Assignment 2: CUDA Programming and Execution Model Exploration

Sruthi Manepalli

1 Introduction

This assignment explores the fundamentals of CUDA programming and the GPU execution model through hands-on implementation and experimentation. The primary goals are to understand how GPU threads are mapped to data, how memory is managed between the host and device, and how different grid and block configurations influence kernel execution. The assignment is divided into two parts: implementation of basic CUDA kernels and an exploration of grid–block configurations for correctness and behavior.

2 Part 1: CUDA Kernel Implementations

Part 1 focuses on implementing basic data-parallel operations using CUDA. In all kernels, a single GPU thread is responsible for computing one element of the output array. The global thread index is computed using the CUDA execution model as:

$$\text{idx} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$$

This mapping allows a one-dimensional grid of threads to process a one-dimensional data array efficiently.

2.1 Vector Addition

The vector addition kernel computes:

$$c[i] = a[i] + b[i]$$

Each thread loads one element from arrays a and b, performs the addition, and stores the result in array c. This kernel demonstrates the simplest form of data parallelism, where all threads execute the same instruction sequence on different data elements.

2.2 Multiply and Scale

The multiply-and-scale kernel computes:

$$c[i] = \alpha \cdot a[i] \cdot b[i]$$

where α is a scalar constant passed as a kernel argument. This kernel illustrates how scalar parameters are broadcast to all threads and how additional arithmetic operations are performed within each thread.

2.3 ReLU Activation

The ReLU (Rectified Linear Unit) kernel computes:

$$y[i] = \max(x[i], 0)$$

This kernel introduces conditional execution within each thread. Threads independently evaluate a comparison and write either the input value or zero to the output array, demonstrating branch behavior in GPU kernels.

2.4 Bounds Checking and Correctness

In all kernels, an explicit bounds check (`idx < n`) is used. This is necessary because the total number of launched threads may exceed the input size due to grid size rounding. Without this check, threads with indices outside the valid range would access illegal memory locations, leading to incorrect results or runtime errors.

Correctness was verified by copying GPU results back to the host and comparing them against CPU-computed reference values using a small numerical tolerance. All kernels produced correct results for all tested input sizes.

3 Part 2: Grid and Block Configuration Exploration

Part 2 investigates how varying grid and block dimensions affect kernel execution. The vector addition kernel from Part 1 was reused, and no changes were made to the kernel logic. Instead, only launch parameters were varied.

3.1 Experimental Setup

The kernel was executed for the following input sizes:

$$n = 10^3,\ 10^5,\ 10^7$$

and block sizes:

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

For each configuration, the grid size was computed as:

$$\text{gridSize} = \left\lceil \frac{n}{\text{blockSize}} \right\rceil$$

3.2 Observed Results

Table 1 summarizes the observed results. In all configurations, the kernel produced correct output.

3.3 Answers to Part 2 Questions

Why does the number of threads often exceed the input size?

CUDA launches threads based on the specified grid and block dimensions, not directly on the input size. Since the grid size must be an integer, it is rounded up when the input size is not an exact multiple of the block size. This rounding results in extra threads being launched.

Table 1: Grid and Block Configuration Results for Vector Addition

Input Size (n)	Block Size	Grid Size	Total Threads	Result
1000	32	32	1024	PASS
1000	128	8	1024	PASS
1000	256	4	1024	PASS
1000	512	2	1024	PASS
100000	32	3125	100000	PASS
100000	128	782	100096	PASS
100000	256	391	100096	PASS
100000	512	196	100352	PASS
10000000	32	312500	10000000	PASS
10000000	128	78125	10000000	PASS
10000000	256	39063	10000128	PASS
10000000	512	19532	10000384	PASS

What happens to threads with indices greater than or equal to n?

Threads whose computed index satisfies idx $\geq n$ fail the bounds check and do not perform any computation. These threads exit immediately without accessing memory, ensuring correctness.

Why is bounds checking necessary in CUDA kernels?

Without bounds checking, threads with indices beyond the valid data range would access invalid memory locations, leading to incorrect results or undefined behavior. Bounds checking is therefore essential for correctness whenever the total number of launched threads exceeds the problem size.

How does block size affect execution behavior?

Changing the block size affects the grid size and the total number of launched threads. Smaller block sizes generally lead to finer-grained parallelism with fewer excess threads, while larger block sizes may improve hardware utilization but increase the number of unused threads due to rounding. In all cases, correctness is preserved due to bounds checking.

Does block size affect correctness?

No. As long as the kernel uses proper bounds checking, correctness is independent of block size. Different block sizes only affect how work is distributed among threads, not the final result.

4 Conclusion

This assignment provided practical experience with CUDA kernel development and the GPU execution model. Part 1 demonstrated how simple data-parallel operations can be efficiently implemented on the GPU using threads and blocks. Part 2 highlighted how grid and block configurations influence execution behavior and reinforced the importance of bounds checking. Together, these experiments build a strong foundation for understanding performance optimization and memory-aware GPU programming in more advanced applications.