Parallel Graph Algorithms

GPU Acceleration with CUDA: Adaptive BFS & Afforest

Evangelos Moschou
AEM: 10986

Abstract

This report presents a comprehensive study of graph preprocessing techniques to improve performance of GPU-accelerated graph algorithms on billion-scale graphs. The primary focus is the Reverse Cuthill-McKee (RCM) reordering algorithm, which optimizes cache locality by reducing the bandwidth of the adjacency matrix. We implement and evaluate RCM preprocessing alongside other optimizations including Delta-Varint compression, Zero-Copy memory management, and adaptive kernel strategies on two fundamental graph algorithms: Breadth-First Search (BFS) and Afforest for connected components. Experimental results on Friendster (65.6M nodes, 3.6B edges) demonstrate that preprocessing choices significantly impact performance: RCM reordering provides speedups for uncompressed algorithms while traditional graph orderings favor compression-based approaches. This study highlights that effective preprocessing is algorithm-dependent and must be co-designed with runtime optimizations.

1 Introduction

Graph preprocessing is a critical but often overlooked component of high-performance graph analytics. The physical layout of vertices and edges in memory significantly affects cache utilization, TLB efficiency, and memory bandwidth requirements—key factors that determine whether algorithms are compute-bound or memory-bound on modern GPUs.

This project's primary contribution is a systematic study of graph preprocessing techniques, with emphasis on the Reverse Cuthill-McKee (RCM) reordering algorithm, and their impact on GPU-accelerated graph algorithms. While we implement two fundamental algorithms (BFS and Afforest) as case studies, the main objective is to demonstrate that **preprocessing choices are algorithm-dependent** and must be co-designed with runtime optimizations.

We evaluate several preprocessing and optimization strategies:

- **RCM Reordering**: Bandwidth minimization for improved spatial locality
- **Delta-Compressed CSR**: Varint encoding reducing data transfer by 37%
- **Zero-Copy Memory**: Streaming access to graphs exceeding VRAM
- **Adaptive Kernels**: Dynamic load balancing for irregular graphs

Our experimental results reveal a key insight: **RCM reordering and compression are antagonistic.** RCM improves cache locality for uncompressed algorithms but increases compression overhead by scattering neighbor IDs. This trade-off guides preprocessing decisions based on the target algorithm's memory access patterns.

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 $\sim$97%. However, the bandwidth bottleneck (reading compressed edge data) remains, limiting the overall speedup to $\sim$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 $\sim$46s to $\sim$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: $\sim$16 GB/s theoretical, $\sim$12 GB/s effective
- Zero-copy latency: Higher than VRAM ($\sim$400ns vs $\sim$100ns)
- Bandwidth-bound: Performance limited by data transfer rate, not compute

Our streaming access patterns achieve $\sim$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 $\rightarrow$ 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 Preprocessing: RCM Reordering

5.1 Motivation and Algorithm

The Reverse Cuthill-McKee (RCM) algorithm is a graph reordering technique that reduces the **bandwidth** of the adjacency matrix by renumbering vertices to place connected nodes close together in memory. This improves:

- **Cache locality**: Adjacent vertices in BFS trees map to nearby memory addresses
- **TLB hit rate**: Reduced page working set due to spatial clustering
- **Prefetcher efficiency**: Sequential access patterns enable hardware prefetching

RCM Algorithm Overview.

1. Select a peripheral node (low degree, far from graph center) as the starting vertex
2. Perform BFS traversal, visiting nodes level-by-level
3. Within each level, sort nodes by **increasing degree** (Cuthill-McKee)
4. **Reverse** the final ordering (Reverse Cuthill-McKee)

The reversal step places high-degree hub nodes at the beginning of the ordering, which empirically improves performance for many graph algorithms by prioritizing well-connected vertices early in traversals.

5.2 Implementation Details

Our CUDA implementation uses:

- **Parallel BFS**: GPU-based level-synchronous traversal for fast reordering
- **Radix Sort**: Per-level sorting by degree using CUB library primitives
- **Pseudo-Peripheral Search**: Heuristic to find optimal starting vertex

Reordering time for Friendster: $\sim$2.3 seconds (one-time preprocessing cost).

