CS 239 Exercise 1 Report

Gerry Agluba Jr.

CS 239 WZZQ- Parallel Computing
Date of Submission: March 4, 2025

1. OBJECTIVES

This report highlights capabilities to write simple parallel programs in a heterogonous computer systems. Additionally, this report showcase insights on designing a simple programs by implementing well-known concepts in parallel computing as well as basic concepts in CUDA programming. This exercise involves creating a scalable parallel program for a simple Matrix Addition program.

2. METHODOLOGY

The following methods lies on the assumption that a heterogonous computer system is already setup. In particular, a working CUDA environment with compatible C++ compiler should be set up prior to design, programming and tests.

2.1 Environment GPU Device Properties

Initially, different device properties were queried to plan different experimental setup that can be made as well as determining the limitations of current environment. A utility function to print device properties were written with the following values observed.

Device Count **1**
==========
Device NVIDIA GeForce GTX 1650 SUPER Properties:
==========
Clock Rate (KHz) 1740000
Memory Clock Rate (KHz) 6001000
Total Global Memory (MB): 4095
Shared Memory / Block (KB): 48
Warp Size: 32
Pitch (MB): 2047
Max Threads / Block: 1024
Max Dimension Size of Block : 1024, 1024, 64
Max Dimension Size of Grid : 2147483647, 65535, 65535
Multiprocess Count: 20
Max Blocks / Multiprocessor: 16
Concurrent Kernels: 1
Max Threads / Multiprocessor: 1024
Shared Memory (KB) / Multiprocessor: 64
==========

The following values will ultimately determine the optimal parameters: in our case are **block dimensions** , **number of threads / block** and **input size**. In our environment, **20 multiprocessor (MP) count** each accommodating **1024 threads** where observed. We can draw initial insights that we can use a maximum of **1024 / 16 = 64 threads/block** to fully utilized the a single processor.

2.2 Notes on Shared Memory, Blocking and Tiling

In this report, optimization on shared memory were not explored for the reason that the algorithm (matrix addition) do not have that much input codependency. Meaning, the operation involves in each parallel threads will always gonna access elements of the matrix once ($B[i][j] + C[i][j]$), and thus shared memory is of little benefit to the algorithm.

2.3 Experimental Setup

2.3.1 General Notes

- Each experiment will be replicated 10 times.
- Each experiment input matrices will be randomly generated from a uniform distribution ($M[i][j] \sim \mathcal{U}(0, 100)$) with matric sizes from $2^9 - 2^{12}$.
- To isolate GPU performance, Experiment are timed only **before and after kernel function**. Timing the whole execution might lead to unexpected results due to high runtime of loading data between device/s and host.

2.3.2 Experiments

Table 1 details the experiments in this report. Setup 1 details on how the algorithm scale with increasing matrix size. Setup 2 and Setup 3 details how fix matrix size with varying block sizes. Additionally, it can be said that Setup 1.1 is a theoretical benchmark based on the device properties that we gathered. Block size **32 x 32 = 1024** can represent the theoretical Max Threads / Multiprocessor *(assuming the threads are scheduled in a single processor)*. Parameter variation can be built up from here.

2.4 Implementation [1]

Input matrices are initialized using the **C++ random** library 2. Kernel function implementationa are furthere discuss in the next subsection.

[1]The whole implementation can be found on github repository

ID	Matrix Size	Grid Dim	Block Size	Runs
Setup 1				
1.1	512 x 512	16 x 16	32 x 32	10
1.2	1024 x 1024	32 x 32	32 x 32	10
1.3	2048 x 2048	64 x 64	32 x 32	10
Setup 2				
2.1	2048 x 2048	256 x 256	8 x 8	10
2.2	2048 x 2048	128 x 128	16 x 16	10
2.3	2048 x 2048	64 x 64	32 x 32	10
Setup 3				
3.1	4096 x 4096	512 x 512	8 x 8	10
3.2	4096 x 4096	256 x 256	16 x 16	10
3.3	4096 x 4096	128 x 128	32 x 32	10

Table 1: Experiment Setup for Matrix Addition

2.4.1 Kernel Functions

Three kernel functions are implemented:

1. **kernel_1t1e** - each thread producing one output matrix element
2. **kernel_1t1r** - each thread producing one output matrix row.
3. **kernel_1t1c** - each thread producing one output matrix column

Please see implementation reference here 1

2.4.2 Matrix Addition Kernel Functions [htt25]

The

3. RESULTS

Table 2 details the kernel runtime results of different experiment setup in 1. Values in **bold face** are kernel minimum and values in *italics* are setup minimum.

