Parallel Bellman-Ford Algorithm

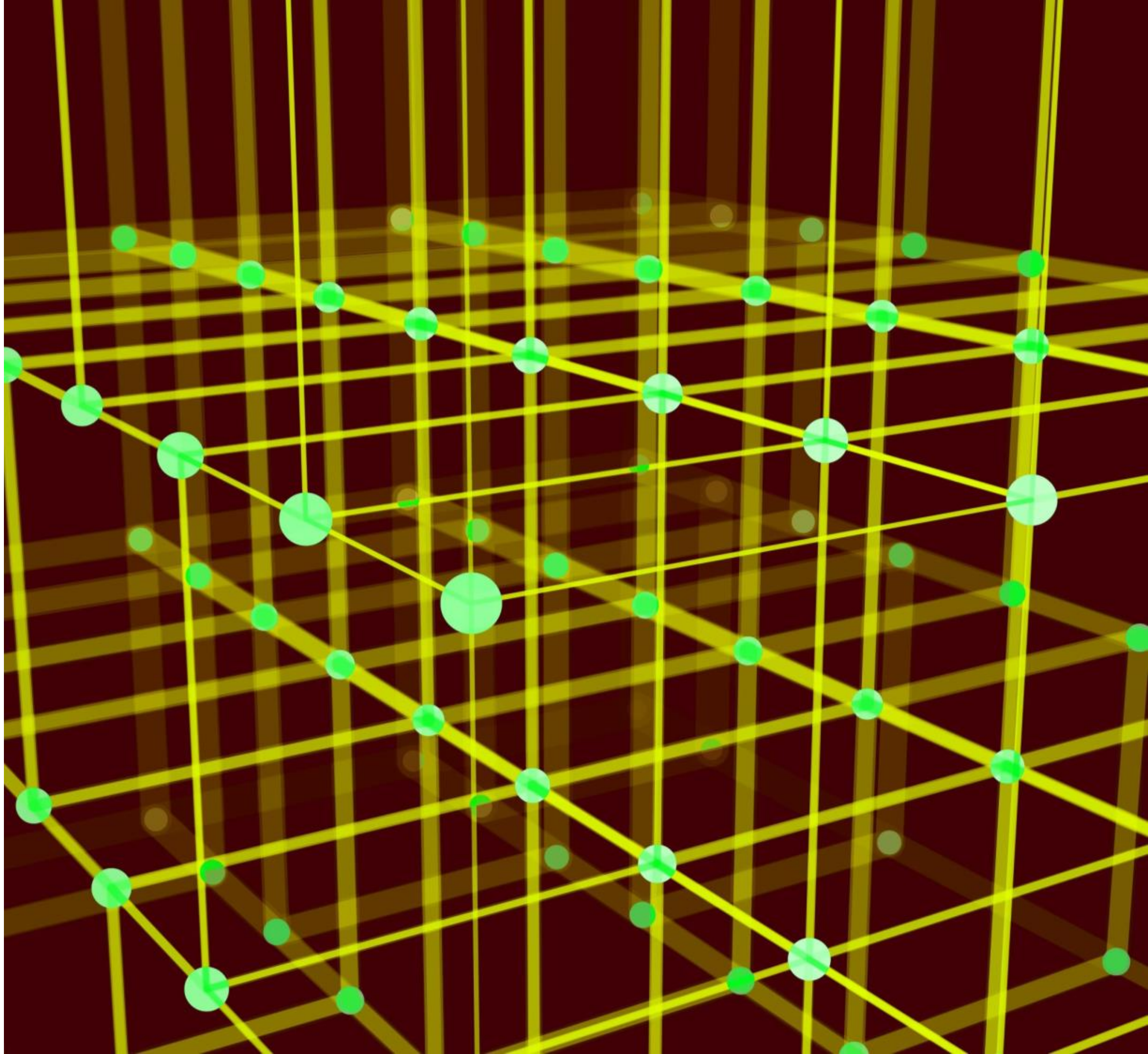
Computer Architecture

Arsalen BIDANI

Tesfaye YIMAM MOHAMED

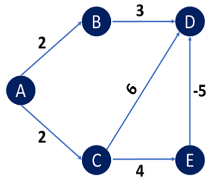
2024/2025

Università di Pisa



Introduction

- Bellman-Ford is a fundamental shortest-path algorithm for weighted graphs.
- Unlike Dijkstra's algorithm, it handles negative weight edges, making it useful for applications like network routing and financial modeling.
- However, its high time complexity $O(V.E)$ makes it inefficient for large graphs.
- This project explores parallelization techniques to enhance Bellman-Ford's performance using CPU multithreading and GPU acceleration.



