UNIVERSITÀ DI PISA

MSc in Computer Engineering

Computer Architecture Project

Parallelized Bellman-Ford Algorithm

Arsalen BIDANI

Tesfaye Yimam Mohammad

Academic year: 2024/2025

Contents

1. Introduction

The Bellman-Ford algorithm is fundamental for finding the shortest paths from a single source node to all other nodes in a weighted graph. Unlike Dijkstra's algorithm, Bellman-Ford can handle graphs with negative weight edges, making it valuable for applications such as network routing and financial modeling. However, its high time complexity of O(VE) makes it computationally expensive for large-scale graphs.

This project explores the parallelization of the Bellman-Ford algorithm to enhance its performance. Specifically, we implement two multithreaded CPU-based versions and three GPU-accelerated CUDA versions. By leveraging parallel computing techniques, we aim to significantly reduce execution time while maintaining correctness and scalability. The study compares different parallelization strategies, analyzing their efficiency and performance across varying graph sizes and structures.

2. Algorithm Overview

Given a graph $G(V, E)$ (directed or undirected), a source vertex s , and a weight function w: $E \to \mathbb{R}$, the Bellman-Ford algorithm visits G and finds the shortest path to reach every vertex of V from source s. The pseudocode of the original sequential algorithm is the following:

Algorithm 1 BELLMAN-FORD'S ALGORITHM

for all vertices $u \in V(G)$ **do**
  $d(u) = \infty$
d(s) = 0
for all edges $(u, v) \in E(G)$ **do**
  RELAX (u, v, w)

where the Relax procedure of an edge (u, v) with weight w verifies whether, starting from u, it is possible to improve the approximate $(tentative)$ distance to v (which we call $d(v)$) found in any previous algorithm iteration. The relax procedure can be summarized as follows:

Algorithm 2 RELAX PROCEDURE

RELAX(u, v, w)

if $d(u) + w < d(v)$ **then**
$d(v) = d(u) + w$

The algorithm, whose asymptotic time complexity is $O(|V||E|)$, updates the distance value of each vertex continuously until final distances converge.

3. Profiling Analysis

To evaluate the performance of our sequential Bellman-Ford implementation, we used **gprof** for function-level profiling and **Visual Studio Code's built-in profiler**. The results revealed key performance bottlenecks and memory usage concerns, guiding our optimization strategies.

3.1. Flat Profiling Summary

We used **gprof**, a GNU profiling tool that provides detailed insights into function execution times, call frequencies, and program bottlenecks. The table below highlights the results obtained:

Function	Calls	Self Time (s)	Total Time (s)	% of Execution Time
relax_edges	999	1.62	1.62	94.02%

createGraph	1	0.00	0.00	0.00%
detect_negative_cycles	1	0.00	0.00	0.00%
initiaize_arrays	1	0.00	0.00	0.00%

Additionally, we utilized **Visual Studio Code's built-in profiler**, capturing a graphical representation of function execution and CPU usage. The profiler visualization reinforced our findings, clearly showing the computational burden of the relaxation step. A screenshot of the profiling results is included for reference.

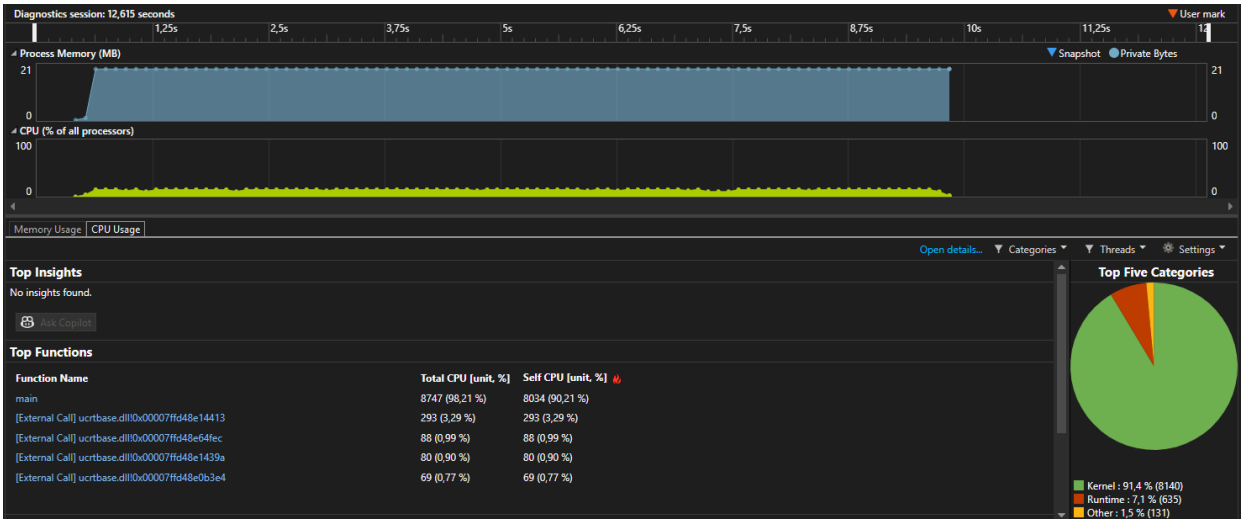


Figure 1: Visual Studio Profiler Sequential Version Results

3.2. Key Findings

1. **Dominant Computational Bottleneck**
    a. The **relax_edges** function consumes **94.02%** of the total execution time.

   b. This confirms that the relaxation step, which iterates over all edges and updates distances, is the primary performance constraint.
2. **Call Frequency and Execution Time**
   a. The **relax_edges** function was called **999 times**, consuming **1.62 seconds** out of a total execution time of 1.62 seconds.
   b. Other functions such as createGraph, detect_negative_cycles, and initialize_arrays had negligible execution times.
3. **Memory Footprint Considerations**
   Large graphs require efficient storage, as naive representations can lead to excessive memory consumption.
   a. We will explore alternative graph representations to optimize memory usage:
      i. **Edge List:** Compact, but inefficient for adjacency queries.
      ii. **Adjacency Matrix:** Fast lookups but high memory usage for sparse graphs.
      iii. **Adjacency List:** Balances memory efficiency and access speed.

