

# 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 approaching 1 billion edges per second on social network graphs. However, performance is highly workload-dependent: for very small graphs, GPU overhead dominates, while for extremely sparse 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.

```
1 typedef struct {  
2     size_t nrows;      /**< Number of rows in the matrix */  
3     size_t ncols;      /**< Number of columns in the matrix */  
4     size_t nnz;        /**< Number of non-zero (1) entries */  
5     uint32_t *rowi;    /**< Row indices of non-zero elements (length nnz) */  
6     uint32_t *colptr;  /**< Column pointers (length ncols + 1) */  
7 } 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 iterative label-propagation-style approach. Initially, each vertex is assigned a unique label equal to its global ID. During each iteration, vertices inspect their neighbors and update their label to the minimum label observed. Iterations continue until convergence, detected when no label changes.

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	Characteristics
dictionary28	52.7k	178k	Small, sparse graph
hollywood-2009	1.14M	113M	Medium social graph
com-LiveJournal	4.00M	69.4M	Social Network
com-Orkut	3.07M	234M	Dense social graph
maw_201512020330	226M	480M	Extremely sparse, irregular

Table 1: Datasets used in evaluation.

## 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  $1\times$ . Kernel launch overhead and memory transfer costs dominate execution, limiting the effectiveness of GPU parallelism for such small workloads.

### 5.2 Medium Graphs: *hollywood-2009*

On *hollywood-2009*, GPU implementations significantly outperform the CPU baseline. **Warp per row** achieves the best performance, reaching nearly 1 billion edges per second and providing a speedup of approximately  $4\times$ . **Thread per vertex** performs slightly worse due to load imbalance, while **block per row** introduces additional overhead without clear benefits for this graph size.

### 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 880 million edges per second, corresponding to speedups of around  $5 - 6\times$  over CPU execution. These graphs provide enough parallel work per iteration to effectively amortize GPU overhead.

### 5.4 Large Sparse Graph: *MAWI*

The *MAWI* dataset exhibits dramatically different behavior. The **thread per vertex** mapping performs extremely poorly, with runtimes an order of magnitude worse than the CPU baseline. **Warp per row** improves performance substantially, but only **block per row** achieves a modest speedup over CPU execution. Even in this best case, GPU gains are limited, reflecting the low average degree and irregular access patterns of *MAWI*.

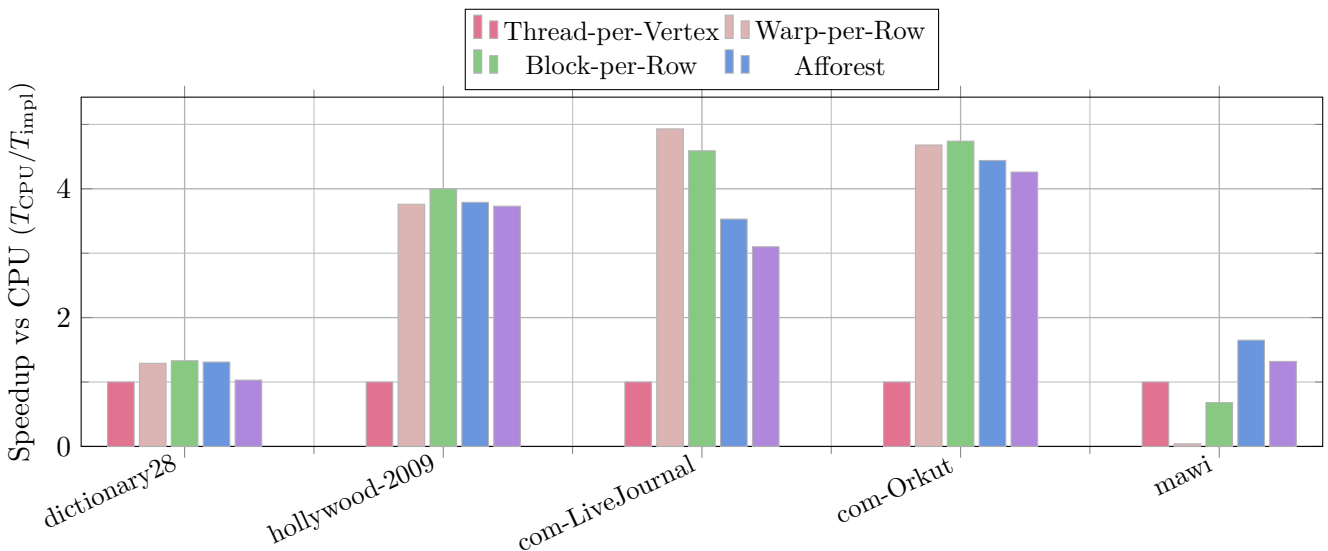


Figure 1: Speedup relative to the sequential CPU baseline (higher is better).