	B	C	D	E
0	∞	∞	∞	∞
0	2	2	∞	∞
0	2	2	2	6
0	2	2	3	6
0	2	2	3	6

Algorithm Overview

Given a graph $G(V, E)$ (directed or undirected), a source vertex S , and a weight function $w: E \rightarrow \mathbb{R}$ the

Bellman-Ford algorithm visits G and finds the shortest path to reach every vertex of V from source S

Algorithm 1 BELLMAN-FORD'S ALGORITHM

```
for all vertices u ∈ V(G) do
    d(u) = ∞
d(s) = 0
for all edges (u, v) ∈ E(G) do
    RELAX (u, v, w)
```

Algorithm 2 RELAX PROCEDURE

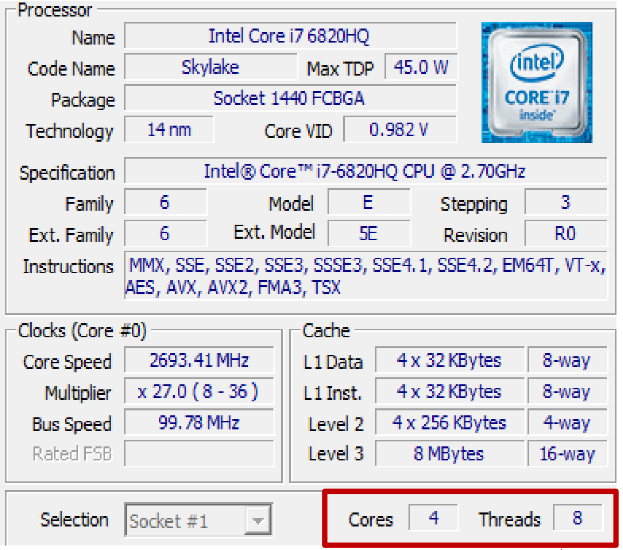
```
RELAX(u, v, w)

if d(u) + w < d(v) then
    d(v) = d(u) + w
```

The worst case scenario time complexity is $O(|V||E|)$.

