GPU Programming - Vector Addition Analysis

1 Introduction to CUDA

The implementation and performance analysis of vector addition using NVIDIA's CUDA (Compute Unified Device Architecture) is discussed in detail. CUDA programming utilizes a hierarchy of logical components: threads, blocks, and grids, to manage parallel execution. In this model, a thread is the smallest unit of execution. Threads are organized into thread blocks, which are then grouped into a grid. A thread block is assigned to a specific Streaming Multiprocessor. This mapping ensures that all threads within a block can share high-speed data through shared memory and synchronize efficiently. The objective is to evaluate how different configurations of these components, specifically varying the number of blocks and threads, affect the performance of vector addition compared to a traditional sequential CPU baseline.

2 Specifications

2.1 Hardware

2.1.1 GPU Specifications

- GPU: NVIDIA GeForce RTX 4090
- CUDA Cores: 16,384
- Tensor Cores: 512 (4th Generation)
- RT Cores: 128 (3rd Generation)
- Architecture: Ada Lovelace (AD102)
- Compute Capability: 8.9
- Base Clock: 2.23 GHz
- Boost Clock: 2.52 GHz
- Memory: 24 GB GDDR6X
- Memory Bandwidth: 1,008 GB/s
- Maximum Threads per Block: 1024
- Maximum Grid Size (X-dimension): 2,147,483,647
- Streaming Multiprocessors (SMs): 128
- Warp Size: 32 threads
- L2 Cache Size: 75,497,472 bytes (~72 MB)
- Registers per Block: 65,536
- Shared Memory per Block: 49,152 bytes (48 KB)
- Total Constant Memory: 65,536 bytes (64 KB)
- PCIe Generation: Gen4
- Link Width: x16

2.1.2 Memory and Bandwidth

- Memory Capacity: 24 GB GDDR6X.
- Memory Interface Width: 384-bit.
- Memory Bandwidth: 1,008 GB/s (approximately 1 TB/s).
- Memory Speed: 21 Gbps.

2.1.3 CPU Specifications

- Processor: Intel Core i9-14900K

- Architecture: x86_64
- Maximum CPU Frequency: 6.0 GHz
- Total Logical CPUs: 32
- Cores per Socket: 24
- Threads per Core: 2
- System Memory: 62 GB RAM
- Swap: 31 GB
- L3 Cache: 36 MB
- L2 Cache: 32 MB

2.2 Software

- Driver Version: 580.95.05
- CUDA Version: 13.0
- Operating System: Rocky Linux 9.6 (Blue Onyx)
- Kernel Version: Linux 5.14.0-570.49.1.el9_6.x86_64

3 Development and Compilation Process

3.1 Environment Setup

Before compilation, the CUDA development environment was configured by setting the appropriate environment variables in the ~/.bashrc file:

- export CUDA_HOME=/usr/local/cuda
- export PATH=$CUDA_HOME/bin:$PATH
- export LD_LIBRARY_PATH=$CUDA_HOME/lib64:${LD_LIBRARY_PATH}

These environment variables ensure that the CUDA compiler (nvcc) and runtime libraries are accessible from the command line.

3.2 Compilation

After sourcing the updated bashrc file using the command:

- source ~/.bashrc

The program was compiled using the NVIDIA CUDA Compiler:

- nvcc Vector-Addition.cu -o Vector-Addition

This command invokes nvcc, which handles both host (CPU) and device (GPU) code compilation. The -o flag specifies the output executable name. The compiler generates PTX (Parallel Thread Execution) intermediate code and then produces optimized binary code for the target GPU architecture.

3.3 Execution

The compiled program was executed using the following command:

- ./Vector-Addition

The program automatically performs a comprehensive performance analysis across multiple vector sizes and configuration strategies, measuring both inclusive time (including memory transfers) and kernel-only execution time.

4 Implementation Details

4.1 Program Overview

This laboratory exercise implements a fundamental parallel computing operation: vector addition. The program compares the performance of sequential CPU execution against various CUDA GPU parallelization strategies. The

implementation demonstrates the power of parallel computing by distributing the workload across thousands of GPU threads.
The vector addition operation computes C[i] = A[i] + B[i] for all elements i in the vectors. While conceptually simple, this operation provides an excellent benchmark for understanding GPU performance characteristics, memory transfer overhead, and optimal thread configuration.

4.2 CUDA Kernel Implementation

The CUDA kernel for vector addition is designed to exploit massive parallelismEach thread computes its global index using the formula: i = blockIdx.x * blockDim.x + threadIdx.x. This calculation maps each thread to a unique element in the vector. The boundary check (i < n) ensures that threads do not access memory beyond the vector bounds, which is critical when the vector size is not an exact multiple of the block size.

4.3 Sequential CPU Implementation

For comparison purposes, a traditional sequential implementation was developed that processes vector elements one at a time in a single thread. This baseline implementation allows us to quantify the speedup achieved through GPU parallelization.

5 Experimental Setup and Execution Configurations

5.1 Test Methodology

The experimental framework systematically evaluates multiple parallelization strategies across vector sizes ranging from 8 elements (2^3) to 1,048,576 elements (2^{20}). Each configuration measures two critical metrics:

1. Inclusive Time: Total wall-clock time including memory allocation, data transfer between host and device, kernel execution, and result retrieval.
2. Kernel Time: Pure GPU computation time measured using CUDA events, excluding memory operations.