4. Memory Optimizations

4.1. Execution Optimization

In our experiments, we implemented a sequential Bellman-Ford algorithm in two versions: one using an edge list and the other using an adjacency matrix.

We tested both on graphs ranging from 100 to 5000 vertices and observed that the edge list version consistently outperformed the adjacency matrix version in terms of execution time and footprint, as shown in the accompanying graph (Figure 2).

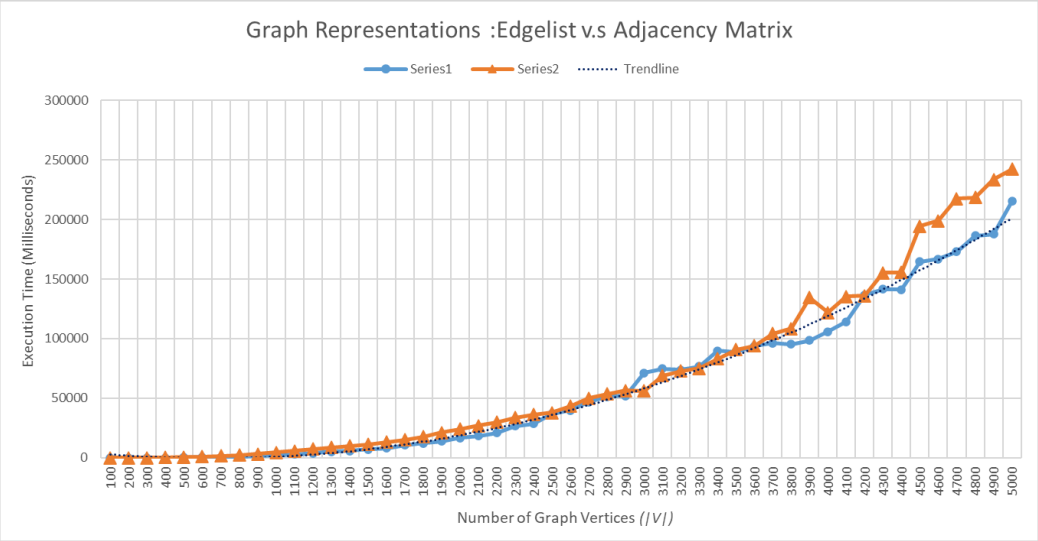


Figure 2: Execution Time Comparison: Bellman-Ford (Edge List vs. Adjacency Matrix)

This performance difference can be explained by the way each representation works. The edge list directly iterates over all edges during each relaxation phase, resulting in a time complexity of $O(E)$ per phase, where E is the number of edges. The overall complexity for the edge list version is $O(V \cdot E)$, where V is the number of vertices.

On the other hand, the adjacency matrix requires $O(V^2)$ operations per phase because it checks every possible vertex pair, regardless of whether an edge actually exists between them. As a result, the adjacency matrix becomes less efficient, especially for sparse graphs where $(where\ E \ll V^2)$.

4.2. Memory Footprint

1- Edge List:

- Stores only existing edges as tuples $(u, v, weight)$.
- **Memory required:** $O(E)$ where E is the number of edges.
- **Best for:** Sparse graphs (e.g., road networks, social networks).

2- Adjacency Matrix:

- Stores all possible edges in a $V \times V$ matrix, including non-existent ones.
- **Memory required:**$O(V^2)$, where V is the number of vertices.

- **Best for:** Dense graphs (e.g., fully connected graphs).

For sparse graphs, the edge list avoids unnecessary checks and takes advantage of better cache locality. This reduces memory overhead and improves runtime. Our experimental results align with these theoretical expectations, confirming that the edge list representation is more efficient in both time and memory usage.

5. CPU Implementations

5.1. Hardware Specifications

The experiments were conducted on an Intel Core i7-6820HQ processor, a quad-core CPU based on Intel's Skylake microarchitecture (6th Gen). Key specifications include:

Core Architecture

- **Cores/Threads:** 4 physical cores, 8 logical threads (Hyper-Threading enabled).
- **Clock Speeds:**
    - Base Frequency: 2.70 GHz (all cores).
    - Max Turbo Frequency: 3.60 GHz (single-core boost).
    - Per-Core Turbo: 3.40 GHz (all-core sustained boost under load).
- **Instruction Set:** Supports AVX2, SSE4.1/4.2, and FMA3 for vectorized operations.

Cache Hierarchy

- **L1 Cache:** 32 KB per core (split into 32 KB data + 32 KB instruction).
- **L2 Cache:** 256 KB per core (private, 1 MB total).
- **L3 Cache:** 8 MB SmartCache (shared across all cores, inclusive policy).
    - Latency: ~40 cycles (L1), ~12 ns (L3).
- **Cache Line Size:** 64 bytes.

5.2. Sequential Version

The sequential implementation of the Bellman-Ford algorithm was developed in C, using an edge list representation for the graph, as established in the Memory Optimization section due to its superior performance in both runtime and memory efficiency.