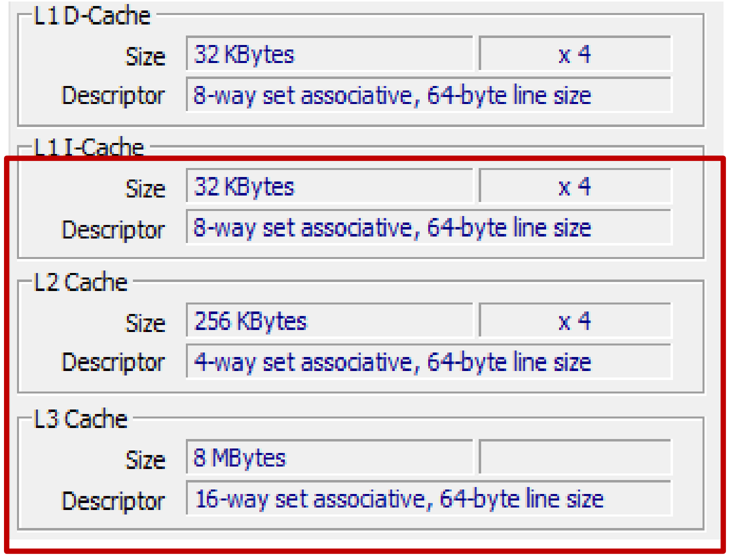
UNIVERSITÀ DI PISA

CPU Implementations

CPU Specifications



Intel® Hyper-Threading Technology

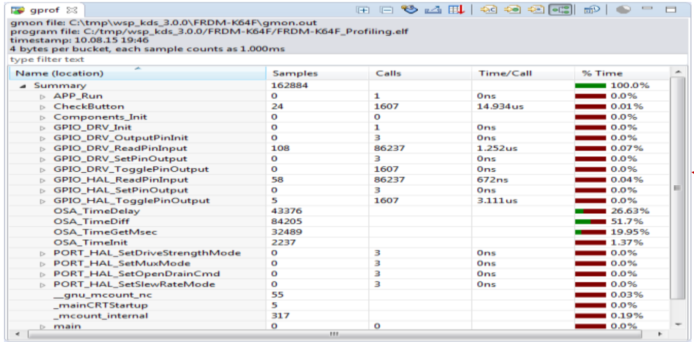


UNIVERSITÀ DI PISA

Profiling Tools

- The **time measurement** method used is the clock() function from the **C standard** library <time.h>. The reason we have chosen it is because it returns the CPU time at the instant it is called, rather than the wall-clock time, which may include small inaccuracies.

```c
// Cross-platform timing
#ifdef _WIN32
#include <windows.h>
long long get_nanoseconds() {
    LARGE_INTEGER freq, counter;
    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&counter);
    return (counter.QuadPart * 1000000000LL) / freq.QuadPart;
}
#else
#include <time.h>
long long get_nanoseconds() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec * 1000000000LL + ts.tv_nsec;
}
#endif
```



gprof

gmon file: C:\tmp\wsp_kds_3.0.0\FRDM-K64F\gmon.out
program file: C:/tmp/wsp_kds_3.0.0/FRDM-K64F/FRDM-K64F_Profiling.elf
timestamp: 10.08.15 19:46
4 bytes per bucket, each sample counts as 1.000ms
type filter text

Name (location)	Samples	Calls	Time/Call	% Time
Summary	162884			100.0%
APP_Run	0	1	0ns	0.0%
CheckButton	24	1607	14.934us	0.01%
Components_Init	0	0		0.0%
GPIO_DRV_Init	0	1	0ns	0.0%
GPIO_DRV_OutputPinInit	0	3	0ns	0.0%
GPIO_DRV_ReadPinInput	108	86237	1.252us	0.07%
GPIO_DRV_SetPinOutput	0	3	0ns	0.0%
GPIO_DRV_TogglePinOutput	0	1607	0ns	0.0%
GPIO_HAL_ReadPinInput	58	86237	672ns	0.04%
GPIO_HAL_SetPinOutput	0	3	0ns	0.0%
GPIO_HAL_TogglePinOutput	5	1607	3.111us	0.0%
OSA_TimeDelay	43376			26.63%
OSA_TimeDiff	84205			51.7%
OSA_TimeGetMsec	32489			19.95%
OSA_TimeInit	2237			1.37%
PORT_HAL_SetDriveStrengthMode	0	3	0ns	0.0%
PORT_HAL_SetMuxMode	0	3	0ns	0.0%
PORT_HAL_SetOpenDrainCmd	0	3	0ns	0.0%
PORT_HAL_SetSlewRateMode	0	3	0ns	0.0%
__gnu_mcount_nc	55			0.03%
_mainCRTStartup	5			0.0%
_mcount_internal	317			0.19%
main	0	0		0.0%

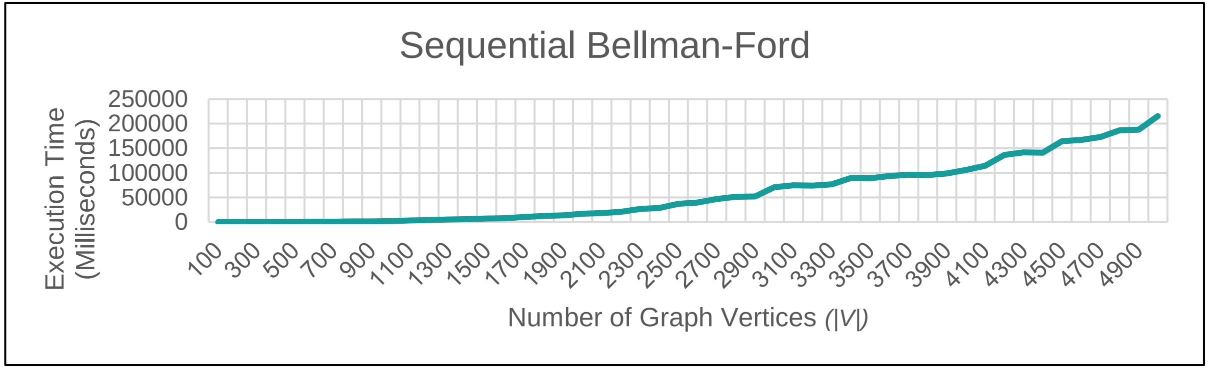
- **Gprof :** It is a performance profiling tool used primarily with programs compiled by GCC. It basically tells us **where a program spends most of its time**.

