Parallel Breadth-First Search

GPU Acceleration with CUDA

Evangelos Moschou
`AEM: 10986`

January 3, 2026

Abstract

This report presents three CUDA-based BFS implementations exploring different parallelization strategies: dynamic thread assignment, chunk-based processing, and warp-cooperative shared memory with hybrid Top-Down/Bottom-Up. A critical correctness bug in the warp kernel was identified through rigorous verification against a Julia CPU reference implementation. The bug caused approximately 50% of edges to be silently skipped due to a mismatch between the defined neighbor chunk size (64) and actual warp size (32). After fixing this bug, the implementation achieves 100% correctness on the Friendster graph (65.6M nodes, 3.6B edges) with 541ms execution time, representing a 70x speedup over CPU and 6.67 GTEPS throughput.

1 Introduction

Breadth-First Search is a fundamental graph traversal algorithm with applications in social network analysis, shortest path computation, web crawling, and connectivity analysis. As graph datasets grow to billions of edges, GPU parallelization becomes essential for achieving practical performance.

This project implements three progressively optimized CUDA BFS algorithms, each exploring different trade-offs between simplicity, load balancing, and hardware utilization. The implementations handle graphs stored in Compressed Sparse Row (CSR) format, which provides $O(1)$ neighbor access and excellent cache locality. For graphs exceeding GPU VRAM capacity, we employ Zero-Copy Mapped Memory techniques to enable direct GPU access to host memory via PCIe.

2 Algorithm Design

2.1 Graph Representation

The CSR format consists of two arrays: `row_ptr[n+1]` defining adjacency list boundaries and `col_idx[m]` storing neighbor indices. This representation is ideal for GPU parallelism as it allows coalesced memory access patterns and efficient neighbor iteration. Graphs are cached as binary `.csrbin` files to reduce load times from minutes to seconds.

2.2 Version 1: Dynamic Thread Assignment

Each thread dynamically fetches work from a shared frontier queue using atomic operations. When a thread finishes processing its assigned node, it atomically increments a global counter to fetch the next available node from the frontier.

Advantages:

- Excellent load balancing for irregular graphs with varying node degrees
- Handles high-degree nodes efficiently by distributing work dynamically
- No wasted work from idle threads

Disadvantages:

- High atomic contention on the frontier counter, especially with many threads
- Unpredictable memory access patterns reduce cache efficiency
- Serialization bottleneck when many threads compete for work

2.3 Version 2: Chunk-Based Processing

Each thread processes a fixed chunk of frontier nodes using a simple for-loop, similar to CPU-style parallelization. This approach reduces atomic operations to one per thread (for adding discovered nodes to the next frontier) rather than one per work item.

Advantages:

- Simple, straightforward implementation
- Significantly reduced atomic contention compared to Version 1
- More predictable memory access patterns
- Better instruction cache utilization

Disadvantages:

- Potential load imbalance with irregular degree distributions
- High-degree nodes can create bottlenecks if assigned to a single thread
- Fixed chunk sizes may not adapt well to varying graph structures

2.4 Version 3: Warp-Cooperative Shared Memory

This version leverages CUDA's warp-level primitives and shared memory for maximum efficiency. It implements several advanced optimizations:

Warp Cooperation. Threads within a warp (32 threads) cooperatively process one frontier node's neighbors. Each warp loads up to 32 neighbors into shared memory, processes them in parallel, and uses `__ballot_sync` to aggregate results before writing to global memory. This reduces atomic operations by 32x (one per warp instead of per thread).

Hybrid Top-Down/Bottom-Up Strategy. The algorithm dynamically switches between two traversal modes based on frontier size:

- **Top-Down**: When frontier is small, expand from frontier nodes to their neighbors (standard BFS)
- **Bottom-Up**: When frontier exceeds 5% of nodes, unvisited nodes check if any neighbors are in the frontier

This hybrid approach reduces redundant edge traversals. For example, if 50% of nodes are in the frontier, Bottom-Up checks only unvisited nodes' edges rather than all frontier nodes' edges.

Streaming for Large Graphs. For graphs exceeding VRAM (e.g., Friendster's 13.5GB edge data on 5.8GB GPU), the algorithm streams chunks asynchronously using double-buffered `cudaMemcpyAsync`. While the GPU processes one chunk, the next chunk is transferred in parallel.

Memory Optimizations.

- **Zero-Copy**: `cudaHostRegister` with `cudaMemAdviseSetReadMostly` allows GPU to access host memory
- **Visited Bitmap**: 1 bit per node reduces memory bandwidth by 32x in Bottom-Up phase
- **Memory Prefetching**: `cudaMemPrefetchAsync` hints to driver for optimal data placement

3 Critical Bug Fix: Warp Kernel Correctness

3.1 Bug Discovery and Symptoms

During verification against a Julia CPU reference implementation, we discovered that `bfs_v3` was producing incorrect results:

- **Friendster**: 99.59% reachable (expected 100%) - missing 269,000 nodes
- **Mawi**: 47.54% reachable (expected 94.47%) - missing 106 million nodes

The bug was particularly severe on Mawi, suggesting a systematic issue rather than a race condition.