5.2 Configuration Strategies

Flow 0: Sequential CPU Baseline

FLOW: 0.0 - Sequential CPU Baseline

Size	Blocks	Threads	Incl. ms	Kernel ms
8	0	0	0.0018	0.0001
16	0	0	0.0006	0.0001
32	0	0	0.0010	0.0001
64	0	0	0.0018	0.0001
128	0	0	0.0035	0.0002
256	0	0	0.0068	0.0003
512	0	0	0.0136	0.0011
1024	0	0	0.0273	0.0010
2048	0	0	0.0543	0.0040
4096	0	0	0.1101	0.0095
8192	0	0	0.2263	0.0203
16384	0	0	0.4821	0.0435
32768	0	0	0.9764	0.0845
65536	0	0	1.8002	0.1313
131072	0	0	2.9505	0.2282
262144	0	0	5.3923	0.4619
524288	0	0	10.7074	0.8785
1048576	0	0	21.5072	1.3933

Single-threaded CPU implementation serving as a performance baseline. No GPU acceleration is employed, with all computations performed sequentially on a single CPU core.

Flow 1.1: Fixed Single Block, Varying Threads

This configuration uses exactly one thread block with the number of threads equal to min(N, 1024), where N is the vector size. This approach demonstrates the limitations of single-block execution and helps identify the point where additional parallelism is needed.

FLOW: 1.1 - Fixed 1 Block, Varying Threads

Size	Blocks	Threads	Incl. ms	Kernel ms
8	1	8	14.1231	13.9549
16	1	16	0.0626	0.0041
32	1	32	0.0578	0.0031
64	1	64	0.0571	0.0031
128	1	128	0.0554	0.0031
256	1	256	0.0530	0.0031
512	1	512	0.0486	0.0031
1024	1	1024	0.0391	0.0031
2048	1	1024	0.0355	0.0020
4096	1	1024	0.0393	0.0031
8192	1	1024	0.0484	0.0031
16384	1	1024	0.0634	0.0031
32768	1	1024	0.0898	0.0086
65536	1	1024	0.1407	0.0103
131072	1	1024	0.2976	0.0516
262144	1	1024	0.4842	0.0035
524288	1	1024	0.9580	0.0039
1048576	1	1024	1.5851	0.0333

Flow 1.2: Fixed 1024 Threads, Varying Blocks

Each block contains 1024 threads (the maximum for most CUDA architectures), with the number of blocks calculated as ceil(N/1024), capped at 128 blocks. This configuration explores how increasing block count affects performance.

FLOW: 1.2 - Fixed 1024 Threads, Varying Blocks (1 to 128)

Size	Blocks	Threads	Incl. ms	Kernel ms
8	1	1024	0.0425	0.0041
16	1	1024	0.0579	0.0031
32	1	1024	0.0575	0.0020
64	1	1024	0.0568	0.0031
128	1	1024	0.0549	0.0031
256	1	1024	0.0558	0.0031
512	1	1024	0.0483	0.0031
1024	1	1024	0.0362	0.0023
2048	2	1024	0.0349	0.0033
4096	4	1024	0.0380	0.0031
8192	8	1024	0.0436	0.0031
16384	16	1024	0.0532	0.0031
32768	32	1024	0.0668	0.0069
65536	64	1024	0.0997	0.0098
131072	128	1024	0.1661	0.0075
262144	128	1024	0.5190	0.0037
524288	128	1024	0.9029	0.0126
1048576	128	1024	1.3694	0.0057

Flow 1.3: Adaptive Grid (Varying Both)

An adaptive strategy that dynamically adjusts both blocks and threads based on vector size. Thread count is set to min(N, 1024), and block count is calculated as ceil(N/threads). This represents an optimal configuration strategy that scales efficiently with problem size.

FLOW: 1.3 - Varying Both (Adaptive Grid)

Size	Blocks	Threads	Incl. ms	Kernel ms
8	1	8	0.0438	0.0051
16	1	16	0.0577	0.0031
32	1	32	0.0578	0.0031
64	1	64	0.0574	0.0031
128	1	128	0.0549	0.0020
256	1	256	0.0528	0.0020
512	1	512	0.0482	0.0031
1024	1	1024	0.0406	0.0031
2048	2	1024	0.0345	0.0031
4096	4	1024	0.0374	0.0031
8192	8	1024	0.0467	0.0031
16384	16	1024	0.9172	0.0031
32768	32	1024	0.1280	0.0668
65536	64	1024	0.1036	0.0097
131072	128	1024	0.1693	0.0097
262144	256	1024	0.5031	0.0052
524288	512	1024	0.8215	0.0062
1048576	1024	1024	1.3117	0.0275

Flow 2.1: Stress Test - Single Block, Single Thread

An extreme corner case using only one thread in one block, forcing the GPU to process the entire vector sequentially. This configuration demonstrates the worst-case scenario for GPU parallelization.

FLOW: 2.1 - Stress Test - Fixed 1 Block, Fixed 1 Thread

Size	Blocks	Threads	Incl. ms	Kernel ms
8	1	1	0.0412	0.0031
16	1	1	0.0579	0.0031
32	1	1	0.0573	0.0032
64	1	1	0.0567	0.0031
128	1	1	0.0549	0.0031
256	1	1	0.0531	0.0020
512	1	1	0.0480	0.0031
1024	1	1	0.0390	0.0020
2048	1	1	0.0346	0.0031
4096	1	1	0.0374	0.0031
8192	1	1	0.0422	0.0032
16384	1	1	0.0535	0.0031
32768	1	1	0.0667	0.0067
65536	1	1	0.0985	0.0103
131072	1	1	0.2436	0.0073
262144	1	1	0.4911	0.0034
524288	1	1	0.8250	0.0176
1048576	1	1	1.2903	0.0056

Flow 2.2: Stress Test - Exceeding Maximum Threads

This test intentionally violates CUDA constraints by attempting to launch kernels with thread counts exceeding the hardware limit of 1024 threads per block. The configuration demonstrates error handling and validates that the CUDA runtime properly rejects invalid kernel launches.