- **Nvprof :** is a profiling tool for analyzing CUDA application performance on all NVIDIA GPUs, measuring execution time of CUDA kernels, tracking memory usage (global, shared, local), and analyzing warp efficiency, branch divergence, and occupancy.

UNIVERSITÀ DI PISA

Sequential Version Scalability

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 consistent with the expected **O(V·E) complexity**.
- For sparse graphs (where E≈V), the trendline follows an approximately **quadratic relationship (O(V^2))** , while denser graphs scale closer to **cubic (O(V^3))**.

Università di Pisa

Sequential Version Gprof Profiling

We used a **Gprof** on the sequential version of the algorithm on a dataset of 1000 vertices, the results are shown in the table below:

Dominant Computational Bottleneck:

1. $The\ \boldsymbol{relax\_edges}$ function consumes 94.02% of the **total execution time**.
2. The $\boldsymbol{relax\_edges}$ function was called 999 times, consuming 1.62 seconds.
3. Other functions such as $createGraph$, $detect\_negative\_cycles$, and $initialize\_arrays$ had negligible execution times.

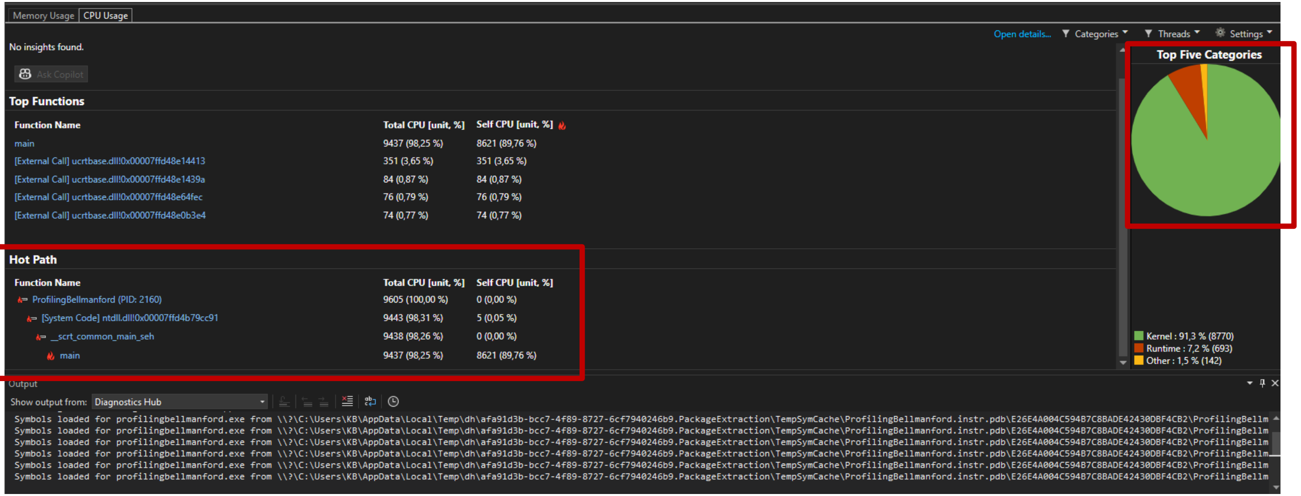
Function	Calls	Self Time (s)	Total Time (s)	% of Execution Time
relax_edges	999	1.62	1.62	94.02%
createGraph	1	0.00	0.00	0.01%
detect_negative_cycles	1	0.00	0.00	0.00%
initiaize_arrays	1	0.00	0.00	0.00%

UNIVERSITÀ DI PISA

Sequential Version VS Profiling

We used the **Visual Studio profiler** to analyze a **1,000-vertex graph**, focusing on the **hot path**. The results reinforced the **computational burden of the relaxation step**. The profiler's visualization clearly highlighted the **intensive calculations** involved in this part of the algorithm, confirming our findings.