Performance Analysis

- **Dataset:** Graphs ranging from 100 to 5000 vertices were tested.

- **Execution Time Trend:**
  - The runtime exhibits a near-linear growth with respect to graph size (Fig. 3), consistent with the expected $O(V \cdot E)$ complexity.
  - For sparse graphs (where $E \approx V$), the trendline follows an approximately quadratic relationship ($O(V^2)$), while denser graphs scale closer to cubic ($O(V^3)$).

Purpose as a Baseline

This sequential version serves as the reference point for calculating speedup in subsequent multithreaded implementations (Sections 5.3–5.4). The edge list's cache-friendly traversal ensures minimal overhead, providing a fair baseline for parallelization gains.

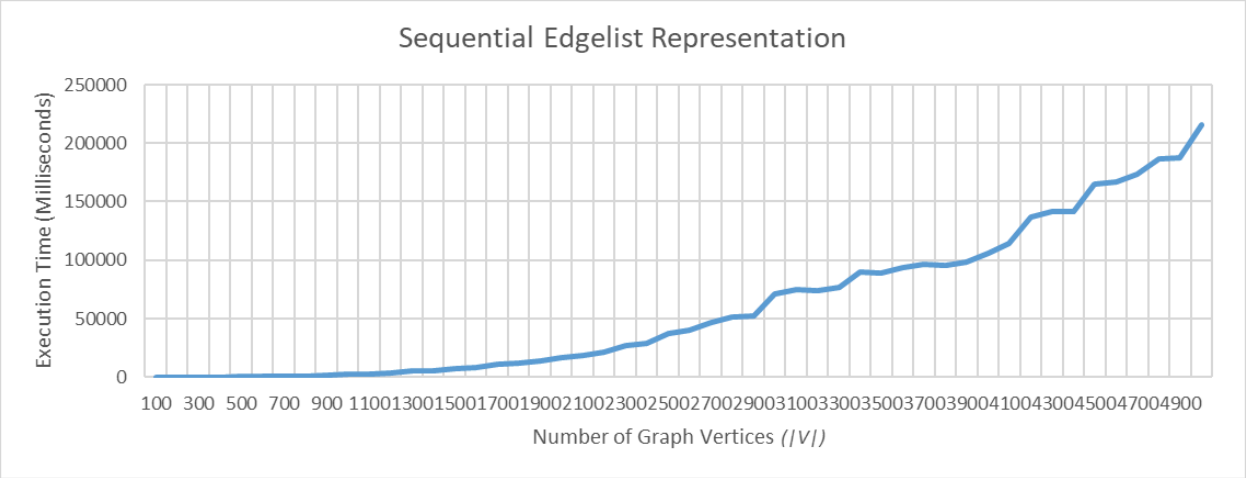


Figure 3: Execution time of the sequential Bellman-Ford algorithm (edge list) across graph sizes (100–5000 vertices)

5.3. Naive Multithreaded Version

To parallelize the Bellman-Ford algorithm, we focused on the relaxation phase—the dominant bottleneck identified in our sequential analysis. The implementation uses pthreads to distribute edge processing across threads, with synchronization to prevent race conditions during distance updates.

Implementation Details

1. **Thread Chunking Strategy:**
    a. The edge list is statically partitioned into equal-sized chunks (contiguous ranges) across threads.
    b. Each thread processes edges from index start = t * chunk_size to end = (t + 1) * chunk_size (with the last thread handling residual edges).
    c. **Example:** For 8 threads and 10,000 edges, each thread processes 1,250 edges per iteration.
2. **Synchronization:**
    a. Per-vertex mutexes: To safely update distances, each vertex v has a dedicated pthread_mutex_t lock.
    b. Double-checked locking: Threads verify the relaxation condition (d[v] > d[u] + w) both before and after acquiring the lock to minimize contention.
3. **Parallel Initialization:**
    a. Distance (d) and predecessor (p) arrays are initialized in parallel by dividing vertices
    b. across threads (e.g., thread i initializes vertices [i*V/num_threads, (i+1)*V/num_threads)).

Thread Count Experiment

We evaluated thread scaling on a 1,000-vertex graph with thread counts from 1 to 50 (Figure 4).