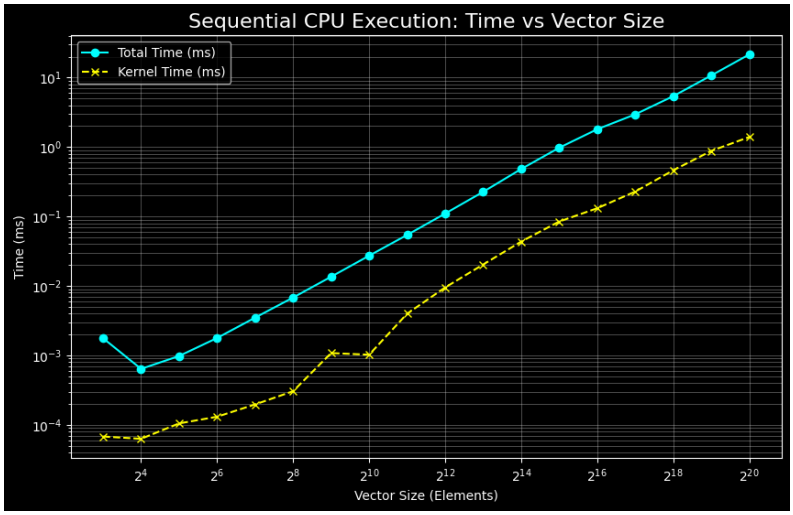
```
============================================================
 FLOW: 2.2 - Fixed 1 Block, Exceeding Maximum threads
------------------------------------------------------------
      Size | Blocks |  Threads |     Incl. ms |    Kernel ms
------------------------------------------------------------
      1024 |      1 |     1024 |       0.0478 |       0.0051
CUDA Error: invalid argument
      2048 |      1 |     2048 |       0.0431 |       0.0094
CUDA Error: invalid argument
      4096 |      1 |     4096 |       0.0368 |       0.0031
CUDA Error: invalid argument
      8192 |      1 |     8192 |       0.0414 |       0.0020
CUDA Error: invalid argument
     16384 |      1 |    16384 |       0.0499 |       0.0010
CUDA Error: invalid argument
     32768 |      1 |    32768 |       0.0618 |       0.0013
CUDA Error: invalid argument
     65536 |      1 |    65536 |       0.0919 |       0.0013
CUDA Error: invalid argument
    131072 |      1 |   131072 |       0.2458 |       0.0013
CUDA Error: invalid argument
    262144 |      1 |   262144 |       0.5099 |       0.0013
CUDA Error: invalid argument
    524288 |      1 |   524288 |       0.8103 |       0.0013
CUDA Error: invalid argument
   1048576 |      1 |  1048576 |       1.2849 |       0.0015
```

6 Results and Analysis

In this section, the behaviour of plots of each execution configuration is discussed in detail.

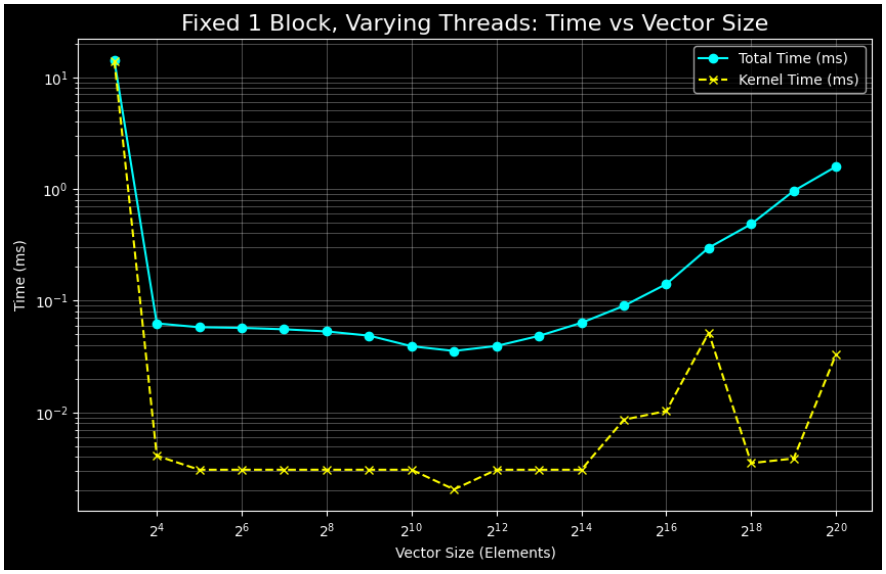
6.1 Sequential CPU Execution

This plot shows a linear relationship between the vector size (N) and execution time. As N grows, the time increases predictably because the CPU processes each element one by one in a single thread. There are small steps or sudden increases in time as the vector size exceeds the CPU's L1, L2, or L3 cache capacities. In sequential execution, drops are rare and usually represent minor fluctuations in background system processes rather than an increase in computational efficiency.