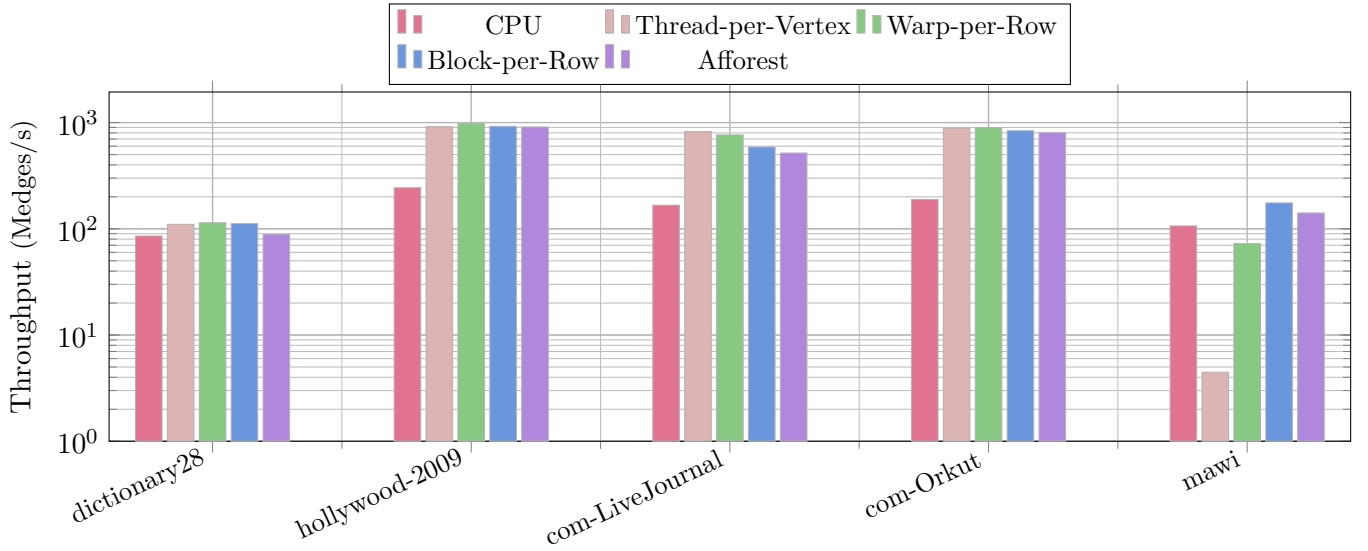


Figure 2: Throughput across datasets (log scale). Social graphs approach  $\sim 10^3$  Medges/s; MAWI exposes poor mapping choices (notably Thread-per-Vertex).

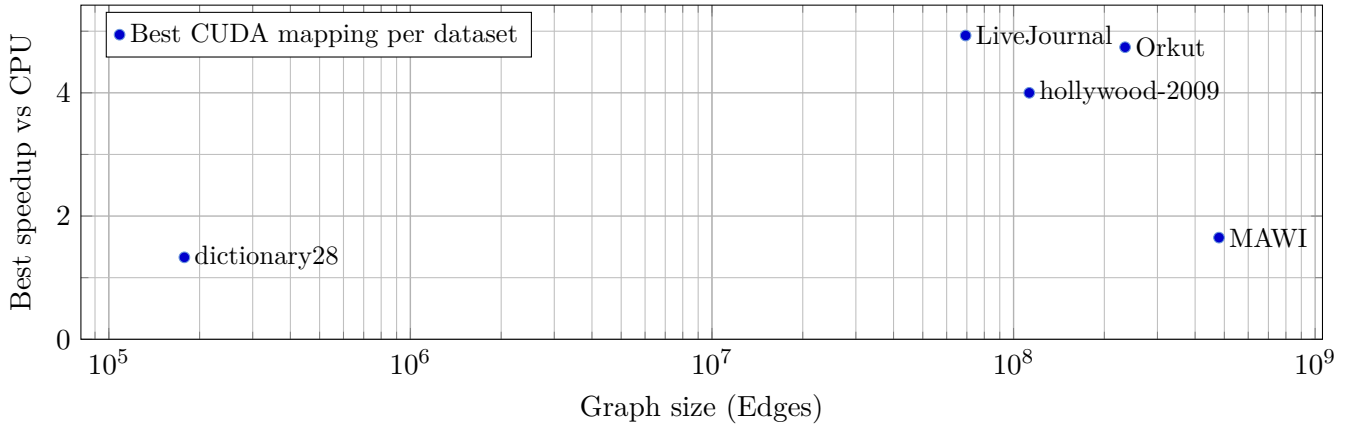


Figure 3: Best observed speedup vs graph size. Small graphs are overhead-dominated; social graphs benefit most; extremely sparse massive graphs (MAWI) limit GPU gains.

## 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) benefit the most from GPU parallelism.
- **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 fails on extremely sparse graphs.
- **Overheads dominate small workloads:** For small graphs, GPU kernel launch and synchronization costs outweigh any parallelism benefits.
- **Sparse massive graphs remain challenging:** *MAWI* highlights the limits of GPU acceleration when computation per vertex is insufficient to hide memory latency and synchronization costs.

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, GPU acceleration achieves substantial speedups over a sequential CPU baseline, with peak throughput approaching 1 billion edges per second. However, performance varies widely across datasets and kernel mappings. Simple GPU strategies fail on extremely sparse 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 and execution mapping.

## 8 Future Work

Potential directions for further improvement include:

- Implementing graph reordering to improve locality and reduce divergence.
- Exploring hybrid CPU–GPU approaches for irregular or sparse graphs.
- Investigating multi-GPU scaling and overlap of computation with data transfers.
- Adding fine-grained profiling to separate computation, memory, and synchronization costs.

**Code Availability:** Source code and benchmark results are available at:

<https://github.com/dimgerasimou/pds-hw3-cuda-connected-components>.