Homework Assignment 1: Matrix Addition Using CUDA

Haohan Tsao B09607009

April 26, 2025

1 Introduction

This report presents the implementation and optimization of a modified matrix addition operation using CUDA parallel programming. The specific operation is defined as $c(i, j) = 1/a(i, j) + 1/b(i, j)$, where a and b are input matrices of size $N \times N$ with $N = 6400$. The elements of matrices A and B were initialized with random values between 0.0 and 1.0.

The primary objective of this assignment is to determine the optimal block size for this computation on the NVIDIA GeForce GTX 1060 GPU architecture. By testing different block sizes, we can identify the configuration that provides the best performance for this specific operation on the given hardware.

2 Experimental Setup

2.1 Hardware and Software Configuration

The experiments were conducted on the TWCP1 cluster with the following configuration:

- **GPU**: NVIDIA GeForce GTX 1060 (Pascal architecture, Compute Capability 6.1)
- **CUDA Version**: 12.3
- **Operating System**: Debian GNU/Linux 11.9, kernel version 4.19.172

2.2 Implementation

The implementation follows the standard CUDA programming model. For each block size configuration, we defined a two-dimensional grid of thread blocks. Each thread computes one element of the output matrix C using the formula $c(i,j) = 1/a(i,j) + 1/b(i,j)$.

The tested block sizes were 4, 8, 16, 24, and 32 threads per dimension. For each block size, we adjusted the grid dimensions to cover the entire matrix.

2.3 Performance Metrics

To evaluate performance, we measured:

- **GPU Processing Time**: Time spent on computation (excluding I/O operations)
- **GPU Gflops**: Floating point operations per second
- **Speedup**: Ratio of CPU processing time to GPU processing time

3 Results and Analysis

3.1 Performance Comparison

The performance results for different block sizes are presented in Table 1.

Table 1: Performance Comparison for Different Block Sizes

Block Size	Grid Dimensions	GPU Time (ms)	GPU Gflops	Speedup
4	1600×1600	19.26	6.38	2.62
8	800×800	9.67	12.71	2.86
16	400×400	9.67	12.71	2.91
24	267×267	9.78	12.57	2.89
32	200×200	9.72	12.64	2.90

3.2 Analysis of Block Size Impact

From the results, we can make several key observations:

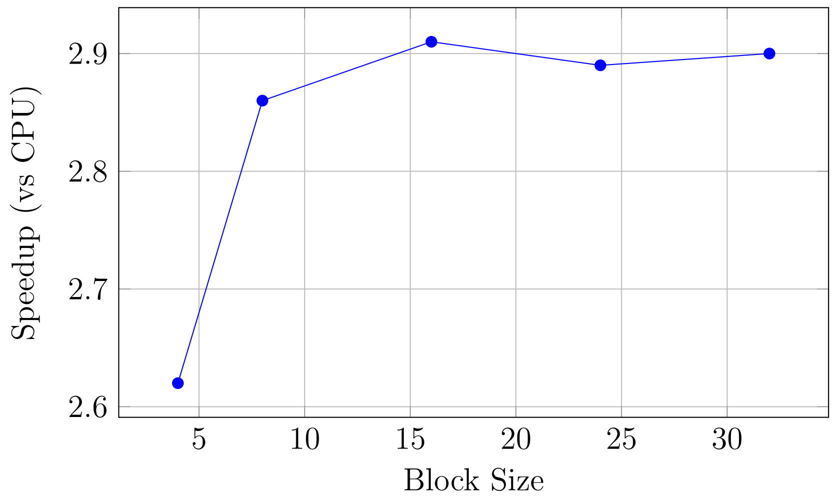

Figure 1: Speedup vs Block Size

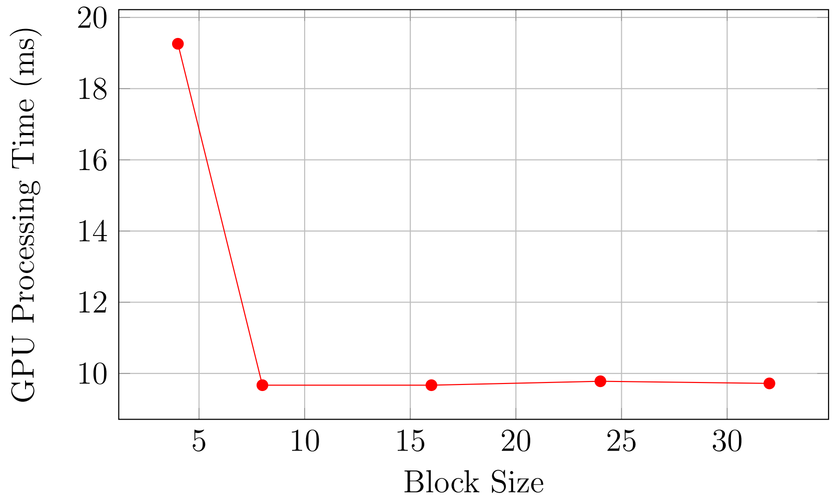

Figure 2: GPU Processing Time vs Block Size

3.2.1 Block Size 4

The smallest block size ($4 \times 4 = 16$ threads per block) performed significantly worse than all other configurations, with almost double the processing time. This can be attributed to:

- **Excessive Scheduling Overhead**: With a grid size of $1600 \times 1600 =$ 2,560,000 blocks, the overhead of block scheduling becomes significant.
- **Underutilization of SM Resources**: Each block contains only 16 threads, which is half a warp (32 threads). This leads to inefficient warp execution.
- **Low Occupancy**: The small number of threads per block limits the ability to hide latency through thread-level parallelism.

3.2.2 Block Sizes 8, 16, 24, and 32

These block sizes showed similar performance, with differences in the range of 2-3%. However, block size 16 achieved the highest speedup (2.91×). The reasons for this optimal performance include:

- **Balanced Resource Utilization**: A 16×16 block contains 256 threads, which is 8 complete warps. This provides good occupancy while not oversubscribing the SM resources.
- **Efficient Grid Size**: The grid dimensions ($400 \times 400 =$ 160,000 blocks) provide sufficient parallelism without excessive scheduling overhead.
- **Memory Access Patterns**: This block size facilitates efficient coalesced memory accesses, which is crucial for memory-bound operations like matrix addition.
- **Warp Execution Efficiency**: With 8 complete warps per block, this configuration allows for efficient scheduling and execution of warps within each SM.

3.2.3 Larger Block Sizes (24 and 32)

While block sizes 24 and 32 still perform well, they show a slight decrease in performance compared to block size 16:

- Block size 24 creates irregular warps ($24 \times 24 = 576$ threads, which is 18 warps with the last warp only partially filled).

- Block size 32 (32×32 = 1024 threads) uses the maximum number of threads per block, which may limit the number of blocks that can be scheduled concurrently on each SM due to resource constraints.

4 Conclusion

Based on our experimental results, we conclude that a block size of 16×16 provides the optimal performance for the modified matrix addition operation on the NVIDIA GeForce GTX 1060 GPU. This configuration achieves a speedup of 2.91× compared to the CPU implementation.

The performance characteristics observed align with the theoretical considerations of the CUDA programming model and the Pascal architecture:

- Block sizes that are too small create excessive scheduling overhead
- Block sizes that are too large may limit concurrent block execution
- The optimal configuration balances these factors to maximize throughput

The relatively modest speedup (2.91×) suggests that this particular operation is likely memory-bound rather than compute-bound, as matrix addition involves simple arithmetic operations but requires significant memory bandwidth.

5 Appendix: Source Code

The complete source code for this implementation is included as a separate file in the submission package.