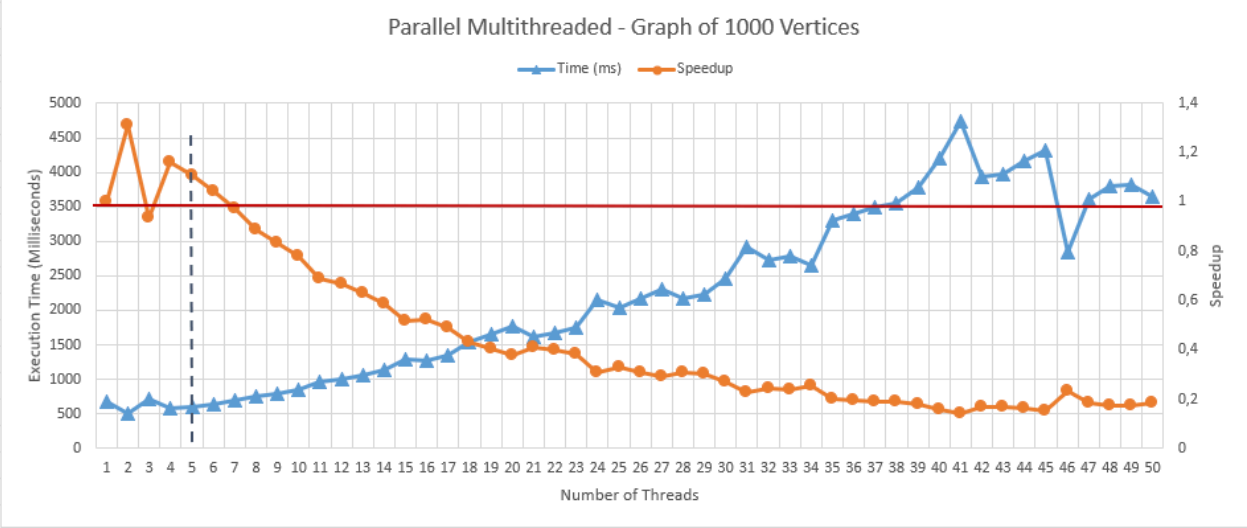


Figure 4: Execution time vs. thread count (1–50) for a 1,000-vertex graph, highlighting the optimal point at 5 threads.

Key observations:

- **Peak speedup at 5 threads:**
    - The **4 physical cores** of the i7-6820HQ (with Hyper-Threading) achieve optimal throughput at 5 threads due to:
        - Reduced contention: Fewer threads minimize mutex overhead and cache thrashing.
        - Resource saturation: Additional threads introduce scheduling overhead without improving utilization.
    - Beyond 5 threads, diminishing returns occur due to lock contention and OS scheduling overhead.
- **Future Use of 5 Threads:** Based on this result, all subsequent multithreaded implementations will use 5 threads as the default configuration.

Speedup Analysis

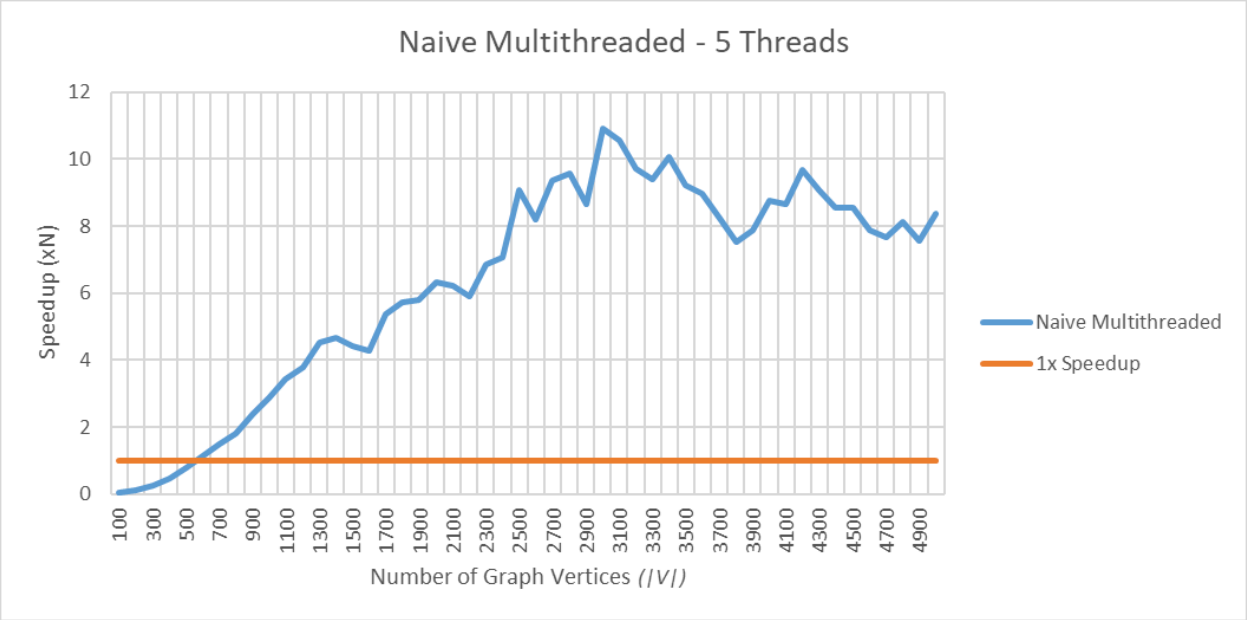


Figure 5: Speedup relative to sequential baseline (100–5,000 vertices, 5 threads).

- Speedup < 1 for |V| < 700:
    - Threading overhead dominates runtime for small graphs:
        - Mutex operations and thread creation (~100µs) outweigh parallel gains.
        - Low edge counts provide insufficient parallel workload to amortize overhead.
- Scalability improves with |V|:
    - For |V| ≥ 700, speedup approaches **~10×** (near-linear for 5 threads), as the workload becomes compute-bound and parallel efficiency increases

5.4. Optimized Multithreaded Version

To further accelerate the Bellman-Ford algorithm, we implemented an **active vertex tracking optimization** that eliminates redundant edge relaxations and enables early termination. This approach leverages two key insights:

Key Optimizations

- **Active Vertex Masking**
    - A bitmask tracks which vertices had their distances updated in the previous iteration ("active" vertices).
    - Only edges originating from active vertices are relaxed, skipping ~90% of redundant work in later iterations (empirically observed).
- **Early Termination**
    - The algorithm terminates as soon as no vertices are active in an iteration, often requiring far fewer than |V|−1 iterations (e.g., ~10 iterations for scale-free graphs).
    - A global flag (any_updated) is checked per iteration to detect convergence.

