Connected Components of Undirected Graphs (CUDA)

Parallel and Distributed Systems (Homework 3)

Apostolos Pantazis **AM:** 10910 **January 2025**

1 Abstract

This report provides a comparative analysis of two CUDA-based implementations for finding Connected Components (CC) in undirected graphs. We evaluate a baseline model (V1) utilizing a one-thread-per-vertex approach and an optimized model (V2) utilizing a Warp-Centric collaborative model. Our findings highlight the critical impact of memory coalescing and warp divergence, demonstrating that the V2 model achieves up to a 36x speedup on massive sparse graphs like *mawi*. The study concludes with a discussion on the efficiency of asynchronous memory-mapped I/O and parent-linkage optimizations.

2 Introduction

Connected Components (CC) identification is a fundamental primitive in graph analytics. On modern hardware, the challenge lies in scaling these algorithms to handle irregular graphs with billions of edges. While CPU-based parallel solutions like OpenCilk are effective for small to medium graphs, they are bounded by limited core counts and high memory latency.

NVIDIA's CUDA architecture provides a massively parallel alternative. However, naive implementations often fail to utilize the GPU's full potential due to the "irregular" nature of graph data. This report investigates two versions of the algorithm:

1. **V1 (Baseline):** A thread-per-vertex model where each thread independently traverses neighbors.
2. **V2 (Optimized):** A Warp-Centric model where threads within a warp collaborate to process high-degree nodes.

3 Methodology: GPU Architectures

3.1 V1: Naive One-Thread-per-Vertex

In the V1 implementation, a single CUDA thread is mapped to each vertex in the graph. The thread reads its assigned vertex's offset, enters a loop, and scans all neighbors in global memory.

- **Strength:** Simplicity and direct mapping of the sequential logic.
- **Weakness:** Severe **Warp Divergence**. If one thread in a warp is assigned to a "hub" node (millions of edges), the other 31 threads must wait, wasting computational cycles. Furthermore, memory access is uncoalesced as threads in a warp access non-contiguous memory locations.

3.2 V2: Warp-Centric Collaboration

The V2 implementation assigns a **full Warp (32 threads)** to each vertex row. Threads within the warp cooperatively scan the neighbor list, accessing the `edges` array in a linear, coalesced fashion.

- **Communication:** Coordination is handled via `__shfl_down_sync`, allowing threads to find the minimum label across the warp without writing to global memory.

- **Convergence:** Both versions utilize **Parent-Linkage** (hooking) and **Pointer Jumping** (compression), but V2 executes fewer iterations due to the immediate intra-warp propagation of the smallest labels.

```
int warp_id = (blockIdx.x * blockDim.x + threadIdx.x) / 32;
int lane_id = threadIdx.x % 32;
if (warp_id < n) {
    int v = warp_id;
    int min_l = labels[v];
    for (long long k = offsets[v] + lane_id; k < offsets[v+1]; k += 32) {
        int neighbor_l = labels[edges[k]];
        if (neighbor_l < min_l) min_l = neighbor_l;
    }
    // Reduction using register shuffles
    for (int offset = 16; offset > 0; offset /= 2) {
        int remote = __shfl_down_sync(0xFFFFFFFF, min_l, offset);
        if (remote < min_l) min_l = remote;
    }
}
```

Listing 1: V2 Collaborative Warp-Centric Scanning

4 Optimized File I/O

To support billion-edge graphs within a single execution, we optimized the *Matrix Market* parser. Standard `sscanf` was replaced with `mmap()` to map files directly to virtual memory. Our pointer-based parser converts ASCII to integers at near-SSD speeds, allowing the *com-Friendster* dataset (1.8B edges) to load and populate the CSR structure in under 45 seconds locally.

5 Experimental Results

Benchmarking was performed on an NVIDIA RTX 40-series GPU (sm_89). We conducted five trials for each version on two primary datasets: *mawi_201512020330* (226M vertices) and *com-Friendster* (1.8B edges).

5.1 Dataset 1: mawi_201512020330 (226M Vertices, 38M Edges)

The *mawi* dataset highlights the overhead of total vertex count (V) versus edge count (E).

Table 1: mawi_201512020330: V1 vs V2 Comparison

V1 (One Thread / Vertex)			**V2 (Warp-Centric)**		
Trial	**Iterations**	**Kernel Time (s)**	**Trial**	**Iterations**	**Kernel Time (s)**
1	8	59.1026	1	4	1.6496
2	8	62.3235	2	4	1.6203
3	8	62.4350	3	4	1.6215
4	8	60.1701	4	4	1.5984
5	8	60.3158	5	4	1.6139
Average	**8**	**60.8694**	**Average**	**4**	**1.6207**

5.2 Dataset 2: com-Friendster (65.6M Vertices, 1.8B Edges)

The *com-Friendster* dataset serves as a stress test for memory throughput and hub-node management.

Table 2: com-Friendster: V1 vs V2 Comparison

V1 (One Thread / Vertex)			**V2 (Warp-Centric)**		
Trial	**Iterations**	**Kernel Time (s)**	**Trial**	**Iterations**	**Kernel Time (s)**
1	9	9.8580	1	3	3.2530
2	9	8.2668	2	3	3.4452
3	9	8.8390	3	3	4.1613
4	9	9.9399	4	3	4.3464
5	9	9.3123	5	3	3.6154
Average	**9**	**9.2432**	**Average**	**3**	**3.7643**

6 Performance Analysis and Discussion

6.1 The V1 Speed Paradox

A counter-intuitive result observed was that *mawi* (38M edges) took significantly longer than *Friendster* (1.8B edges) in the V1 implementation. This is explained by the **Vertex Scaling Bottleneck**. In V1, the kernel launches one thread per vertex. For *mawi*, the GPU must manage 226 million threads per iteration, compared to 65 million for *Friendster*. Even with inactive node masking, the hardware overhead of scheduling 226M threads—most of which perform no work—creates a massive performance floor.

6.2 Algorithmic Efficiency and Iteration Counts

The V2 implementation achieved convergence in significantly fewer iterations (4 vs 8 for *mawi*). This is due to the **Warp Reduction** and **Parent-Linkage** logic. In V1, labels propagate linearly across edges. In V2, the collaborative scanning and shortcutting flatten the component trees logarithmically. This ensures that even billion-edge components settle into a unified root in 3 to 4 steps.

6.3 Memory Bandwidth Utilization

The 36x speedup on *mawi* (1.6s vs 60s) is primarily a result of **Memory Coalescing**. Because V2 assigns a warp to a row, the 32 threads read the adjacency list in a contiguous block. This allows the GPU memory controller to group requests into a single 128-byte transaction. In V1, each thread accesses a different row, leading to scattered, uncoalesced memory reads that stall the execution pipeline.

7 Conclusion

This study confirms that the naive parallelization of graph algorithms (V1) is insufficient for large-scale, irregular data. By transitioning to a Warp-Centric model (V2) and utilizing hardware-level primitives like shuffles and memory-mapped I/O, we successfully developed a solver capable of processing 1.8 billion edges in under 4 seconds. This represents a significant breakthrough over sequential and fork-join CPU models, providing a highly scalable solution for modern big-data architectures.

All the necessary files as well as build and run instructions can be found in the following GitHub repository: `https://github.com/pantazisa/Homework`