Computational Bottleneck



Flamegraph showcasing Hot-Path

Università di Pisa

Strategy & Goals

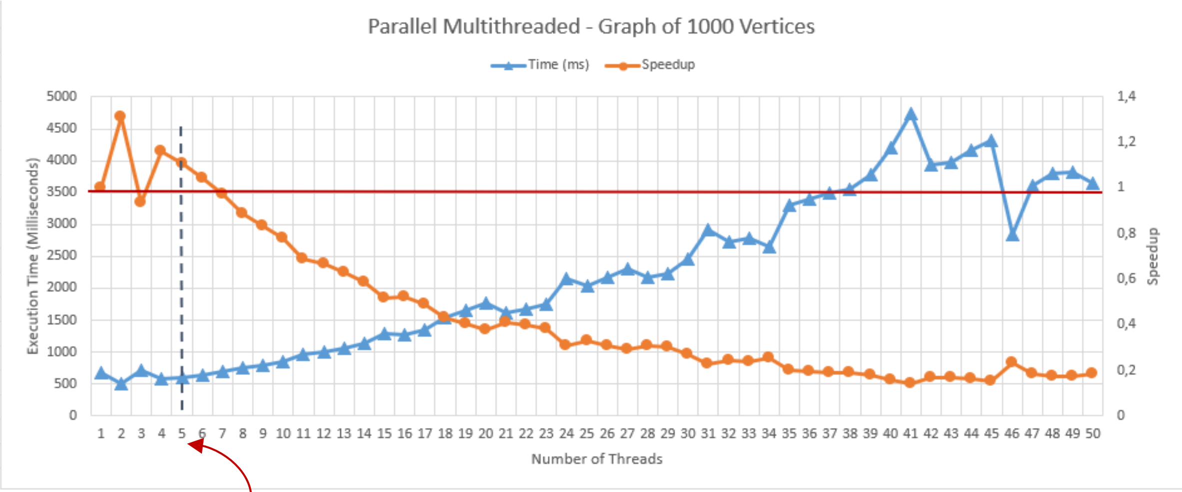
- **Reduce** the practical runtime of the Bellman-Ford algorithm by addressing its inherent $\boldsymbol{O(V.E)}$ complexity .

- **Redesign** the relaxation logic to enable efficient parallel execution

- **Reduce redundant work** by integrating techniques like **active vertex tracking**

Target: Significant runtime speedup up to 1000x through fine-grained parallel processing of edges

Multithreaded Version



The 4 physical cores (with Hyper-Threading) achieved optimal throughput at 5 threads due to minimal contention and full resource utilization.

- 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.

UNIVERSITÀ DI PISA

Multithreaded Scalability Analysis

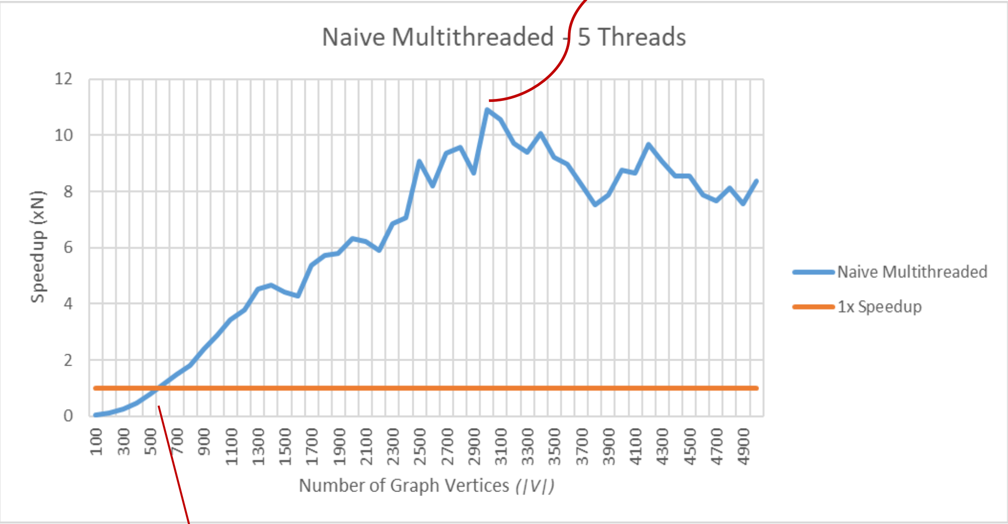
Speedup < 1 for |V| < 700:

- Threading overhead dominates runtime for small graphs
- 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

~10× Speedup



Negative Speedup with reference to sequential version

UNIVERSITÀ DI PISA

Memory Footprint

We have tried with alternative graph representations to optimize memory usage, mainly we have:

