_frontier_size, 1);
        frontier_next[next_pos] = v;
    }
}
```

Here, `frontier_curr` contains all currently active nodes (i.e., nodes with in-degree 0), and `frontier_next` collects newly activated nodes for the next round. Each thread atomically updates a global `topo_order` array with the node's position in the topological traversal.

Host Function: The function `benchmark_dfs_cuda` orchestrates memory allocation, initialization of the in-degree array, and repeated kernel launches. It initializes all in-degrees based on the CSR graph structure and identifies initial source nodes. In each iteration, it launches the `topological_step` kernel, swaps frontiers, and continues until all nodes have been visited.

4 Benchmarking DFS on Synthetic DAGs

4.1 Experimental Setup

All experiments were performed on the ARC computing cluster using an NVIDIA RTX 2070 GPU. The implementation was tested using CUDA 11.x with default optimization flags enabled. To ensure consistency, each test was run multiple times and averaged to account for variations in GPU scheduling.

4.2 Synthetic Dataset Generation

Due to the lack of publicly available large-scale DAGs suitable for DFS benchmarking, we generated synthetic DAGs using a custom Python function. The DAGs are created with varying sizes and edge densities using NetworkX:

Listing 6: DAG generation procedure

```
def generate_random_dag(num_nodes, edge_prob):
    G = nx.DiGraph()
    G.add_nodes_from(range(num_nodes))
    for i in range(num_nodes):
        for j in range(i + 1, num_nodes):
            if random.random() < edge_prob:
                G.add_edge(i, j)
    return G
```

The datasets used for benchmarking are listed below:

Table 3: Synthetic DAG dataset characteristics

Nodes	Edges
5000	4,997,475
10000	19,998,546
20000	80,002,251

4.3 Runtime Comparison

We compare the performance of two traversal methods:

- **Sequential DFS (CPU):** A single-threaded recursive DFS implementation executed on the CPU.
- **Parallel Topological Sort (GPU):** A Kahn's algorithm-style GPU implementation that visits all nodes in dependency-respecting order.

Table 4: Runtime comparison of Sequential DFS (CPU) vs. Topological Sort (GPU)

Dataset	DFS (ms)	Topological Sort (ms)
5000, 4.9M	18.80	493.85
10000, 19.9M	72.95	1902.47
20000, 80.0M	288.95	7520.81

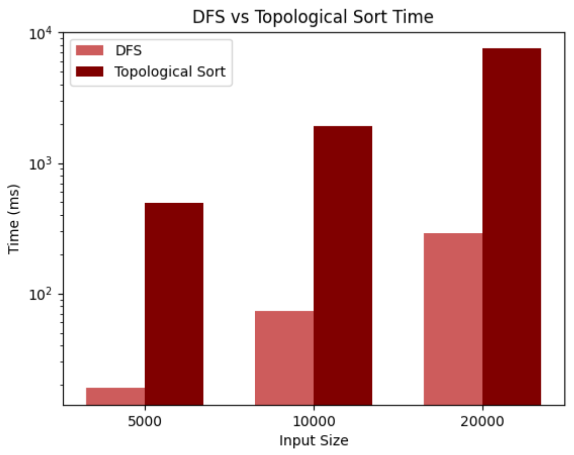

Figure 2: Bar chart comparing sequential DFS (CPU) with parallel topological sort (GPU) across synthetic DAGs. The CPU DFS performs better on connected graphs with deep dependencies.

4.4 Discussion

The GPU-based topological sort approach performed significantly worse than the sequential CPU DFS in all tested datasets. This result is expected, given the structure of the graphs: each DAG was essentially a single large connected component with deep dependency chains. Consequently, the GPU algorithm progressed in many fine-grained rounds, each processing only a small number of ready nodes. This led to substantial overhead from frequent kernel launches and GPU synchronization.

In contrast, the sequential DFS on the CPU traversed the graph deeply and continuously, leveraging recursion with minimal overhead. Because the entire graph was connected, the CPU could walk through nodes efficiently without needing to manage global state or track frontier readiness in rounds.

The underperformance of the GPU topological sort can be attributed to several structural and architectural factors:

- **High indegree variance:** Many vertices had incoming edges, delaying their eligibility and reducing frontier size during early rounds.
- **Large sequential depth:** The graphs had long dependency chains, forcing the GPU into $O(N)$ rounds, each with minimal active work.
- **Single-component DAGs:** All datasets lacked disconnected components, removing opportunities for parallel traversal of independent subgraphs.

I hypothesize that on datasets with multiple small disconnected DAGs or broad forest-like structures, the GPU topological approach would perform better, as more subgraphs could be traversed concurrently. However, our synthetic DAGs were specifically constructed to stress-test depth and density scaling, rather than width or component parallelism.

