Connected Components Detection on GPUs Using CUDA

Gerasimou Dimitrios AEM: 10813

Department of Electrical and Computer Engineering
Aristotle University of Thessaloniki
050: Parallel and Distributed Systems

Thessaloniki, Greece
January 2026

Abstract

Connected components (CC) detection is a fundamental primitive in graph analytics, often serving as a preprocessing step for higher-level algorithms. While shared and distributed-memory parallelism can significantly accelerate CC on CPUs, modern GPUs offer massive parallelism that can further reduce runtime, provided that the algorithm and workload are well matched to the GPU execution model.

This report evaluates several CUDA-based CC implementations on a single GPU, including **thread per vertex**, **warp per row**, **block per row**, and an **Afforest style** variant. Experiments were conducted on *Google Colab* using an *NVIDIA Tesla T4 GPU* across a range of real-world graphs, from small sparse matrices to large social and network traffic graphs. Results show speedups of up to $\sim 5\times$ over a sequential CPU baseline and peak throughputs **exceeding 1.9 billion edges per second** on social network graphs. However, performance is highly workload-dependent: for very small graphs, GPU overhead dominates, while for extremely low-degree large graphs (MAWI), naïve GPU mappings perform poorly. These results highlight both the potential and the limitations of GPU acceleration for connected components.

1 Introduction

A *connected component* of an *undirected graph* is a maximal set of vertices such that each vertex is reachable from any other vertex in the set. Computing connected components is a core operation in graph analytics, with applications in social network analysis, web graph processing, biological networks, and preprocessing for more complex algorithms.

On CPUs, connected components can be efficiently implemented using algorithms such as *label propagation* or *union-find*, and parallelized using shared-memory or distributed-memory approaches. However, for large graphs, CPU-based implementations are often limited by memory bandwidth and irregular memory access patterns. GPUs offer an attractive alternative, providing thousands of lightweight threads and high memory bandwidth, but require careful algorithm design to avoid divergence and excessive synchronization.

This project focuses on a **standalone CUDA implementation** of connected components, exploring multiple kernel mappings to understand how different GPU execution strategies interact with graph structure. The goal is not only to achieve high performance, but also to analyze when GPU acceleration is effective and when it fails.

2 Graph Representation

Graphs are represented using a binary *Compressed Sparse Column (CSC)* format, storing only adjacency information. Numerical values are discarded, as connected components depend solely on graph topology.

```
typedef struct {
        size_t   nrows;   /* Number of rows in the matrix */
        size_t   ncols;   /* Number of columns in the matrix */
        size_t   nnz;     /* Number of non-zero (1) entries */
        uint32_t *rowi;   /* Row indices of non-zero elements (length nnz) */
        uint32_t *colptr; /* Column pointers (length ncols + 1) */
} Matrix;
```

Compressed Sparse Column is well-suited for iterative *Connected Components* algorithms because it allows efficient traversal of each vertex's adjacency list. The same representation is used for both CPU and GPU implementations to ensure fair comparison.

3 CUDA Connected Components Implementations

All GPU implementations follow an atomic **union-find (disjoint-set)** approach. Initially, each vertex is its own parent (canonical representative equal to its global ID). The kernels process edges in parallel and perform atomic

union operations to merge sets, followed by path compression passes to flatten the parent forest. An optional Afforest-style sampling phase performs a small number of preconditioning rounds before the main union-find pass.

The implementation uses **union-by-index** rather than the theoretically optimal union-by-rank heuristic. This design choice trades asymptotic optimality for simpler concurrent execution: union-by-index always links the higher-index root to the lower-index root, ensuring a canonical form without requiring atomic rank updates. This simplification reduces synchronization overhead and is well-suited for GPU parallelism where minimizing atomic operations is critical.

The following CUDA mappings were implemented and evaluated:

3.1 Thread per Vertex

Each CUDA thread is responsible for processing a single vertex and scanning all of its neighbors. This approach is conceptually simple but can suffer from severe load imbalance when vertex degrees vary widely. On sparse graphs with many low-degree vertices, threads perform little useful work, while a few high-degree vertices dominate execution time.

3.2 Warp per Row

In this mapping, a full warp cooperatively processes the adjacency list of a single vertex. Neighbor scanning is distributed across the 32 threads of the warp, reducing per-thread work and improving memory coalescing. This approach is more robust to degree variability and reduces divergence compared to thread-per-vertex.

3.3 Block per Row

A full thread block is assigned to each vertex, allowing even larger adjacency lists to be processed in parallel. This mapping increases parallelism for high-degree vertices but introduces higher overhead and requires careful synchronization within the block. It is most effective for very large graphs with significant per-vertex work.

3.4 Afforest Style

An Afforest-inspired variant is also implemented, aiming to reduce work by quickly contracting large components using a limited number of edges before full propagation. While effective in some cases, its benefits depend strongly on graph structure and connectivity patterns.

