

WIDS 25-26: GPU Programming

Week 2 Assignment
Parallel Thinking and CUDA Programming Basics

Anshika Jain

December 18, 2025

1 Introduction

Graphics Processing Units (GPUs) provide massive parallelism that is well-suited for data-parallel workloads. CUDA exposes this parallelism through a grid–block–thread execution model. This assignment focuses on implementing basic CUDA kernels and understanding how grid and block configurations affect correctness and execution behavior.

2 Task 1: Implementation of Basic CUDA Kernels

Three fundamental CUDA kernels were implemented using single-precision floating-point arithmetic.

2.1 Vector Addition

The vector addition kernel computes:

$$c_i = a_i + b_i$$

Each thread processes one element based on its global index.

2.2 Elementwise Multiply-and-Scale

This kernel computes:

$$c_i = \alpha \cdot a_i \cdot b_i$$

where α is a scalar constant.

2.3 ReLU Activation

The ReLU activation kernel computes:

$$y_i = \max(x_i, 0)$$

2.4 Implementation Details

For each kernel:

- GPU memory was allocated using `cudaMalloc`
- Input data was copied using `cudaMemcpy`
- Kernels were launched using a 1D grid–block configuration

- Results were copied back to the host
- Output was verified against a CPU reference implementation

Bounds checking was included in every kernel using:

```
if (idx < n)
```

to prevent out-of-range memory access.

2.5 Numerical Correctness Verification

Numerical correctness was verified by comparing GPU outputs against CPU-computed reference results. An element-wise comparison was performed using an absolute error tolerance of 10^{-5} to account for floating-point rounding differences.

A kernel was considered correct if all output elements satisfied:

$$|\text{GPU}_i - \text{CPU}_i| < 10^{-5}$$

All three kernels produced numerically correct results for the tested input sizes.

3 Task 2: Grid and Block Configuration Exploration

3.1 Experimental Setup

Input sizes tested:

$$n \in \{10^3, 10^5, 10^7\}$$

Block dimensions tested:

$$\text{blockDim.x} \in \{32, 128, 256, 512\}$$

Grid size was computed using ceiling division:

$$\text{gridSize} = \left\lceil \frac{n}{\text{blockDim.x}} \right\rceil$$

3.2 Results

Table 1: Grid and Thread Configuration Results

n	Block Size	Grid Size	Total Threads
10^3	32	32	1024
10^3	128	8	1024
10^3	256	4	1024
10^3	512	2	1024
10^5	32	3125	100000
10^5	128	782	100096
10^5	256	391	100096
10^5	512	196	100352
10^7	32	312500	10000000
10^7	128	78125	10000000
10^7	256	39063	10000128
10^7	512	19532	10000384

All configurations produced correct output.

```
... ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z9vectorAddPKfs0_Pfi' for 'sm_75'
ptxas info    : Function properties for _Z9vectorAddPKfs0_Pfi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 12 registers, 380 bytes cmem[0]
Vector Addition PASSED
n = 1000, blockSize = 32, gridSize = 32, totalThreads = 1024
```

Figure 1: n=1000, blocksize=32

```
... ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z9vectorAddPKfs0_Pfi' for 'sm_75'
ptxas info    : Function properties for _Z9vectorAddPKfs0_Pfi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 12 registers, 380 bytes cmem[0]
Vector Addition PASSED
n = 1000, blockSize = 128, gridSize = 8, totalThreads = 1024
```

Figure 2: n=1000, blocksize=128

```
... ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z9vectorAddPKfs0_Pfi' for 'sm_75'
ptxas info    : Function properties for _Z9vectorAddPKfs0_Pfi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 12 registers, 380 bytes cmem[0]
Vector Addition PASSED
n = 1000, blockSize = 256, gridSize = 4, totalThreads = 1024
```

Figure 3: n=1000, blocksize=256

```
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z9vectorAddPKfs0_Pfi' for 'sm_75'
ptxas info    : Function properties for _Z9vectorAddPKfs0_Pfi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 12 registers, 380 bytes cmem[0]
Vector Addition PASSED
n = 1000, blockSize = 512, gridSize = 2, totalThreads = 1024
```

Figure 4: n=1000, blocksize=512

```
... ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z9vectorAddPKfs0_Pfi' for 'sm_75'
ptxas info    : Function properties for _Z9vectorAddPKfs0_Pfi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 12 registers, 380 bytes cmem[0]
Vector Addition PASSED
n = 100000, blockSize = 32, gridSize = 3125, totalThreads = 100000
```

Figure 5: n=100000, blocksize=32

```
... ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z9vectorAddPKfs0_Pfi' for 'sm_75'
ptxas info    : Function properties for _Z9vectorAddPKfs0_Pfi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 12 registers, 380 bytes cmem[0]
Vector Addition PASSED
n = 100000, blockSize = 128, gridSize = 782, totalThreads = 100096
```