Overall, the sequential DFS outperformed the GPU topological traversal in both runtime and efficiency for the tested graphs. These results underscore the importance of aligning the traversal strategy with the graph's structural properties. GPU-based methods offer scalability but are highly sensitive to dependency depth and parallel workload availability. In contrast, CPU-based DFS remains competitive on deeply connected or narrow graphs due to its simplicity and low-overhead execution model.

5 Conclusion and Future Work

In this work, I explored approaches to accelerate BFS and DFS graph traversals on CUDA-enabled GPUs. For BFS, the focus was on bitmap frontier-based methods and warp-centric optimizations. For DFS, I evaluated a GPU-parallel topological sort strategy designed for DAGs and compared it to a standard sequential DFS running on the CPU.

For BFS, the results confirmed that GPU-based traversal can offer substantial performance benefits on large graphs, though the efficacy of specific optimizations depends heavily on graph structure. The bitmap frontier implementation served well as a baseline but was not optimal for very sparse graphs where many threads do minimal work. The warp-centric variant offered modest performance improvements on road-network graphs, but we expect more substantial benefits on graphs with skewed degree distributions due to better warp utilization and coalesced memory access.

For DFS, the results highlighted both potential and limitations. On DAGs with ample parallelism, the GPU-based topological sort traversed nodes efficiently in parallel. However, in cases where the DAG resembled a long chain (high depth, low width), the GPU approach was slower than a simple recursive DFS on the CPU. This performance degradation stemmed from the overhead of launching many fine-grained GPU kernels and managing atomic updates across rounds. These results underscore the challenge of parallelizing inherently sequential algorithms like DFS and demonstrate that success is highly dependent on graph topology.

Future Work

There are several promising directions to extend this work:

- **Hybrid top-down/bottom-up BFS:** Inspired by direction-optimizing BFS [1], implementing a hybrid approach on GPUs that switches direction based on frontier density could reduce redundant computation and improve performance in dense graph regions.
- **Dynamic parallelism:** Exploring CUDA's dynamic parallelism (i.e., launching kernels from within kernels) might reduce host-device overhead for BFS, though its benefits are workload-dependent and constrained by warp divergence.
- **GPU-based full DFS order:** A valuable future direction would be to implement the DFS ordering algorithm from Naumov et al. [7], which computes pre- and post-visit times for DFS using multiple GPU passes. This would support use cases that require explicit DFS structure, such as dominator trees or planarity testing.
- **CPU-GPU hybrid DFS execution:** Investigating work partitioning strategies that delegate parallel-friendly parts of the graph to the GPU and retain serial components for the CPU (e.g., using work stealing or dynamic scheduling) could enhance performance on irregular DAGs.

- **Integration into GPU graph libraries:** Applying these traversal strategies in frameworks such as Gunrock [8] or GraphBLAST could improve accessibility and real-world adoption, especially when combining BFS and DFS into larger pipelines (e.g., SSSP or SCC).

In summary, while BFS and DFS present fundamentally different challenges for parallelization, both remain core primitives in graph analytics. This work shows that GPU acceleration can yield notable benefits, especially for BFS and for DFS in highly parallel DAGs, but also that caution is needed when applying parallelism to sequentially structured problems. Future improvements will likely emerge from hybrid models, library-level integration, and evolving hardware capabilities.

References

[1] A. Beamer, K. Asanović, and D. Patterson. 2012. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. Article 12. Introduces bottom-up BFS optimization on CPU.

[2] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. https://sparse.tamu.edu. doi:10.1145/2049662.2049663 Accessed April 2025.

[3] P. Harish and P. J. Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing (HiPC)*. 197–208. https://m-sp.org

[4] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 267–276. https://ppl.stanford.edu

[5] Lijuan Luo, Ming Liu, Martin D. F. Wong, and Wen mei W. Hwu. 2010. An effective GPU implementation of Breadth-First Search. In *Design Automation Conference (DAC)*. 52–55. https://m-sp.org

[6] D. Merrill, M. Garland, and A. Grimshaw. 2012. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://research.nvidia.com NVIDIA Technical Report NVR-2011-001.

[7] Maxim Naumov, Alysson Vrielink, and Michael Garland. 2017. *Parallel Depth-First Search for Directed Acyclic Graphs*. Technical Report NVR-2017-001. NVIDIA. https://research.nvidia.com

[8] Yangzihao Wang et al. 2017. Gunrock: GPU Graph Analytics. *ACM Transactions on Parallel Computing (TOPC)* 4, 1 (2017), Article 3, 49 pages. https://arxiv.org

[9] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. 2011. On-the-fly elimination of dynamic irregularities for GPU computing. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 369–380. https://m-sp.org

[10] J. Zhong and B. He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 25, 6 (2014), 1543–1552. Describes a GPU framework that supports BFS/DFS.

Appendix

The source code and additional implementation details for the parallel graph algorithms (BFS and DFS on GPU) are available in the following repository:

https://github.ncsu.edu/tkalra/parallel-graph-algorithms

Received 30 April 2025; revised 12 May 2025; accepted 5 June 2025