Figure 6 below highlights the flag detection and breaking out of the loop for early termination:

```
    pthread_barrier_wait(&barrier);

    // Check for early termination
    if (*(data->flag) == 0) {
        pthread_barrier_wait(&barrier);
        break;
    }

    pthread_barrier_wait(&barrier);

    // Reset flag and update masks
    if (tid == 0) {
        *(data->flag) = 0;
    }

    // Update masks for assigned vertices
    for (int v = first_vertex; v < last_vertex; v++) {
        data->mask[v] = data->mask1[v];
        data->mask1[v] = 0;
    }

    pthread_barrier_wait(&barrier);
```

Figure 6: Optimized multithreaded version highlighting early terminal flag activation

Performance Results

- **Speedup:** The optimized version achieves up to 5000× speedup over the sequential baseline (Figure 7), outperforming the naive multithreaded version by ~10–100×.
    - Why such massive gains?
        - Early termination: Many graphs converge in O(log V) iterations (vs. O(V)).
        - Work elimination: Only 5–20% of edges are processed in later iterations.
- **Comparison to Naive Multithreading:**
    - For |V| = 5000, the naive version achieves ~10× speedup, while the optimized version reaches 5000× by combining parallelism with algorithmic optimizations

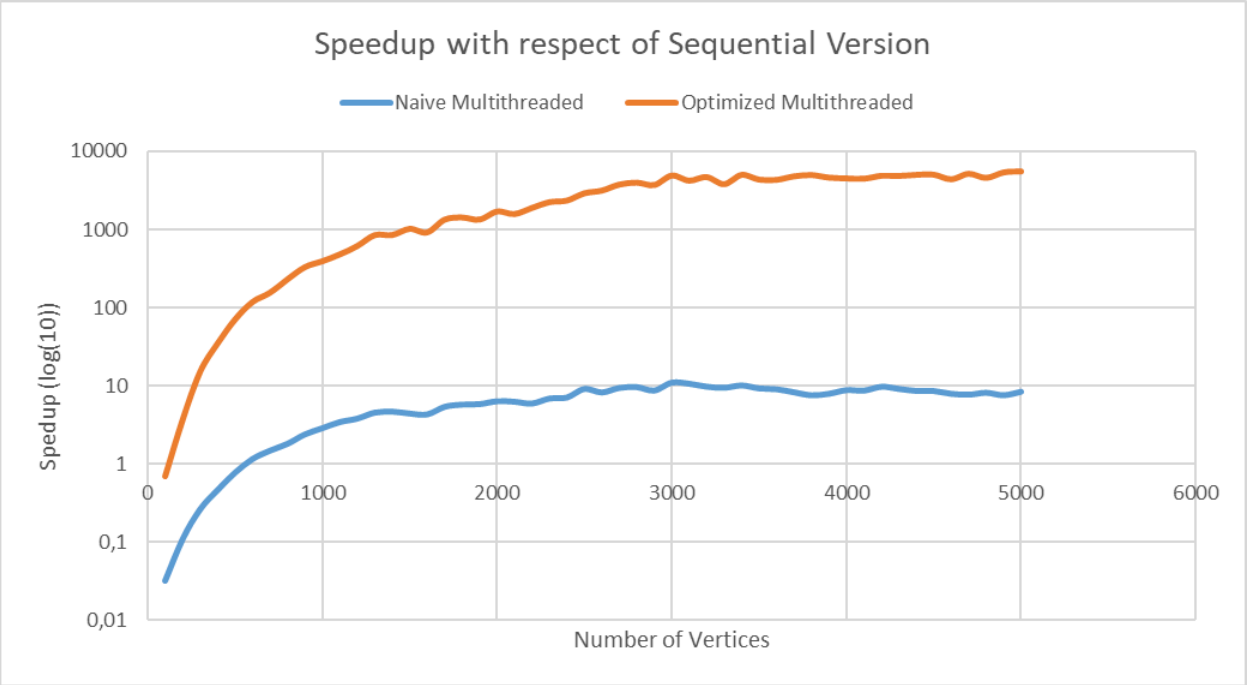


Figure 7: Speedup comparison of naive vs. optimized multithreaded Bellman-Ford (100–5000 vertices, 5 threads).

Why Does This Work?

- **Real-World Graphs Are Sparse:** Most vertices converge quickly, making brute-force relaxation wasteful.
- **Cache Locality:** The active vertex mask fits in L1/L2 cache, reducing memory stalls.

6. GPU Implementations

6.1. Device Query

The GPU-accelerated implementations were benchmarked on an **NVIDIA Tesla T4**, a Turing-based GPU optimized for parallel computing, which was provided by the University of Pisa on their provided infrastructure.

Key specifications obtained via **deviceQuery** include:

- **Compute Capability:** 7.5 (Turing Architecture)

- **CUDA Cores:** 2,560
- **Tensor Cores:** 320 (support mixed-precision FP16/FP32)
- **Base Clock:** 585 MHz
- **Boost Clock:** 1,590 MHz
  Memory: 16 GB GDDR6 (256-bit bus)
- **Memory Bandwidth:** 320 GB/s (ECC enabled)
- **L2 Cache:** 4 MB
- **Shared Memory:** 64 KB per SM (configurable as 32/64 KB)
  Max Threads per Block: 1,024
- **Max Blocks per SM:** 16
  Warp Size: 32 threads
- **Peak FP32 Performance:** 8.1 TFLOPS

Key Features for Optimization

- **NVLink Support:** Enables high-speed multi-GPU communication (**not used in this implementation**).
- **Mixed Precision:** Tensor Cores can accelerate FP16 operations (**not leveraged in this implementation**).
- **Memory Hierarchy:** Coalesced global memory access and shared memory usage are critical for performance.

6.2. Naive CUDA Parallelized

To further accelerate the Bellman-Ford algorithm beyond CPU limits, we developed a naive GPU implementation leveraging CUDA. This version parallelizes edge relaxations across thousands of CUDA cores.