4 Experimental Setup

Experiments were conducted on **Google Colab**, using the following configuration:

- **CPU**: Intel Xeon (no model number provided)
- **RAM**: ~12.7 GB
- **GPU**: NVIDIA Tesla T4 (Compute Capability: 7.5, ~14.7 GB VRAM)

Each dataset was executed for multiple trials, with a number of warm-up iterations discarded to avoid initialization effects. Reported runtimes correspond to the mean execution time across trials. Throughput is reported as processed edges per second.

Dataset	Vertices	Edges	Avg Degree	Characteristics
dictionary28	52.7k	178k	3.4	Small, sparse graph
hollywood-2009	1.14M	113M	99.0	Medium social graph
com-LiveJournal	4.00M	69.4M	17.3	Social Network
com-Orkut	3.07M	234M	76.3	Dense social graph
mawi_201512020330	226M	480M	2.1	Extremely low-degree, irregular

Table 1: Datasets used in evaluation. Average degree = Edges / Vertices.

Dataset	Best GPU	Throughput (GE/s)	Speedup
dictionary28	Warp-per-Row	0.23	1.33×
hollywood-2009	Warp-per-Row	1.95	4.00×
com-LiveJournal	Thread-per-Vertex	1.65	4.93×
com-Orkut	Warp-per-Row	1.79	4.73×
mawi_201512020330	Block-per-Row	0.35	1.65×

Table 2: Best GPU implementation per dataset.

5 Results

5.1 Small Graph: *dictionary28*

For the small *dictionary28* graph, GPU acceleration provides only marginal benefits. All CUDA variants achieve similar runtimes to the CPU sequential baseline, with speedups close to 1× (Figure 1). Kernel launch overhead and memory transfer costs dominate execution, limiting the effectiveness of GPU parallelism for such small workloads. Despite the low absolute performance gains, **warp per row** delivers the best throughput at 228 MEdges/s (Figure 2).

5.2 Medium Graphs: *hollywood-2009*

On *hollywood-2009*, GPU implementations significantly outperform the CPU baseline. **Warp per row** achieves the best performance, reaching **1.95 billion edges per second** and providing a speedup of approximately 4× (Figure 1). **Thread per vertex** performs slightly worse due to load imbalance, while **block per row** introduces additional overhead without clear benefits for this graph size. The high average degree (99.0) provides sufficient parallelism to amortize GPU overheads effectively.

5.3 Social Networks: *LiveJournal* and *Orkut*

Both *LiveJournal* and *Orkut* benefit strongly from GPU acceleration. All CUDA variants outperform the CPU baseline by a wide margin, with **warp per row** and **thread per vertex** achieving the highest throughput. On *Orkut*, throughput exceeds **1.7 billion edges per second**, corresponding to speedups of around 4.7× over CPU execution (Figure 1). These graphs provide enough parallel work per iteration to effectively amortize GPU overhead. Figure 3 shows that these moderately-sized social networks achieve the best speedups relative to graph size.

5.4 Large Low-Degree Graph: *MAWI*

The *MAWI* dataset exhibits dramatically different behavior due to its extremely low average degree (2.1). The **thread per vertex** mapping performs catastrophically poorly, with runtimes an order of magnitude worse than the CPU baseline, achieving only 0.04× speedup (Figure 1). **Warp per row** improves performance substantially, but only **block per row** achieves a modest speedup (1.65×) over CPU execution. Even in this best case, GPU gains are limited, reflecting the insufficient parallelism per vertex and irregular access patterns. Figure 2 clearly shows MAWI as an outlier, with Thread-per-Vertex throughput dropping below 10 MEdges/s on the log scale.