5.3 Experimental Results

We compare three graph orderings on Friendster:

1. **Original (Crawl) Order**: Natural edge insertion order from web crawling
2. **RCM Order**: Bandwidth-minimized via RCM preprocessing
3. **Degree-Sorted Order**: Vertices sorted by descending degree

Algorithm & Mode	Original	RCM	Best Ordering
BFS Uncompressed	12.0 s	**8.6 s**	RCM (-28%)
BFS Compressed	**4.6 s**	5.3 s	Original (+15%)
Afforest Compressed	**21.6 s**	28.1 s	Original (+30%)

Table 3: Performance impact of RCM reordering on Friendster (RTX 3060 Laptop, 5.8GB VRAM).

5.4 Analysis: The Compression-Locality Trade-off

Why RCM Helps Uncompressed Algorithms. RCM places neighbors of a vertex in nearby memory locations. During BFS frontier expansion, accessing neighbors of node u results in clustered memory accesses to neighbors $v_1, v_2, \ldots, v_k$, which map to cache lines that are either already present or prefetchable. The 28% speedup for uncompressed BFS confirms that memory bandwidth and cache efficiency dominate performance for sparse graph algorithms.

Why RCM Hurts Compression: The Community Structure Hypothesis. Social networks like Friendster exhibit strong **community structure** (cliques of friends). The "Original" ordering, derived from a web crawl, likely traverses these communities depth-first or locally, assigning consecutive IDs to members of the same clique. This results in many neighbors having $\Delta \approx 1$.

RCM, by contrast, enforces a strict **global breadth-first layering**. It "smears" local communities across vast global levels. For example, if a community of 50 friends is located $\sim$10 hops from the start node, RCM might assign them IDs scattered across the entire range of Level 10 (which contains millions of nodes).

- **Crawl Order**: Neighbors in clique $\rightarrow$ IDs $\{100, 101, 102\} \rightarrow \Delta = \{1, 1\}$ (Hardware-friendly)
- **RCM Order**: Neighbors in clique $\rightarrow$ IDs $\{1000, 50000, 900000\} \rightarrow \Delta = \{49000, 850000\}$ (Byte-heavy)

This limits RCM's compatibility with Delta compression. To "fix" this, one would need **Community-Aware Reordering** (e.g., Rabbit order or Gorder) which explicitly keeps cliques together, rather than RCM's Bandwidth-Aware Reordering.

5.5 Design Guidelines

Based on our results, we recommend:

- **Use RCM** for: Uncompressed, cache-sensitive algorithms (BFS, SSSP) where memory bandwidth is sufficient.
- **Preserve Natural Order** for: Compressed algorithms on social networks. The natural crawl order implicitly captures community structure better than global geometric reordering.

Key Insight: There is no "universal" optimal preprocessing. RCM optimizes the *global matrix envelope* (bandwidth), while compression requires *local neighbor proximity* (gaps). Ideally, one should use community-preserving reordering for compression, which remains an open area for future optimization in this system.

6 Conclusion

This project provides a comprehensive study of **graph preprocessing and its interaction with GPU optimization strategies**. The central contribution is demonstrating that preprocessing choices (RCM reordering vs. natural ordering) have **algorithm-dependent** performance impacts that must be considered during system design.

Key Findings.

- **RCM reordering provides 28% speedup** for uncompressed BFS by improving cache locality
- **RCM degrades compressed algorithms by 15-30%** due to increased delta magnitudes
- **Preprocessing is not universal**: Optimal ordering depends on whether algorithms are latency-bound or bandwidth-bound
- **Compression and locality are antagonistic**: Gains from one often negate the other

Broader Impact. Beyond the specific algorithms implemented (BFS and Afforest), this work establishes **design principles for graph preprocessing**:

1. Analyze algorithm memory access patterns (random vs. sequential, read-heavy vs. atomic-heavy)
2. Measure sensitivity to cache locality vs. memory bandwidth
3. Select preprocessing strategy that aligns with the dominant bottleneck
4. Consider one-time preprocessing cost vs. repeated execution benefits

Graph preprocessing is not a preprocessing step to be applied blindly—it is a **co-design problem** requiring deep understanding of both algorithm characteristics and hardware constraints. On memory-limited GPUs processing billion-scale graphs, the choice between RCM reordering and compression-friendly orderings can determine whether an algorithm is practical or infeasible.

Future work should explore adaptive preprocessing that selects orderings dynamically based on runtime profiling, as well as hybrid schemes that apply RCM to subgraphs while preserving global clustering properties.

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