ID	kernel_1t1e	kernel_1t1r	kernel_1t1c
Setup 1			
1.1	0.00003405	0.00002935	0.00003494
1.2	0.00003056	0.00002959	0.00003086
1.3	0.00003489	0.00003812	0.00003359
Setup 2			
2.1	0.00004073	0.00004027	*0.00003643*
2.2	***0.0000351***	0.00004632	0.00004185
2.3	0.00003915	***0.00003474***	**0.00003575**
Setup 3			
3.1	***0.00003693***	0.00004558	0.00004144
3.2	*0.00003873*	0.00004335	0.00005409
3.3	0.000037	***0.00003692***	**0.00003787**

Table 2: Runtime (s) of kernel functions against different experiment setup

4. OBSERVATIONS AND INSIGHTS

During the experiments, the following general observations were encountered.The first run of any kernel function always have a signigicantly higher runtime than succeeding runs. With this the setup were modified to ignore the first runtime, and treat as a **warm up run**. Additionally, obvious CUDA Runtime exceptions were encountered when setting thread block sizes over the the set GPU max threads / block.

All kernel functions scales relative well with matrix of dimension sizes: $2^9 - 2^{11}$ on each side given a fix block size of 32×32, with no significant difference against either type of kernel function used or the matrix shape. This might probably because the matrix size is to small enough to see any reliable differences.

In experiments setup 2 and setup3, **kernel_1t1e** is relatively faster (especially on setup3) on thread block sizes lower than the device's set max threads / block. In contrast, **kernel_1t1r** and **kernel_1t1c** have relatively better performances at the limits. This is to be expected considering the complexity of each thread operation on the two kernel functions is $O(n)$ and increasing thread block size up to the limit can signficantly improve its performance.

Its noteworthy, that **kernel_1t1e** ignores this performance improvement trends at higher thread block size probably because the amount of threads that the GPU device can handle at our current experiment setups is not that yet significant to have bottlenecks.

5. RECOMMENDATION

Though initial insights are presented, It can be said that the results are still premature to have a reliable conclusion, at least for the case of **kernel_1t1e**. The following items are therfore still recommended.

1. Analysis on Memory Access and Memory Runtime
2. Feasibility and Analysis(if possible) on Rectangular(Non-Square) shaped thread block
3. Experiment on much bigger matrix shapes

6. REFERENCES

[htt25] https://docs.nvidia.com/cuda/cuda-c-programming-guide/. *CUDA C++ Programming Guide*. 2025. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/` (visited on 02/28/2025).

APPENDIX

Listing 1: Kernel Function Implementation

```
// each kernel function thread has copy of global variables
__global__ void kernel_1t1e(float* A_d, float* B_d, float* C_d, int width) {
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;

        if ((row < width) && (col < width)) {
                // each thread computes one element
                A_d[row * width + col] = B_d[row * width + col] + C_d[row * width + col];
        }
}

__global__ void kernel_1t1r(float* A_d, float* B_d, float* C_d, int width) {
        int row = blockIdx.y * blockDim.y + threadIdx.y;

        if ((row < width)) {
                // each thread computes row
                for (int col = 0;col < width;col++) {
                        A_d[row * width + col] = B_d[row * width + col] + C_d[row * width + col];
                }
        }
}

__global__ void kernel_1t1c(float* A_d, float* B_d, float* C_d, int width) {
        int col = blockIdx.x * blockDim.x + threadIdx.x;

        if ((col < width)) {
                // each thread computes col
                for (int row = 0;row < width;row++) {
                        A_d[row * width + col] = B_d[row * width + col] + C_d[row * width + col];
                }
        }
}
```

Listing 2: Kernel Function Implementation

```
float* generateRandomMatrix(int size, pair<int, int> range) {
        random_device rd;
        mt19937 gen(rd());
        uniform_real_distribution<float> dis(range.first, range.second);

        float* matrix = new float [size*size];

        for (int i = 0;i < size;i++) {
                for (int j = 0;j < size;j++) {
                        matrix[i*size+j] = dis(gen);
                }
        }
        return matrix;
}
```