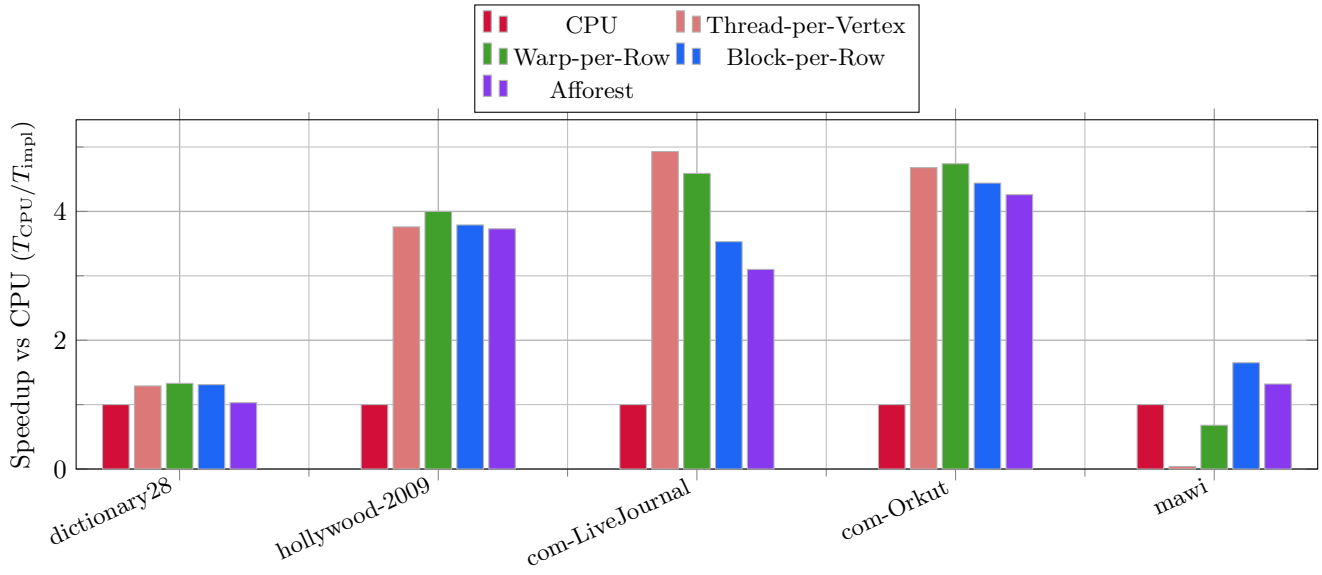

Figure 1: Speedup relative to the sequential CPU baseline (higher is better).

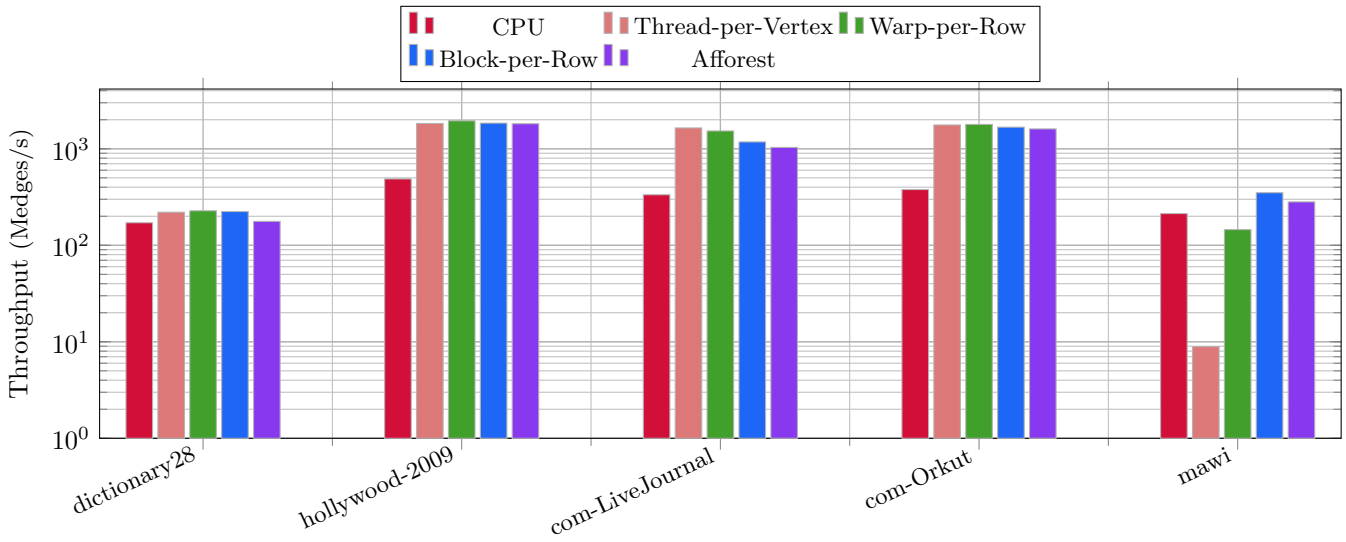

Figure 2: Throughput across datasets (log scale). Social graphs achieve up to $\sim 2 \times 10^3$ Medges/s; MAWI exposes poor mapping choices, with Thread-per-Vertex collapsing below 10 Medges/s.