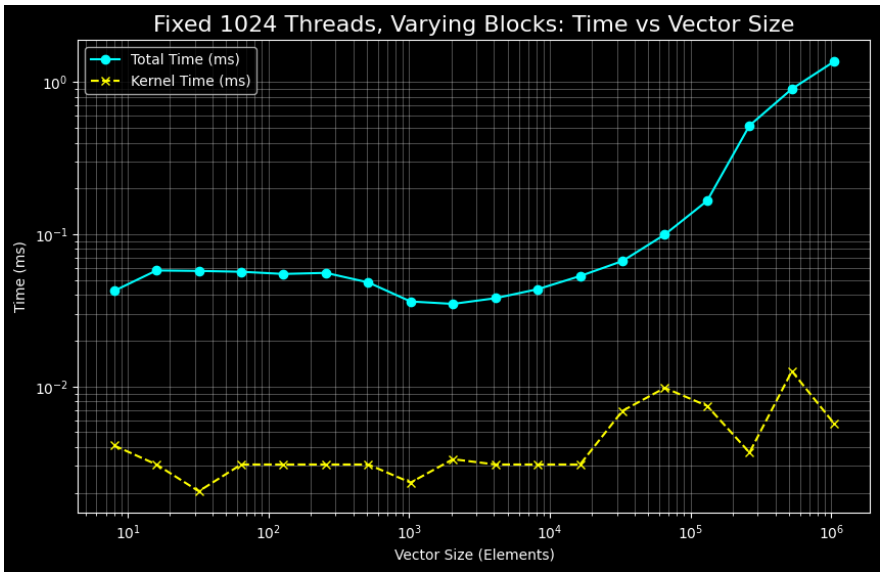
6.2 Varying Threads

Performance improves as threads increase up to 1024, then levels off or degrades for larger vectors. At very small sizes (e.g., 2 power 3 to 2 power 10), execution time drops significantly as you add more threads because the GPU starts utilizing its parallel cores. Once you hit 1024 threads, the block is full. For vectors larger than 1024, a single thread must process multiple elements sequentially.



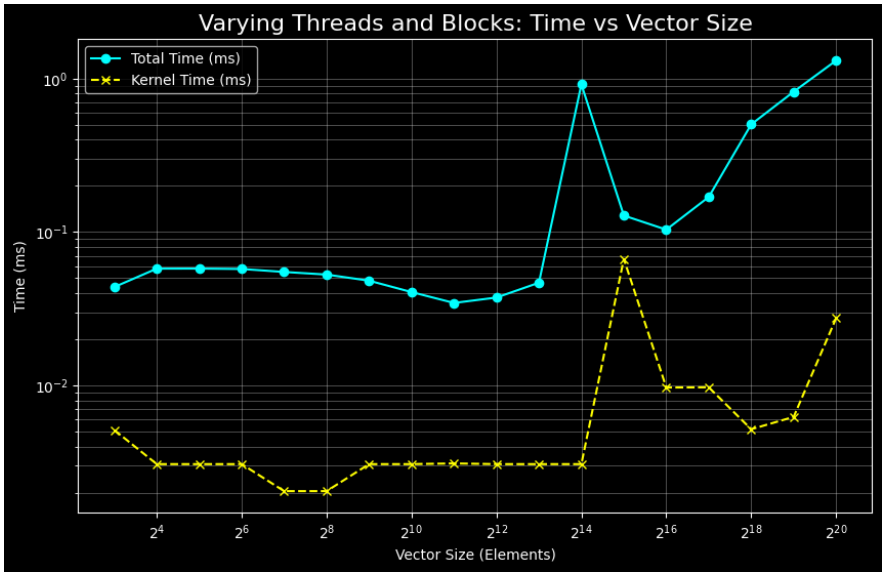
6.3 Varying Blocks

This configuration scales better than Flow 1.1 by increasing the number of blocks (up to 128) to handle larger vectors. There is a performance dip (improvement) around a vector size of 1024 to 2048. At 1024 elements (1 block), the inclusive time is 0.036 ms; at 2048 elements (2 blocks), it is 0.034 ms. This is because the GPU is reaching a balance between workload and its massive core count (16,384 cores). Beyond 128 blocks, the growth becomes linear again because the block count is capped. The Inclusive time climbs to 1.369 ms at 2 power 20 elements.