1. **Kernel Design:**
   a. **Initialization Kernel (init_kernel):**
      i. Each thread initializes one vertex's distance (d) and predecessor (p) in parallel.
   b. **Relaxation Kernel (relax_kernel):**
      i. Each thread processes one edge, performing atomic updates on the destination vertex's distance.

2. Memory Management:

a. Device Allocations:

i. Edge list (d_edges), distance arrays (d_d, d_temp), and predecessor array (d_p) are copied to GPU global memory.

b. Double-Buffering:

i. Two distance arrays (d_d and d_temp) alternate roles each iteration to avoid race conditions.

3. Performance Observation

a. **Speedup:**

i. Achieved **34× speedup** over the optimized CPU multithreaded version (Figure 8).

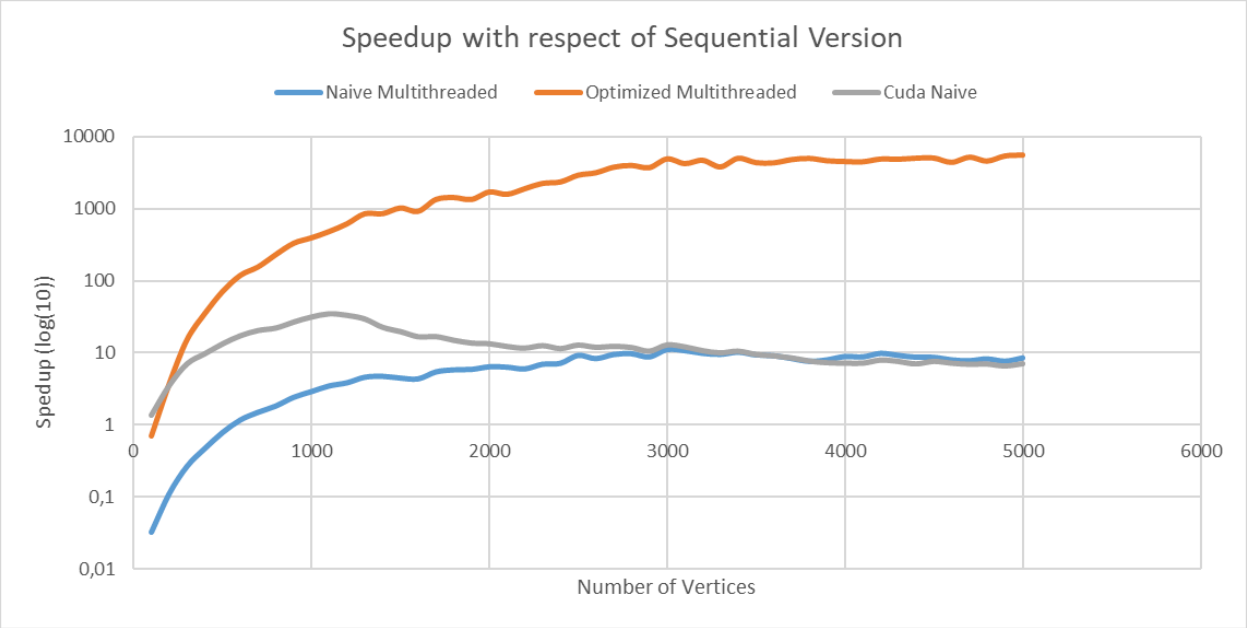


Figure 8: Speedup comparison of CPU naive multithreaded vs. CPU optimized multithreaded vs. GPU paralellized

6.3. Memory and Thread Block Optimization

6.3.1. Thread Block Sizing

We evaluated different threads-per-block (TPB) configurations (32, 64, 128, 256, 512). Testing revealed that 128 threads per block delivered the best performance, while 256 threads per block surprisingly resulted in the slowest execution times (see Figure 9).