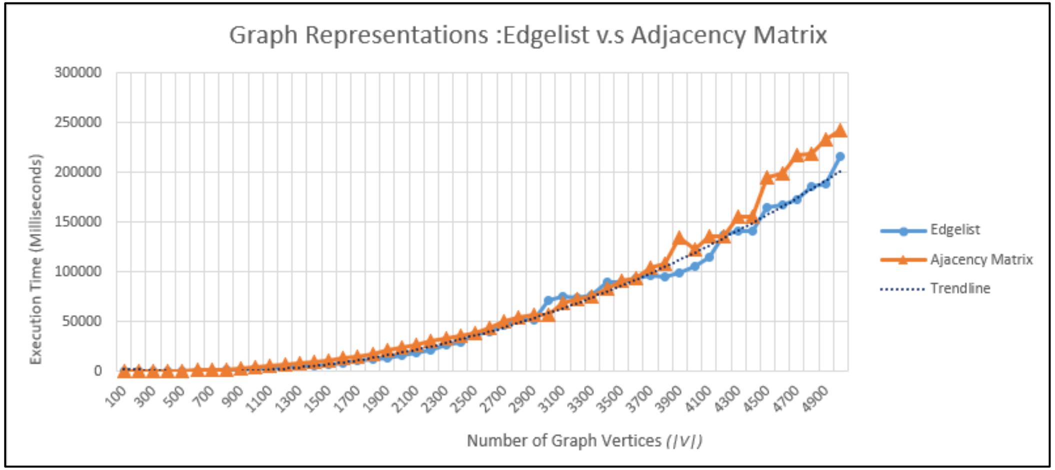
1- Edge List:

- 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 $V.V$ 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).

Memory Footprint (Cont.)



- **Edge list** showed better execution time due to iterating only over actual edges, avoiding unnecessary checks.

- **Adjacency matrix** underperformed on sparse graphs because of its $\boldsymbol{O}(\boldsymbol{V}^2)$ operations per phase, regardless of edge existence.

Università di Pisa

Active Vertex Optimized Multithreaded

To further accelerate the Bellman-Ford algorithm, we implemented an **active vertex tracking** optimization that eliminates redundant edge relaxations and enables **early stopping**.

This approach leverages two key insights:

1. **Active Vertex Masking:**
   - Tracks and relaxes only updated ("active") vertices, skipping ~90% of redundant work.

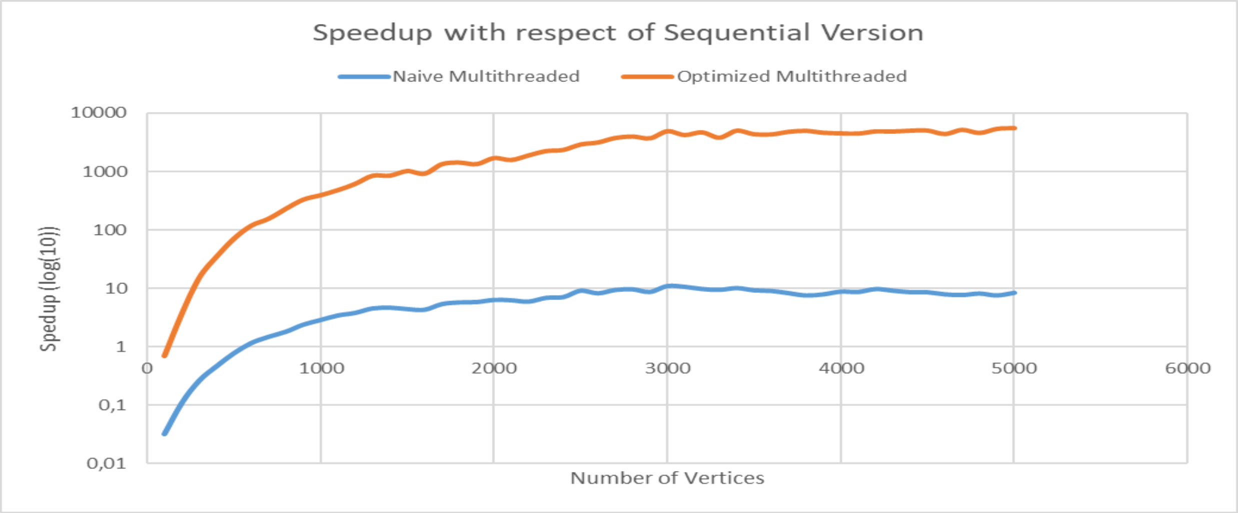
2. **Early Termination:**
   - The algorithm terminates as soon as no vertices are active in an iteration, often requiring far fewer than $|V| - 1$ iterations

```
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);
```

A global flag is checked per iteration

UNIVERSITÀ DI PISA

Active Vertex Optimized Multithreaded (cont.)