Figure 6: n=100000, blocksize=128

```

ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z9vectorAddPKfs0_Pfi' for 'sm_75'
ptxas info    : Function properties for _Z9vectorAddPKfs0_Pfi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 12 registers, 380 bytes cmem[0]
Vector Addition PASSED
n = 1000000, blockSize = 256, gridSize = 391, totalThreads = 1000096

```

Figure 7: n=1000000, blocksize=256

```

... ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z9vectorAddPKfs0_Pfi' for 'sm_75'
ptxas info    : Function properties for _Z9vectorAddPKfs0_Pfi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 12 registers, 380 bytes cmem[0]
Vector Addition PASSED
n = 1000000, blockSize = 512, gridSize = 196, totalThreads = 100352

```

Figure 8: n=1000000, blocksize=512

```

... ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z9vectorAddPKfs0_Pfi' for 'sm_75'
ptxas info    : Function properties for _Z9vectorAddPKfs0_Pfi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 12 registers, 380 bytes cmem[0]
Vector Addition PASSED
n = 100000000, blockSize = 32, gridSize = 312500, totalThreads = 100000000

```

Figure 9: n=100000000, blocksize=32

```

... ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z9vectorAddPKfs0_Pfi' for 'sm_75'
ptxas info    : Function properties for _Z9vectorAddPKfs0_Pfi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 12 registers, 380 bytes cmem[0]
Vector Addition PASSED
n = 100000000, blockSize = 128, gridSize = 78125, totalThreads = 100000000

```

Figure 10: n=100000000, blocksize=128

```

ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z9vectorAddPKfs0_Pfi' for 'sm_75'
ptxas info    : Function properties for _Z9vectorAddPKfs0_Pfi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 12 registers, 380 bytes cmem[0]
Vector Addition PASSED
n = 100000000, blockSize = 256, gridSize = 39063, totalThreads = 10000128

```

Figure 11: n=100000000, blocksize=256

```

... ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z9vectorAddPKfs0_Pfi' for 'sm_75'
ptxas info    : Function properties for _Z9vectorAddPKfs0_Pfi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 12 registers, 380 bytes cmem[0]
Vector Addition PASSED
n = 100000000, blockSize = 512, gridSize = 19532, totalThreads = 10000384

```

Figure 12: n=100000000, blocksize=512

3.3 Discussion

3.3.1 Excess Threads

When the total number of threads exceeded n , extra threads computed indices beyond the valid range. Due to bounds checking, these threads performed no computation, preserving correctness.

3.3.2 Need for Bounds Checking

Bounds checking is essential to prevent illegal memory access. Without this check, threads with invalid indices could cause undefined behavior such as incorrect results or program crashes.

3.3.3 Configuration Behavior

No configurations failed or behaved unexpectedly. Correctness was maintained across all tested block sizes and input sizes.

3.3.4 Block Size Trade-offs

Smaller block sizes offer better flexibility for small workloads but increase scheduling overhead. Larger block sizes reduce launch overhead and improve efficiency for large workloads but may increase register and resource usage. Block sizes between 128 and 256 provided a good balance for this workload.

4 Task 3 (Optional): First Look at Timing

This task provides an initial performance baseline for the vector addition kernel by measuring kernel execution time without applying any optimizations.

4.1 Timing Methodology

Kernel execution time was measured using `cudaEvent_t`. CUDA events were recorded immediately before and after the kernel launch, and the elapsed time was computed using `cudaEventElapsedTime`. This approach measures only the kernel execution time on the GPU and excludes memory allocation and host–device data transfers.

4.2 Experimental Setup

The vector addition kernel was executed for multiple input sizes and block dimensions:

- Input sizes: $n \in \{10^3, 10^5, 10^7\}$
- Block dimensions: `blockDim.x` $\in \{32, 128, 256, 512\}$

For each configuration, the kernel was launched once and the execution time was recorded. No performance optimizations such as shared memory usage or kernel fusion were applied.

4.3 Timing Results

Table 2: Kernel Execution Time for Vector Addition

Input Size (n)	Block Size	Kernel Time (ms)	Correct Output
10^3	32	0.136224	Yes
10^3	128	0.099008	Yes
10^3	256	0.125472	Yes
10^3	512	0.124064	Yes
10^5	32	0.147776	Yes
10^5	128	0.1088	Yes
10^5	256	0.117952	Yes
10^5	512	0.120032	Yes
10^7	32	1.11936	Yes
10^7	128	0.567584	Yes
10^7	256	0.562592	Yes
10^7	512	0.565408	Yes

4.4 Observations

For small input sizes, kernel execution time was dominated by launch overhead, resulting in minimal differences across block sizes. As the input size increased, larger block sizes generally resulted in lower execution times due to improved parallelism and reduced scheduling overhead.

4.5 Summary

This preliminary timing analysis establishes a baseline performance for the vector addition kernel. The results highlight the influence of execution configuration on runtime and motivate further optimization and performance analysis.

5 Conclusion

This assignment demonstrated the implementation of basic CUDA kernels and explored the effects of grid and block configuration on correctness. Proper grid sizing and bounds checking ensured safe and correct execution across a wide range of configurations. These experiments lay the foundation for performance and optimization analysis in subsequent work.