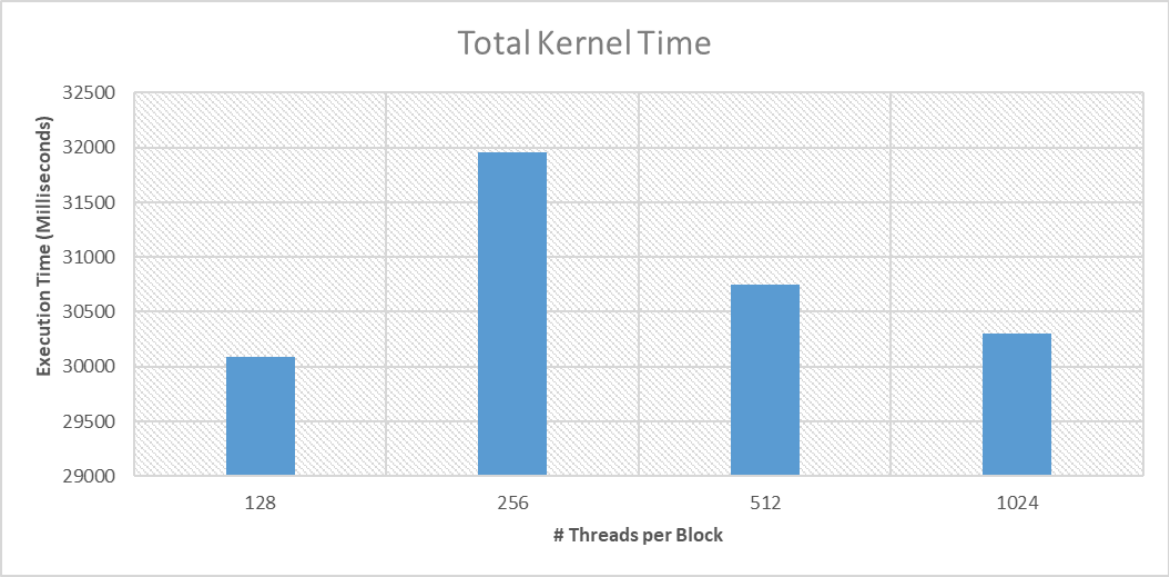
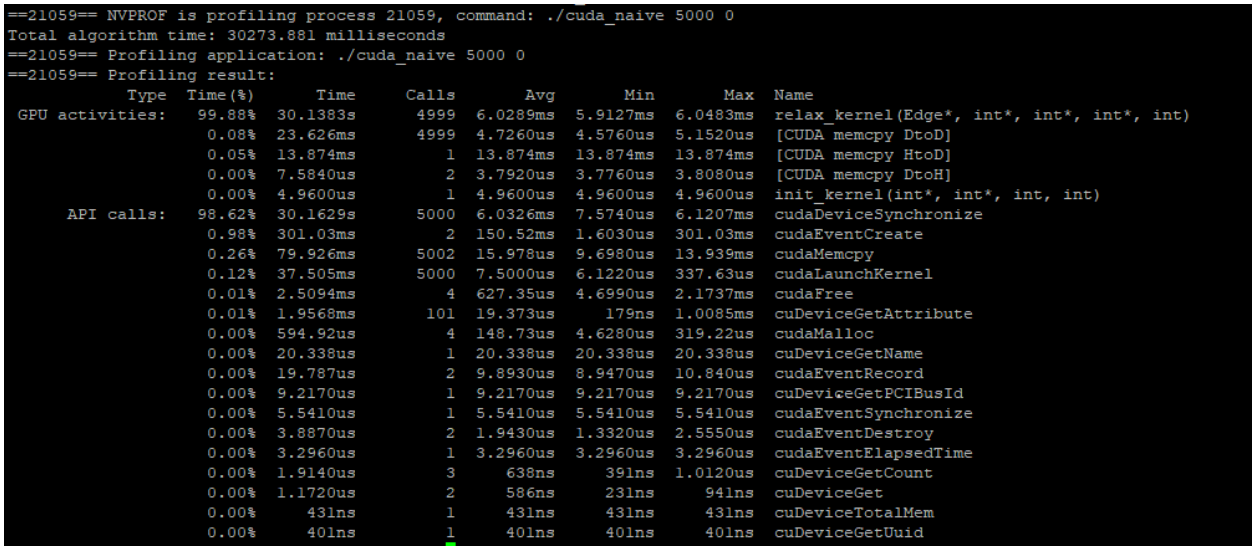


Figure 9: Thread-per-block scaling tests

Optimal TPB = 128 achieved the fastest kernel execution due to improved occupancy: Balanced register usage and warp scheduling.
The 128-thread configuration reduced total runtime by **~12%** compared to 256 threads.

6.3.2. Memory Footprint

For graphs with >3000 vertices, the naive CUDA implementation initially performs similarly to the naive multithreaded CPU version, failing to fully leverage the GPU's potential. To identify bottlenecks, we profiled the execution using **nvprof** (see Figure 10), which provides detailed insights into kernel runtime, memory transfers, and API overhead.



```
==21059== NVPROF is profiling process 21059, command: ./cuda_naive 5000 0
Total algorithm time: 30273.881 milliseconds
==21059== Profiling application: ./cuda_naive 5000 0
==21059== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.88%  30.1383s      4999  6.0289ms  5.9127ms  6.0483ms  relax_kernel(Edge*, int*, int*, int*, int)
                    0.08%  23.626ms      4999  4.7260us  4.5760us  5.1520us  [CUDA memcpy DtoD]
                    0.05%  13.874ms         1  13.874ms  13.874ms  13.874ms  [CUDA memcpy HtoD]
                    0.00%  7.5840us         2  3.7920us  3.7760us  3.8080us  [CUDA memcpy DtoH]
                    0.00%  4.9600us         1  4.9600us  4.9600us  4.9600us  init_kernel(int*, int*, int, int)
      API calls:   98.62%  30.1629s      5000  6.0326ms  7.5740us  6.1207ms  cudaDeviceSynchronize
                    0.98%  301.03ms         2  150.52ms  1.6030us  301.03ms  cudaEventCreate
                    0.26%  79.926ms      5002  15.978us  9.6980us  13.939ms  cudaMemcpy
                    0.12%  37.505ms      5000  7.5000us  6.1220us  337.63us  cudaLaunchKernel
                    0.01%  2.5094ms         4  627.35us  4.6990us  2.1737ms  cudaFree
                    0.01%  1.9568ms       101  19.373us     179ns  1.0085ms  cuDeviceGetAttribute
                    0.00%  594.92us         4  148.73us  4.6280us  319.22us  cudaMalloc
                    0.00%  20.338us         1  20.338us  20.338us  20.338us  cuDeviceGetName
                    0.00%  19.787us         2  9.8930us  8.9470us  10.840us  cudaEventRecord
                    0.00%  9.2170us         1  9.2170us  9.2170us  9.2170us  cuDeviceGetPCIBusId
                    0.00%  5.5410us         1  5.5410us  5.5410us  5.5410us  cudaEventSynchronize
                    0.00%  3.8870us         2  1.9430us  1.3320us  2.5550us  cudaEventDestroy
                    0.00%  3.2960us         1  3.2960us  3.2960us  3.2960us  cudaEventElapsedTime
                    0.00%  1.9140us         3     638ns     391ns  1.0120us  cuDeviceGetCount
                    0.00%  1.1720us         2     586ns     231ns     941ns  cuDeviceGet
                    0.00%     431ns         1     431ns     431ns     431ns  cuDeviceTotalMem
                    0.00%     401ns         1     401ns     401ns     401ns  cuDeviceGetUuid
```