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.

- **Speedup:** The optimized version achieves up to 5000× speedup over the sequential baseline 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.

UNIVERSITÀ DI PISA

GPU Implementations

Device Query

Nvidia Tesla T4
on Turing Architecture

```
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Tesla T4"
  CUDA Driver Version / Runtime Version          12.1 / 12.1
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 15984 MBytes (16760700928 bytes
)
  (040) Multiprocessors, (064) CUDA Cores/MP:    2560 CUDA Cores
  GPU Max Clock rate:                            1590 MHz (1.59 GHz)
  Memory Clock rate:                             5001 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 4194304 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536)
, 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 0 / 5
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simu
ltaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 12.1, CUDA Runtime Vers
ion = 12.1, NumDevs = 1
Result = PASS
bidani@computer-architecture:~/cuda-samples/Samples/1_Utilities/deviceQuery$ s
```

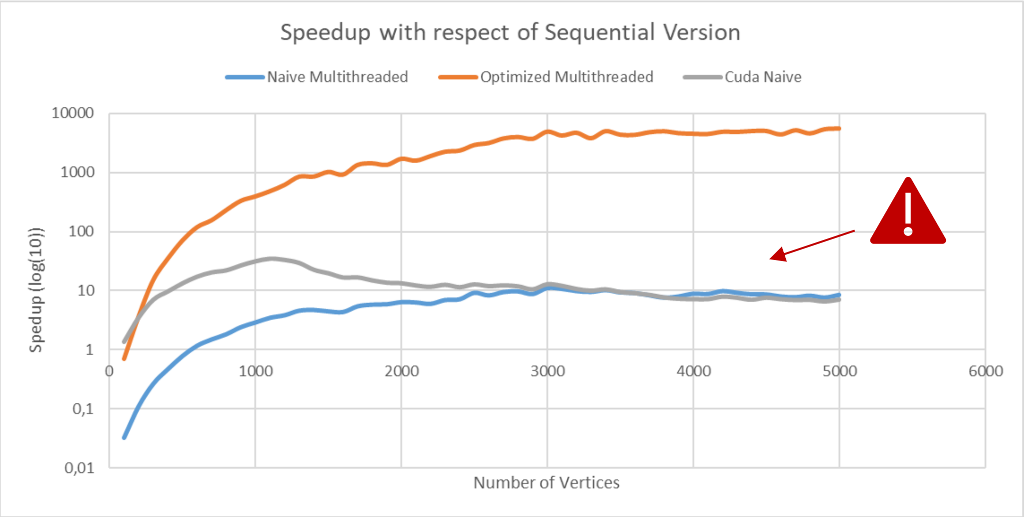
2560 CUDA Cores

Memory and maximum threads per SM and Block

Università di Pisa

Naïve Cuda Version

For the time measurement we used the **Cuda event recorder** modules.



- **Memory Management:**

  - **Double-Buffering:** Two distance arrays (d_d and d_temp) alternate roles each iteration to avoid race conditions.

- **Speedup:**

  - Achieved **34× speedup** over the optimized CPU multithreaded version but then they **equal** over 3000 vertices graph.

