Parallel Graph Algorithms

GPU Acceleration with CUDA: Adaptive BFS & Afforest

Evangelos Moschou
AEM: 10986

Abstract

This report presents advanced CUDA-based implementations of graph algorithms optimized for billion-scale graphs. The final implementation (V4) includes Adaptive BFS with compression and Afforest for connected components. Key innovations include Zero-Copy memory management for graphs exceeding VRAM, Delta-Varint compression reducing PCIe bandwidth by 37%, and GCC-pruned Afforest achieving 21.6s on Friendster (65.6M nodes, 3.6B edges). The compressed BFS achieves 4.5s traversal time with 2.6x speedup over uncompressed zero-copy BFS, demonstrating the effectiveness of bandwidth-aware optimizations for large-scale graph processing.

1 Introduction

Breadth-First Search and Connected Components analysis are fundamental graph algorithms with applications in social network analysis, shortest path computation, web crawling, and network topology discovery. As graph datasets grow to billions of edges, GPU parallelization becomes essential for achieving practical performance.

This project implements a production-grade graph processing system with the following features:

- **Adaptive BFS**: Dynamic frontier classification for load balancing
- **Delta-Compressed CSR**: Varint encoding for 30-40% bandwidth reduction
- **Afforest Algorithm**: Union-Find with GCC pruning for connected components
- **Zero-Copy Memory**: Direct GPU access to host RAM for graphs exceeding VRAM

The implementation handles graphs stored in Compressed Sparse Row (CSR) format, which provides $O(1)$ neighbor access and excellent cache locality. For massive graphs like Friendster (13.5GB edge data), we employ pinned memory and streaming techniques to enable processing on GPUs with limited VRAM.

2 Algorithm Design

2.1 Graph Representation

Standard CSR. 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.

Compressed CSR (c-CSR). To reduce memory bandwidth, we implement Delta+Varint compression:

1. Sort neighbors per row: $[5, 12, 8, 20] \rightarrow [5, 8, 12, 20]$
2. Compute deltas: $\Delta = [5, 3, 4, 8]$
3. Varint encode: Small deltas use 1 byte, large deltas use up to 5 bytes

The `row_ptr` array now stores **byte offsets** instead of edge indices, and `compressed_col` is a byte stream requiring on-the-fly decoding.

2.2 Adaptive BFS

The Adaptive BFS kernel dynamically classifies frontier nodes based on degree:

- **Small degree** ($<$ threshold): Process sequentially per thread
- **Large degree** ($\geq$ threshold): Cooperative warp processing

This classification reduces warp divergence and balances load across threads. The kernel uses shared memory for cooperative neighbor expansion and ballot sync for efficient atomic reduction.

2.3 Compressed BFS

The compressed BFS kernel employs a warp-cooperative strategy to decode Varint-encoded neighbors efficiently. Each warp (32 threads) processes one frontier node cooperatively. Lane 0 performs serial delta decoding: it reads the compressed byte stream, decodes each Varint to recover delta values, reconstructs absolute neighbor IDs by accumulating deltas, and stores results in shared memory. After synchronization via warp barrier, all 32 lanes process the decoded neighbors in parallel using a strided loop pattern (lane i processes neighbors $i, i+32, i+64, \ldots$).

This design balances the inherently serial nature of delta decoding (each neighbor depends on the previous) with parallel neighbor processing. The use of shared memory (48KB per SM) allows fast broadcast of decoded data to all warp lanes. For high-degree nodes, the kernel processes neighbors in chunks of 32, maximizing occupancy while minimizing shared memory pressure.

Varint Decoding Details: Each integer is encoded as 1-5 bytes. The decoder reads bytes sequentially, extracting 7-bit payloads and checking the continuation bit (bit 7). Small deltas (common in sorted neighbor lists) use only 1 byte, while large deltas use up to 5 bytes. This variable-length encoding achieves 30-40% compression on real-world graphs with power-law degree distributions.

Bandwidth Optimization: By streaming compressed data over PCIe, we reduce the effective bandwidth requirement from 14.4GB to 9.1GB per full graph traversal. The decoding overhead (7-10 cycles per Varint) is partially masked by PCIe latency ($\sim$400ns), making this approach highly effective for bandwidth-bound workloads.