3.2 Root Cause Analysis

After extensive debugging, including testing different kernel variants and comparing against the simpler `bfs_v2` implementation (which achieved correct results), we identified the root cause in line 12 of `bfs_shared.cu`:

```
#define SHARED_NEIGHBORS_PER_WARP 64  // BUG: Warps only have 32 threads!
```

Listing 1: Buggy Code

The warp kernel attempted to process 64 neighbors per iteration, but CUDA warps contain only 32 threads (lanes 0-31). The kernel code was:

```
int lane_id = threadIdx.x % WARP_SIZE;  // 0-31
if (lane_id < chunk_size) {  // chunk_size could be up to 64
    s_neighbors[warp_id][lane_id] = col_idx[start + chunk_start + lane_id];
}
```

Listing 2: Kernel Logic

This caused neighbors at indices 32-63 in each chunk to be **silently skipped**, as `lane_id` never reaches values above 31. The bug resulted in approximately 50% of edges being ignored, explaining the severe node loss on Mawi.

3.3 Fix and Verification

```
#define SHARED_NEIGHBORS_PER_WARP WARP_SIZE  // Correctly uses 32
```

Listing 3: Fixed Code

After the fix, both graphs achieved 100% correctness:

- **Friendster**: 100.00% reachable (65,608,366 nodes), diameter 22
- **Mawi**: 94.47% reachable (213,682,593 nodes), diameter 7

Both results now match the Julia CPU reference exactly, confirming the fix.

4 Experimental Results

4.1 Test Environment

- **GPU**: NVIDIA GeForce RTX 3060 Laptop (Ampere architecture, sm_86)
- **VRAM**: 5.8 GB
- **System RAM**: 32 GB
- **CUDA**: Version 12.x
- **Compiler**: nvcc with -O3 optimization, -arch=sm_86

4.2 Benchmark Datasets

Friendster Social Network:

- Vertices: 65,608,366
- Edges: 3,612,134,270 (3.6 billion)
- Edge data size: 13.46 GB (exceeds VRAM)
- Max degree: 5,214
- Graph diameter: 22
- Structure: Real-world social network, power-law degree distribution

Mawi Internet Topology:

- Vertices: 226,196,185
- Edges: 480,047,894
- Edge data size: 1.79 GB
- Max degree: 210,795,477 (extreme supernode)
- Graph diameter: 7
- Structure: Internet routing graph with extreme degree skew

4.3 Performance Comparison

Graph	Version	Time (ms)	GTEPS	Reachable
Friendster	v3 (Fixed)	**541.7**	6.67	100.00%
Friendster	v3 (Buggy)	438.0	-	99.59%
Friendster	Julia CPU	38,000	0.095	100.00%
Mawi	v3 (Fixed)	**24,866**	0.019	94.47%
Mawi	v2 (Chunked)	101,373	0.005	94.16%
Mawi	v3 (Buggy)	23,334	-	47.54%

Table 1: BFS Performance. GTEPS = Giga Traversed Edges Per Second. The buggy version was faster because it incorrectly skipped half the edges.

Algorithm	Time (ms)	Components	Notes
BFS (Single Source)	541.7	-	100% reachable from source
Afforest (All Components)	21,993	1	Confirms single component

Table 2: Afforest (Union-Find) vs BFS on Friendster.

4.4 Analysis

GPU Speedup: The fixed implementation achieves 70x speedup over Julia CPU BFS on Friendster (38 seconds → 541 milliseconds), demonstrating the effectiveness of GPU parallelization for graph algorithms.

Zero-Copy Performance: Despite accessing 13.5GB of edge data via PCIe (exceeding the 5.8GB VRAM), the implementation achieves 6.67 GTEPS. This indicates that the memory access pattern is sufficiently favorable that PCIe bandwidth (16 GB/s) does not become a bottleneck.

Hybrid Strategy Effectiveness: On Friendster, the algorithm switches to Bottom-Up mode when the frontier exceeds approximately 3.3 million nodes (5% of total). This reduces the number of edge checks in later BFS levels where the frontier is large.

Bug Impact Analysis: The warp kernel bug caused a 50% edge skip rate (processing only 32 of 64 neighbors per chunk). This explains the 47.54% reachability on Mawi - approximately half the graph was unreachable due to missing edges. The fix restored correctness with a 24% performance penalty (438ms → 542ms), which is expected since the code now processes twice as many edges per iteration.

5 Conclusion

This project successfully implemented and optimized three CUDA BFS algorithms, with the warp-cooperative version (v3) achieving state-of-the-art performance after fixing a critical correctness bug. Key achievements include:

- **Sub-second execution** on billion-edge graphs (541ms for 3.6B edges)
- **70x speedup** over CPU reference implementation
- **100% correctness** verified against Julia CPU BFS
- **Zero-Copy support** enabling processing of graphs exceeding VRAM
- **6.67 GTEPS** throughput on real-world social network data
- **Hybrid Top-Down/Bottom-Up** strategy adapting to frontier density

The project demonstrates both the power of GPU acceleration for graph algorithms and the critical importance of rigorous correctness verification. The bug discovery process highlighted the value of comparing against independent reference implementations, as the error would have been difficult to detect through testing alone.

Source Code: `https://github.com/EvangelosMoschou/PkDSProject3.git`