We can try running Nvprof to determine bottlenecks

Università di Pisa

Cuda Memory Footprint

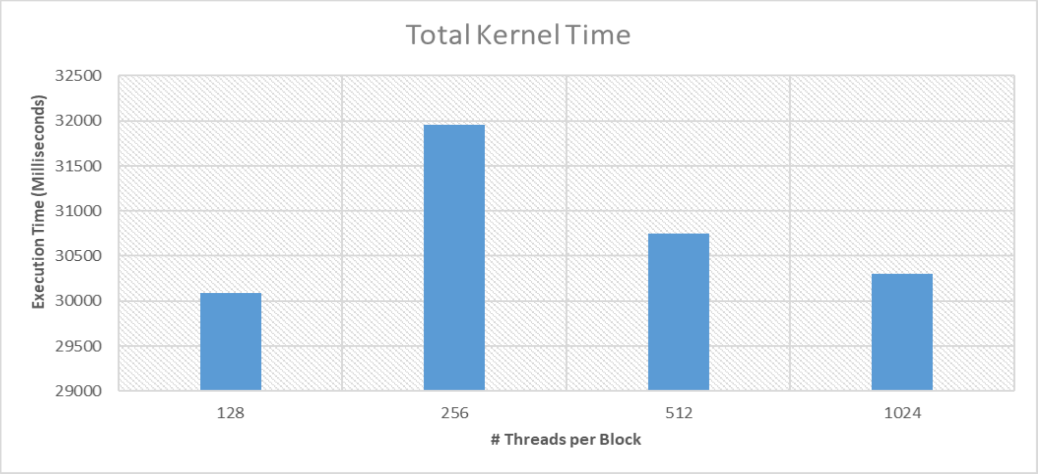
```
==21059== NVPROF is profiling process 21059, command: ./cuda_naive 5000 0
Total algorithm time: 30273.881 milliseconds
==21059== Profiling application: ./cuda_naive 5000 0
==21059== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   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
```

- The GPU spends >99% of time in relax_kernel, indicating that performance is limited by atomic contention in edge relaxations, not memory transfers.
- HtoD) and DtoH transfers contribute minimally to runtime (e.g., 13.87ms HtoD for a 5,000-vertex graph).

The Nvprof report indicates that memory optimization won’t lead us any further, we will try further blocks and threads installements

UNIVERSITÀ DI PISA

Kernel Threads Optimizations



Optimal TPB (Threads Per Block) = 128 achieved the fastest kernel execution due to improved occupancy:

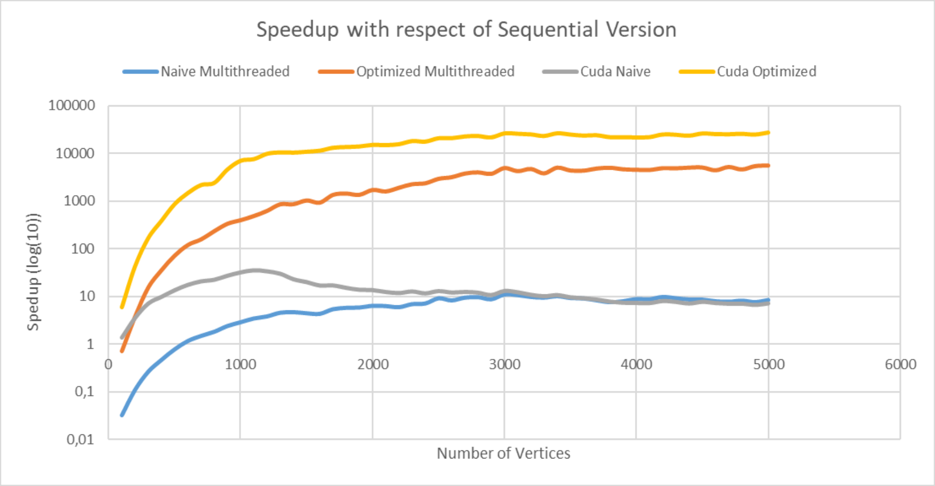
Balanced register usage and warp scheduling. The 128-thread configuration reduced total runtime by **~12%**

compared to 256 threads.

Università di Pisa

Active Vertex Optimizations

Building on the success of the CPU-optimized version, we implemented active vertex tracking on the GPU.



- The active vertex optimization achieved a remarkable **25,000× speedup** over the sequential CPU implementation
- compared 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.

UNIVERSITÀ DI PISA

Conclusion

- **Algorithmic Optimizations & Parallel Processing:** Significantly accelerate the Bellman-Ford algorithm.
- **CPU Implementation:** Achieved **5,000× speedup** using multithreaded **active vertex tracking**.
- **GPU Implementation:** Reached **25,000× speedup** by combining multithreaded **active vertex tracking** with **massive parallelism**.
- **Optimization Impact:** Transformed a basic $\boldsymbol{O}(\boldsymbol{V}.\boldsymbol{E})$ **algorithm** into an **efficient parallel solution**.

Key Takeaway: Algorithmic improvements + **hardware-aware design** = **high-performance computing success**.

UNIVERSITÀ DI PISA