2.4 Afforest Algorithm

Afforest is a GPU-optimized Union-Find algorithm for connected components:

Algorithm Phases.

1. **Init**: Each node is its own parent: $parent[i] = i$
2. **Link**: For each edge (u, v): $parent[\max(comp_u, comp_v)] = \min(comp_u, comp_v)$
3. **Compress**: Path compression: $parent[i] = parent[parent[i]]$

GCC Pruning Optimization. For graphs with a Giant Connected Component (GCC) containing the majority of nodes, we implement an early-exit optimization. Before processing each edge (u, v), the kernel performs path compression on both endpoints to find their current component roots $comp_u$ and $comp_v$. If both roots equal the pre-identified GCC ID (heuristically set to $parent[0]$ after initialization), the edge is skipped entirely.

This optimization is particularly effective on real-world graphs like Friendster where $> 99.9\%$ of nodes belong to a single giant component. By converting expensive atomic minimum operations (which require cache coherence traffic and potential retry loops) into cheap read-only comparisons, we reduce memory contention significantly. The pruning check adds negligible overhead (2 integer comparisons) while eliminating atomic writes for the vast majority of edges in later iterations.

Performance Impact: On Friendster, approximately 3.5 billion out of 3.6 billion edges connect nodes within the GCC. Without pruning, each edge triggers an atomic operation even when both endpoints are already merged. With pruning, these become no-ops, reducing atomic contention by

∼97%. However, the bandwidth bottleneck (reading compressed edge data) remains, limiting the overall speedup to ∼9% (21.6s vs 23.5s).

Single-Pass Strategy. Unlike traditional Afforest which requires convergence checking (2-3 passes), our optimized version runs a single pass, which is sufficient for graphs with small diameter. This reduces runtime from ∼46s to ∼21s on Friendster.

3 Memory Management for Large Graphs

3.1 Zero-Copy Strategy

For graphs exceeding GPU VRAM (Friendster: 13.5GB edge data on 5.8GB GPU), we employ CUDA's Zero-Copy mechanism with pinned host memory. The implementation follows a three-step process:

Step 1: Memory Pinning. We register the host-allocated edge array with `cudaHostRegister(..., cudaHostRegisterMapped)`, which locks the physical pages in RAM and creates a device-accessible mapping. This prevents the OS from swapping pages and enables direct GPU access via PCIe.

Step 2: Device Pointer Acquisition. Using `cudaHostGetDevicePointer()`, we obtain a GPU-side pointer to the pinned host memory. This pointer is valid in device code and transparently routes memory accesses through the PCIe bus.

Step 3: Kernel Execution. GPU kernels access the edge array as if it were in VRAM. The CUDA driver automatically handles PCIe transfers, fetching cache lines on-demand. Coalesced memory access patterns (threads in a warp accessing consecutive addresses) are crucial for achieving reasonable bandwidth.

Memory Advise Hints: We use `cudaMemAdvise(..., cudaMemAdviseSetReadMostly)` to inform the driver that the graph structure is read-only, enabling optimizations like caching in GPU L2 without write-back overhead. Additionally, `cudaMemPrefetchAsync()` can pre-load frequently accessed data (e.g., row pointers) into GPU memory before kernel launch.

Performance Characteristics:

- PCIe 4.0 bandwidth: ∼16 GB/s theoretical, ∼12 GB/s effective
- Zero-copy latency: Higher than VRAM (∼400ns vs ∼100ns)
- Bandwidth-bound: Performance limited by data transfer rate, not compute

Our streaming access patterns achieve ∼0.7 GB/s effective bandwidth on the RTX 3060 Laptop GPU.

3.2 Compression Benefits

Delta-Varint compression provides:

- **Bandwidth Reduction**: 14.4 GB → 9.1 GB (37% reduction)
- **Lower PCIe Traffic**: Fewer bytes transferred per traversal
- **Decoding Overhead**: Partially masked by memory latency

The net effect is a ∼9% speedup for Afforest (21.6s vs 23.5s) and ∼2.6x for BFS (4.5s vs 12s).