6.4 Varying Threads and Blocks

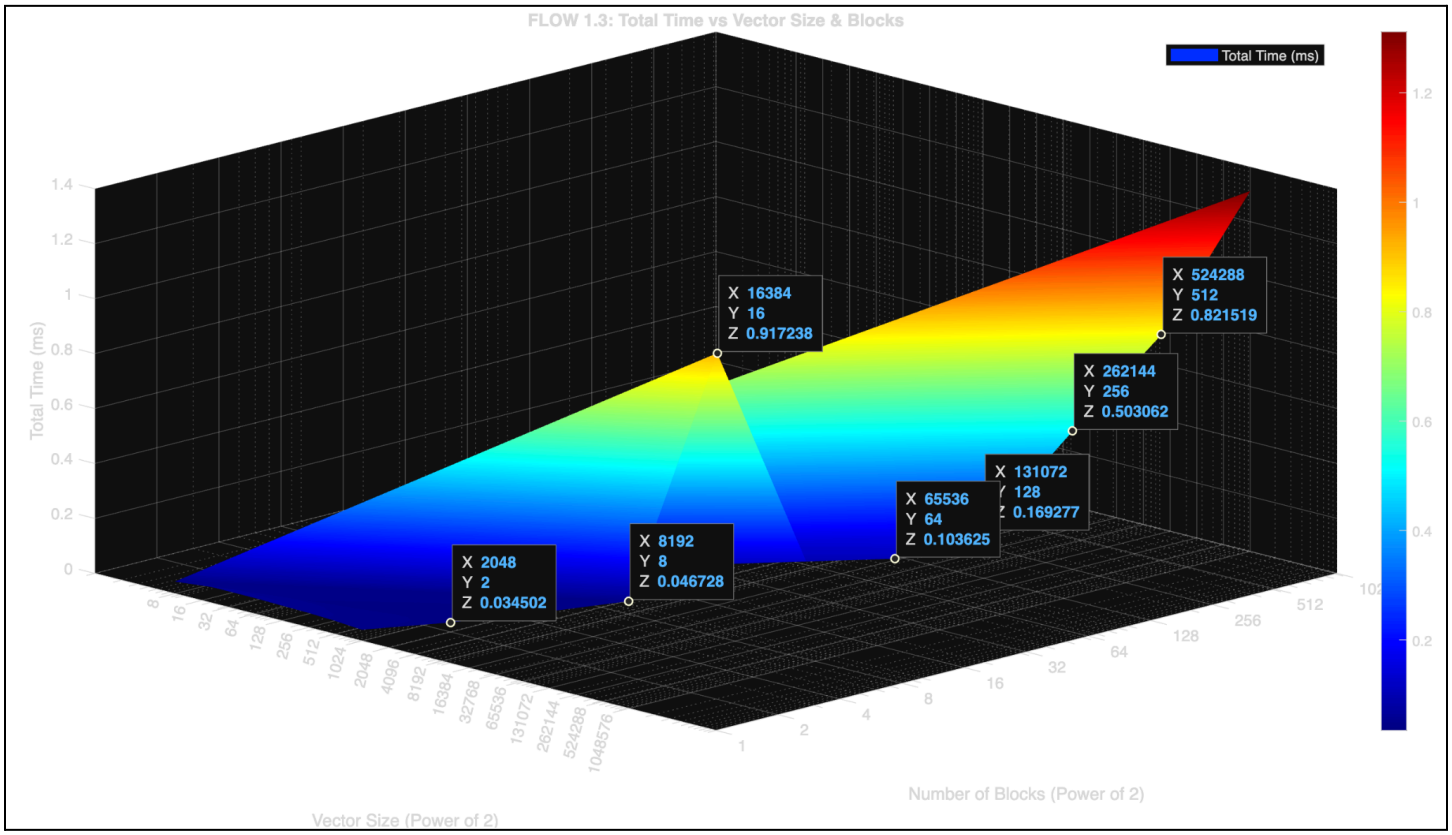
This is the most efficient configuration, dynamically adjusting blocks and threads to match the vector size. There is a sharp peak in the total time at vector size 16,384 (2 power 14), reaching 0.917 ms. This often occurs due to PCIe bus latency or memory allocation overhead. At this specific size, the overhead of moving data from the CPU (Host) to the GPU (Device) outweighs the speed of the computation itself. The time drops back down to 0.128 ms for 2 power 15 as the GPU's high bandwidth (1,008 GB/s) begins to mask the transfer latency through kernel-hide effects.