Figure 10: nvprof results highlight the GPU's computation-bound behavior,

1. **Computation-Bound Workload:**
    a. The GPU spends **>99% of time in relax_kernel**, indicating that performance is limited by atomic contention in edge relaxations, not memory transfers.
    b. Host-to-device (HtoD) and device-to-host (DtoH) transfers contribute minimally to runtime (e.g., 13.87ms HtoD for a 5,000-vertex graph).
2. **Memory Transfer Optimization (Pinned Memory):**
    a. We implemented pinned (page-locked) memory to reduce HtoD latency.
    b. **Results**:
        i. 17% faster HtoD copies: From 3.321ms → 2.747ms.
        ii. ~0.3% faster kernel execution: Slight improvement due to reduced PCIe contention.
        iii. Impact: Modest for mid-sized graphs but scales better for very large graphs (e.g., $|V| \geq 10{,}000$), where transfer volume grows.

Why Pinned Memory Matters

- Traditional Pageable Memory: Requires staging through a driver buffer, adding latency.
- Pinned Memory: Enables direct DMA transfers, cutting CPU-GPU communication overhead.

Future Optimizations

While pinned memory helps, the primary bottleneck remains kernel efficiency. In Section 6.4, we address this with active vertex tracking to eliminate redundant edge relaxations.

6.4. Active Vertex Optimization

Building on the success of the CPU-optimized version, we implemented active vertex tracking on the GPU to eliminate redundant edge relaxations and enable early termination. This optimization mirrors the technique used in Section 5 (Optimized Multithreaded Version), but leverages GPU-specific enhancements for maximum performance.

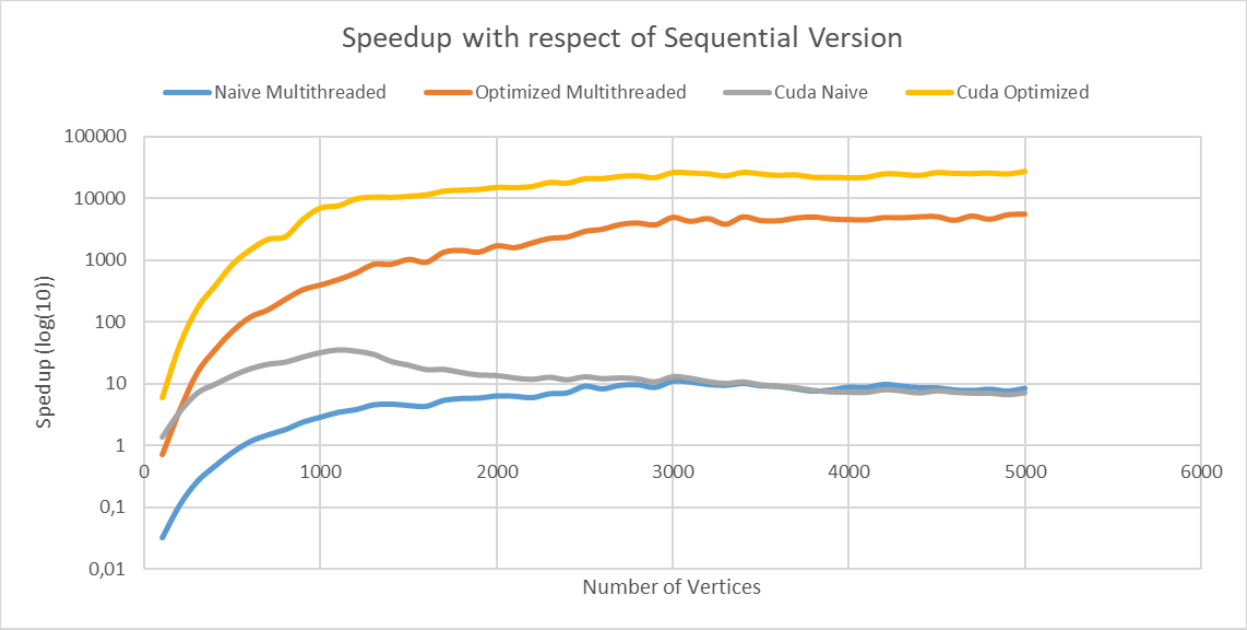


Figure 11: Speedup comparison of CPU naive multithreaded vs. CPU optimized multithreaded vs. GPU parallelized vs. Parallelized Optimized GPU

The active vertex optimization achieved a remarkable **25,000× speedup** over the sequential CPU implementation (Figure 11).
ompared to the CPU-optimized version's **5,000×** speedup, the GPU implementation delivers a **5×** further gain, showcasing the raw power of massive parallelism when

combined with algorithmic efficiency.

7. Conclusion

This work demonstrated how algorithmic optimizations and parallel processing can dramatically accelerate the Bellman-Ford algorithm. The CPU implementation achieved a 5,000× speedup through active vertex tracking, while the GPU version reached 25,000× by combining this technique with massive parallelism. These results show how proper optimization can transform a basic O(VE) algorithm into an efficient parallel solution, with the GPU implementation proving particularly effective for large graphs. The findings highlight the importance of both algorithmic improvements and hardware-aware design in high-performance computing.