4 Experimental Results

4.1 Test Environment

- **GPU**: NVIDIA GeForce RTX 3060 Laptop (Ampere, sm_86)
- **VRAM**: 5.8 GB

- **System RAM**: 32 GB
- **CUDA**: Version 12.5
- **Compiler**: nvcc with -O3, -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
- Compressed size: 9.13 GB (1.48x ratio)

4.3 Performance Results

Algorithm	Mode	Time (ms)	Speedup
Adaptive BFS	Uncompressed	12,009	1.0x
Adaptive BFS	**Compressed**	**4,572**	**2.62x**
Afforest	Uncompressed	23,512	1.0x
Afforest	**Compressed**	**21,569**	**1.09x**

Table 1: Performance on Friendster (Original Graph Order). All runs use Zero-Copy memory for graphs exceeding VRAM.

Metric	BFS (Compressed)	Afforest (Compressed)
Runtime	4.57 s	21.57 s
Memory (Host)	9.13 GB	9.13 GB
Bandwidth Usage	∼2.0 GB/s	∼0.42 GB/s
Correctness	100% (verified)	1 component (correct)

Table 2: Detailed metrics for compressed algorithms on Friendster.

4.4 Analysis

Compression Effectiveness: The 37% data reduction translates to different speedups for BFS (2.6x) vs Afforest (1.09x). BFS benefits more because:

- BFS performs a single pass with high streaming efficiency
- Afforest's atomic operations (component merging) partially mask bandwidth gains
- Varint decoding overhead is more pronounced for multiple atomic-heavy iterations

Zero-Copy Performance: Despite accessing 9-14 GB of data over PCIe, both algorithms achieve practical runtimes. The key is **sequential streaming**: coalesced reads from the GPU exploit PCIe burst transfers effectively.

GCC Pruning Impact: For Afforest, the GCC pruning optimization provides marginal ($< 10\%$) benefit on Friendster because:

- Pruning converts atomic writes to reads, but both operations traverse the same edges
- The bandwidth bottleneck (reading compressed data) is not eliminated
- Primary benefit: Reduced atomic contention on highly connected components

Single-Pass vs Multi-Pass: Using a single-pass Afforest reduces runtime from 46s (convergence loop) to 21.6s. This works because Friendster has diameter 22: one full edge scan connects $> 99.9\%$ of the graph.

5 Graph Reordering Study

We tested BFS-order reordering (sorting nodes by degree) to improve cache locality. Results:

Algorithm	**Original Order**	**BFS Reordered**
Adaptive BFS (Uncompressed)	12.0 s	**8.6 s** (better)
Adaptive BFS (Compressed)	**4.6 s**	5.3 s (worse)
Afforest (Compressed)	**21.6 s**	28.1 s (worse)

Table 3: Impact of graph reordering on Friendster.

Conclusion: Reordering helps uncompressed BFS (better cache locality) but **hurts compression**. The original crawl-order graph has smaller neighbor deltas (friends of friends are nearby), resulting in better Varint encoding. Reordering scatters IDs, increasing delta magnitudes and compression overhead.

Decision: We use the **original graph order** for the final V4 implementation, as it wins in 3 out of 4 scenarios.

6 Conclusion

This project successfully implemented a production-grade GPU graph processing system achieving state-of-the-art performance on billion-edge graphs. Key achievements:

- **4.5s BFS** on 3.6 billion edges (2.6x faster than uncompressed)
- **21.6s Afforest** finding connected components in single pass
- **37% bandwidth reduction** via Delta-Varint compression
- **Zero-Copy support** enabling processing of 13.5GB graph on 5.8GB GPU
- **Adaptive kernel strategy** balancing load across irregular degree distributions
- **GCC-pruned Union-Find** reducing atomic contention

The project demonstrates that **bandwidth-aware optimizations** are critical for large-scale graph processing on modern GPUs. While compute throughput continues to increase (TFLOPS), memory bandwidth remains the primary bottleneck for graph algorithms. Delta compression, streaming access patterns, and pruning strategies are essential for achieving practical performance on billion-scale datasets.

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