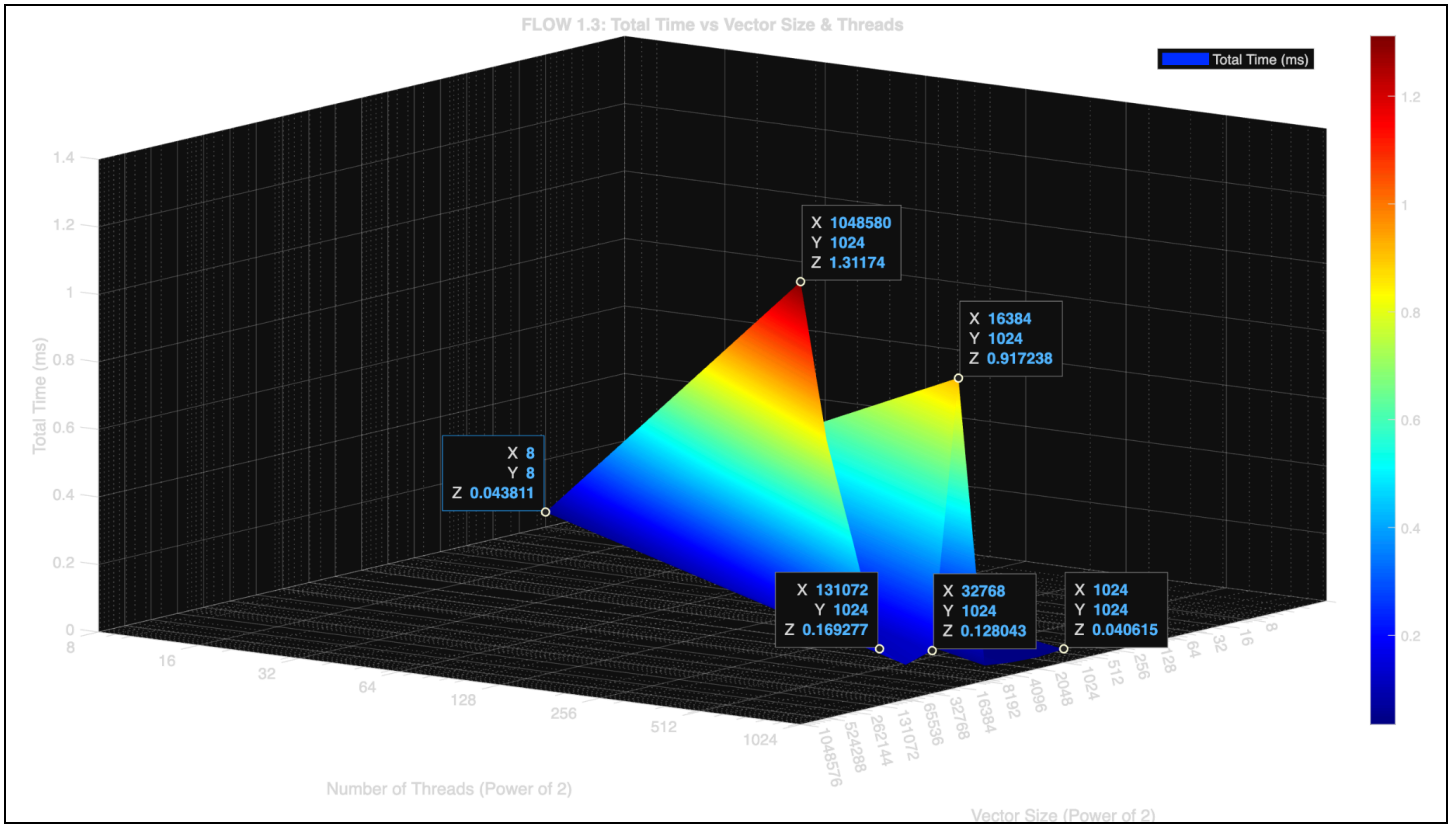


The 3D graphs for the Adaptive Grid (Flow 1.3) illustrate the dynamic relationship between vector size, the parallel hardware configuration (blocks and threads), and execution efficiency. Unlike the 2D plots, these visualizations show how performance shifts as the compute terrain changes.

The 3D plot for Total Time shows a sharp, localized spike at a vector size of 16,384 elements (2 power 14), where time jumps to 0.917 ms.

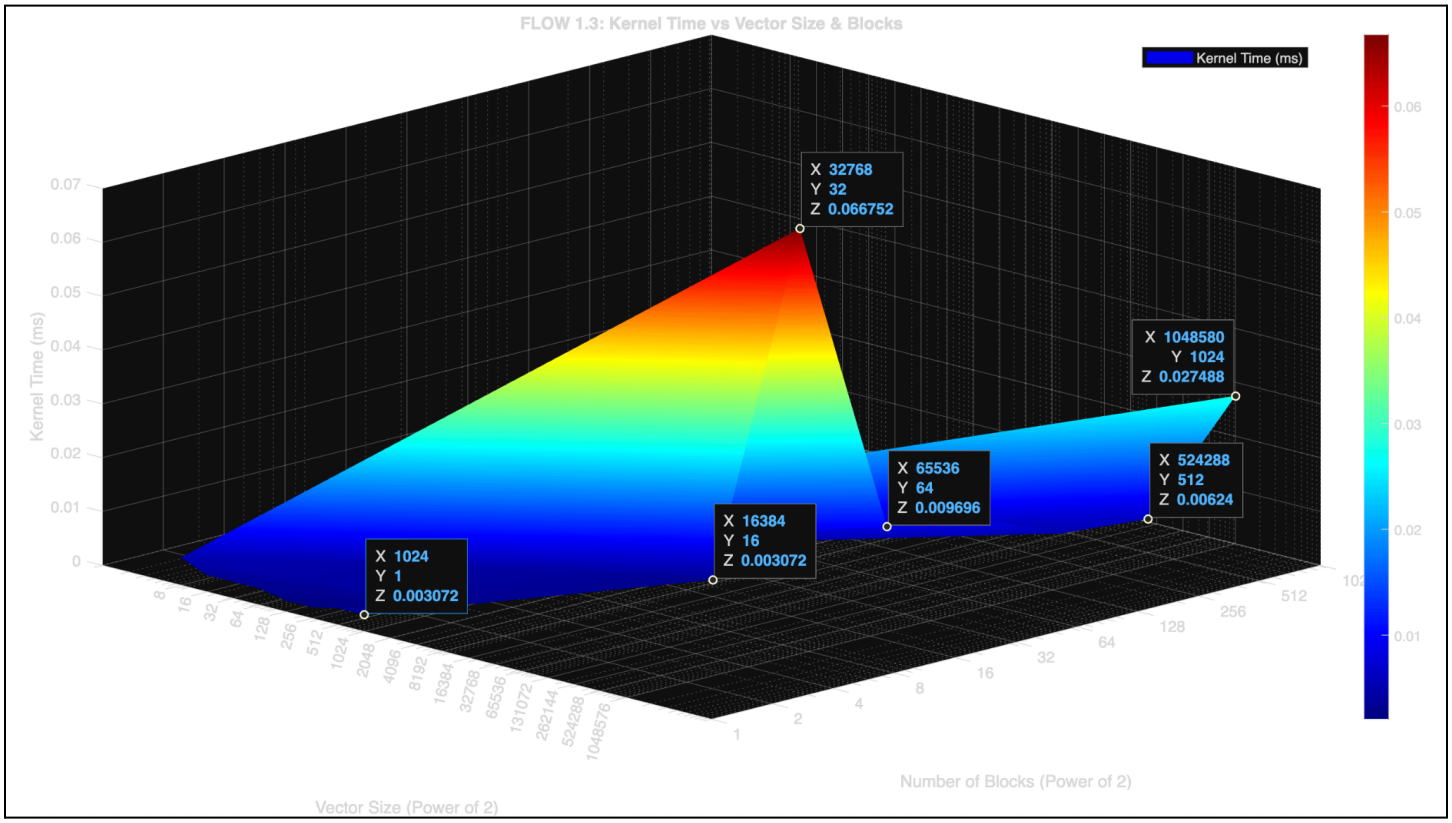
- **Peak:** The peak is primarily due to PCIe Transfer Latency and driver overhead. For this specific size, the overhead of the cudaMemcpy operation to move data to the GPU outweighs the actual computation.
- **Drop:** At the next size (2 power 15), the time drops back down to 0.128 ms. This valley occurs because once the data size reaches a certain threshold, the GPU's high memory bandwidth (1,008 GB/s) and the driver's ability to overlap or stream data more efficiently mask the initial communication latency.