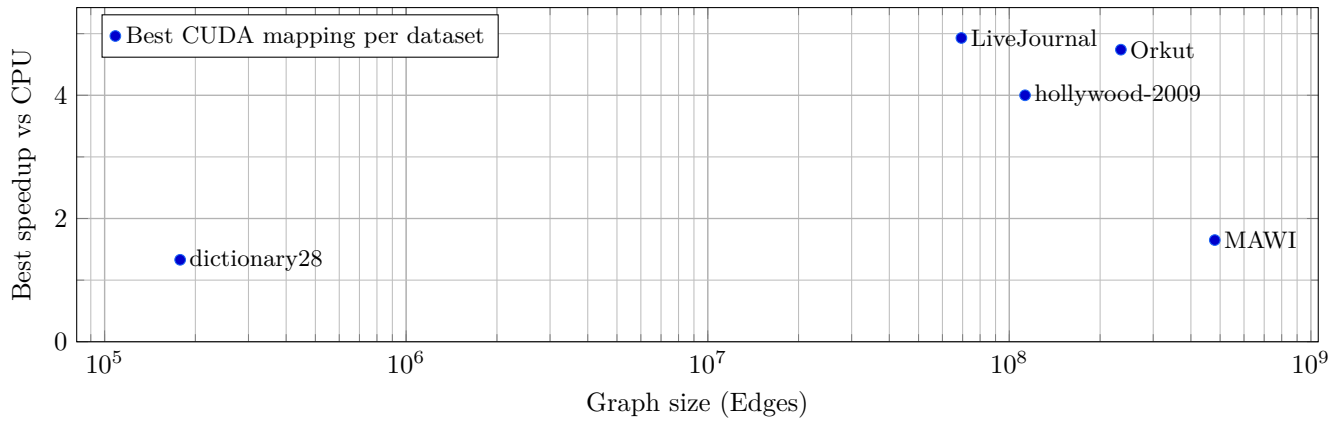

Figure 3: Best observed speedup vs graph size. Small graphs are overhead-dominated; social graphs benefit most; extremely low-degree massive graphs (MAWI) limit GPU gains.

6 Analysis and Discussion

The results demonstrate that **GPU performance for connected components is highly workload-dependent**.

- **Degree distribution matters**: Graphs with higher average degree and more uniform structure (e.g., social networks with avg degree 17–99) benefit the most from GPU parallelism. The correlation between average degree and speedup is evident in Table 1 and Figure 3.
- **Mapping choice is critical**: **Warp per row** consistently provides robust performance across most datasets by balancing parallelism and overhead. **Thread per vertex** is fragile and catastrophically fails on extremely low-degree graphs (MAWI: 0.04× speedup).
- **Overheads dominate small workloads**: For small graphs, GPU kernel launch and synchronization costs outweigh any parallelism benefits, as shown in Figure 1 where dictionary28 achieves minimal speedup.
- **Low-degree massive graphs remain challenging**: *MAWI* highlights the limits of GPU acceleration when computation per vertex is insufficient to hide memory latency and synchronization costs. With an average degree of only 2.1, there is simply not enough parallel work per vertex to justify GPU execution for naïve mappings.

These observations mirror known challenges in GPU graph analytics: maximizing useful work per thread and minimizing divergence are essential for performance.

7 Conclusions

This work evaluated multiple CUDA-based connected components implementations on a single GPU. For medium and large social graphs with moderate-to-high average degree, GPU acceleration achieves substantial speedups over a sequential CPU baseline, with peak throughput **exceeding 1.9 billion edges per second**. However, performance varies widely across datasets and kernel mappings. Simple GPU strategies fail catastrophically on extremely low-degree or small graphs, while more cooperative mappings such as **warp per row** and **block per row** provide better robustness.

Overall, GPUs are a powerful platform for connected components, but achieving high performance requires careful consideration of graph structure—particularly average vertex degree—and execution mapping.

8 Future Work

Potential directions for further improvement include:

- **Convergence analysis**: Instrument the code to track and report the number of iterations required for convergence across different implementations and graph structures. This would help explain performance differences beyond just throughput.
- **Graph reordering**: Implement degree-sorted vertex reordering to improve locality and reduce divergence. High-degree vertices could be processed first to accelerate component merging.
- **Hybrid CPU–GPU approaches**: For irregular or low-degree graphs like MAWI, explore strategies that partition the graph based on degree distribution, processing low-degree vertices on CPU and high-degree vertices on GPU.
- **Multi-GPU scaling**: Investigate graph partitioning strategies for multi-GPU execution with overlap of computation and inter-GPU communication.
- **Memory bandwidth profiling**: Add fine-grained profiling using NVIDIA Nsight Compute to separate computation time, memory transfer time, and synchronization overhead. This would identify bottlenecks more precisely.
- **Alternative algorithms**: Compare against BFS-based connected components and Shiloach-Vishkin algorithms to understand whether union-find is optimal for GPU execution.
- **Dynamic parallelism**: For graphs with highly variable degree distribution, explore CUDA dynamic parallelism to adaptively select thread-per-vertex vs warp-per-row vs block-per-row based on runtime degree analysis.

Code Availability: Source code and benchmark results are available at:
https://github.com/dimgerasimou/pds-hw3-cuda-connected-components.