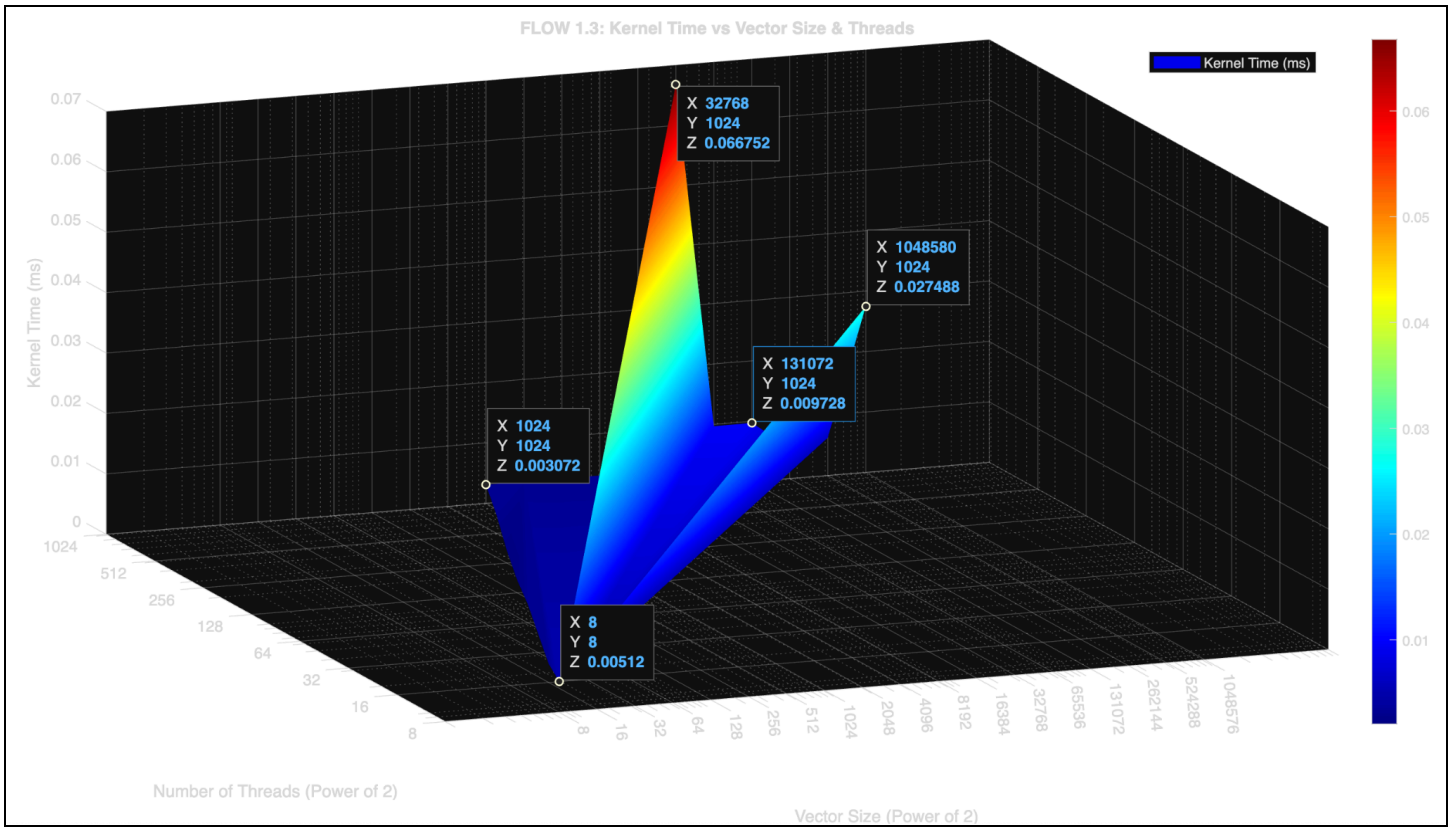




The 3D plot for Kernel Time (pure computation) shows a different behavior than the Total Time. It is mostly flat but features a distinct peak at 32,768 elements (2 power 15), reaching 0.066 ms.

- **Peak:** This peak often occurs due to L2 Cache Thrashing or Warp Scheduling overheads. Because 32,768 is a large power-of-two, the memory access pattern may cause bank conflicts or exceed the capacity of a specific cache level on the RTX 4090, forcing the kernel to wait for data from the slower Global Memory.
- **Drop:** At the very next step (2 power 16), the Kernel Time drops significantly to 0.009 ms. This indicates that with more blocks (64 vs 32), the GPU's scheduler can better hide the memory latency by swapping active warps more effectively.





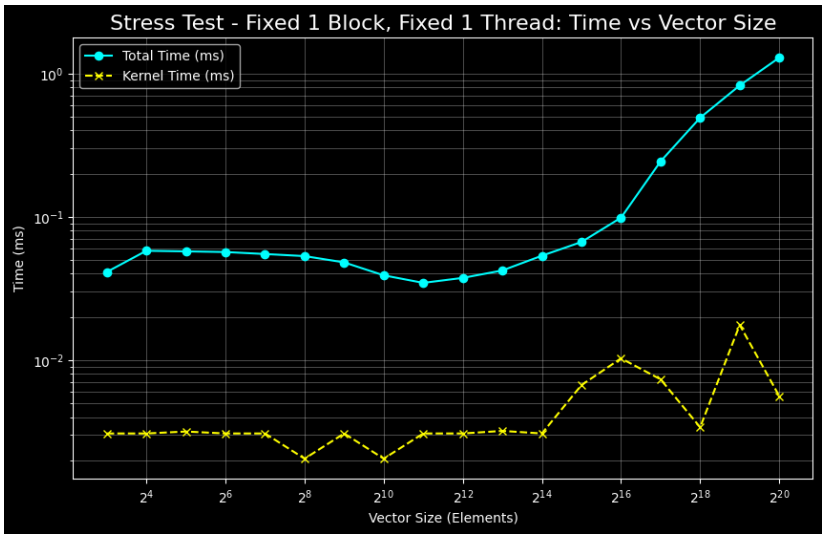
The overall 3D surface looks like a sawtooth or a series of ridges because of two competing factors:

- **Workload Saturation:** As vector size grows, the GPU has more work to do, which naturally pulls the graph up.
- **Parallel Efficiency:** As new blocks are added (2, 4, 8, …), the GPU can distribute that work across more of its 16,384 cores, which pulls the graph "down" in terms of time per element.

7 Corner Cases and Stress Testing

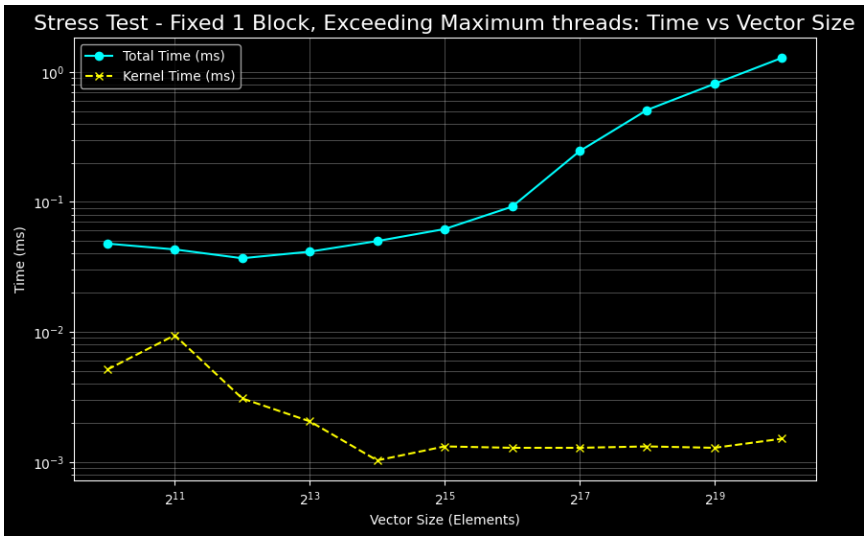
7.1 One Block and One Thread

This is the worst-case scenario. The GPU is forced to act like a very slow sequential processor. The Inclusive Time for 2^{20} elements is 1.290 ms. It lacks the linear predictability of the CPU because the GPU is not designed for single-threaded serial tasks.



7.2 Exceeding Maximum threads

This plot illustrates error handling. Whenever the threads per block exceed 1024, the CUDA runtime returns an invalid argument error. The kernel times recorded here are mostly the time it takes for the system to report the failure. However, there is a steady increase in the total time which includes host to device memory copy. This is because the kernel function throws an error only after the host to device memory copy is completed.



8 Conclusion

The experimental results demonstrate that while sequential CPU execution is predictable and linear, it is quickly outpaced by GPU parallelization as the problem size N increases. The core findings of the study include:

- **Optimal Configuration:** The Adaptive Grid strategy (Flow 1.3), which dynamically scales both the number of blocks and threads based on vector size, proved to be the most efficient approach for achieving maximum throughput.
- **Hardware Constraints:** Stress testing (Flow 2.2) confirmed a strict hardware limit of 1,024 threads per block on the RTX 4090 architecture. Attempting to exceed this limit resulted in kernel launch failures, validating the importance of grid-dimension awareness in CUDA development.
- **Latency vs. Computation:** For smaller vector sizes, the inclusive time (total wall-clock time) was dominated by memory allocation and host-to-device transfer overheads. However, as N grew to 2 power 20, the raw computational power of the GPU's 16,384 CUDA cores effectively amortized these costs.
- **Architectural Efficiency:** The utilization of a single block with varying threads (Flow 1.1) highlighted the diminishing returns of parallelism once the 1,024-thread threshold is reached, emphasizing that true scalability requires multi-block distribution.In summary, effective GPU programming relies on balancing the compute-to-memory ratio and selecting a grid configuration that maximizes the occupancy of the Streaming Multiprocessors.

In summary, effective GPU programming relies on balancing the compute-to-memory ratio and selecting a grid configuration that maximizes the occupancy of the Streaming Multiprocessors.

References

- NVIDIA CUDA C++ Programming Guide (Latest Release 13.1)
- NVIDIA CUDA C++ Best Practices Guide
- "Programming in Parallel with CUDA: A Practical Guide" (2022)