CPSC 524 Assignment 4: CUDA Matrix Multiplication

Rami Pellumbi*

December 10, 2023

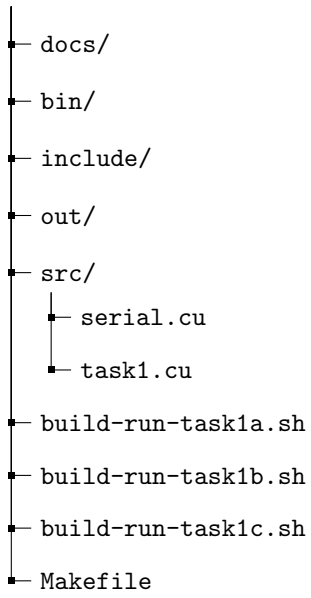
*M.S., Statistics & Data Science

1 Introduction

This assignment explores GPU programming using CUDA, centered around the task of matrix multiplication. The primary challenge involves constructing a CUDA kernel capable of multiplying two random rectangular matrices. This initial task serves as a gateway into the realms of parallel computing and efficient memory management, foundational elements in leveraging GPU architecture for computational tasks. As we progress, the report addresses more advanced techniques, such as the utilization of shared memory and the strategic optimization of thread computations, essential for enhancing the computational performance.

2 Project Organization

The project is laid out as follows:



- **docs/**: This folder contains LaTeX files and other documentation materials that pertain to the report.
- **bin/**: The `bin` folder holds compiled objects and executable files, centralizing the output of the compilation process.
- **include/**: Here, all the header files (`.h`) are stored.
- **out/**: The `out` folder stores the outputs from each task. It also houses the csv file containing data generated by the programs.
- **src/**: This directory houses the source files (`.cu`) that make up the benchmarks.
- **Shell Scripts**: The shell scripts are used to submit the job for the relevant task to slurm via `sbatch`.

3 Code Explanation, Compilation, and Execution

This section outlines the steps required to build and execute the code. The provided Bash scripts automate the entire process, making it straightforward to compile and run the code. All the below steps assume you are in the root of the project directory.

3.1 Automated Building and Execution

All related code is in the `src/` directory. There are multiple programs:

- `task1.cu`
- `task2.cu`
- `task3.cu`
- `task4.cu`

To run any one experiment, execute the relevant bash script, e.g., `build-run-task1a.sh`. It should be noted that for the `build-run-task1b.sh` the `FP` must be defined to double on line 1.

3.2 Post-Build Objects and Executables

Upon successful compilation and linking, an `obj/` subdirectory will be generated within the directory. This directory will contain the compiled output files. Additionally, the executable files for running each program will be situated in the `bin/` subdirectory.

3.3 Output Files From `sbatch`

The output files generated from running the code by submitting the relevant Bash script via `sbatch` will be stored in the `out` directory.

4 Task 1: CUDA Matrix Multiplication

First, a CUDA kernel is built to handle the multiplication of two random rectangular matrices: $C = AB$, where A is $n \times p$ and B is $p \times m$. Global memory is used to store the matrices and perform the computation. Each GPU thread computes only a single element of C. The input matrices A and B are stored row-wise into a one-dimensional array, and the output matrix C is stored in the same manner. In addition to the kernel, a host multiplication takes form using the kij access pattern:

```
for (int k = 0; k < p; k++) {
    size_t ia = i * p + k; // row i column k of A
    r = A[ia];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            size_t ib = k * m + j; // row k column j of B
            size_t ic = i * m + j; // row i column j of C

            C[ic] += r * B[ib];
        }
    }
}
```

4.1 GPU Kernel Performance: float

The kernel was written to accept 7 arguments:

- `int n`: Number of rows in A and C.
- `int p`: Number of columns in A and rows in B.
- `int m`: Number of columns in B and C.
- `int block_dim_x`: The tile size to use for the computation.
- `int block_dim_y`: The tile size to use for the computation.
- `grid_dim_x`: The number of blocks to use in the x-direction.
- `grid_dim_y`: The number of blocks to use in the y-direction.

All device multiplication was validated in error performance against the serial code when able. This code was omitted from the individual task files for clarity, and the serial code put in its own `serial.cu` for clarity. The following bash script outline was used in Task 1, allowing for control over varying the block and grid size:

```
# n,m,p blocks delimited by a space
sizes="1024,1024,1024 8192,8192,8192 1024,1024,8192 8192,8192,1024 8192,1024,8192"

for tuple in $sizes
do
    IFS=',' read -ra ADDR <<< "$tuple"
    n=${ADDR[0]}
    m=${ADDR[1]}
    p=${ADDR[2]}

    echo "Running n=$n, p=$p, m=$m"
    for blockx in 8 16 32 64 128 256
    do
        blocky=$((1024/$blockx))
        Grid_Dim_x=$((($m + $blockx - 1)/$blockx))
        Grid_Dim_y=$((($n + $blocky - 1)/$blocky))
        echo "Running block BLOCK_DIM_X=$blockx BLOCK_DIM_Y=$blocky"
        echo "With GRID_DIM_X=$Grid_Dim_x GRID_DIM_Y=$Grid_Dim_y"
        time ./bin/task1 $n $p $m $blockx $blocky $Grid_Dim_x $Grid_Dim_y
        time ./bin/task1 $n $p $m $blockx $blocky $Grid_Dim_x $Grid_Dim_y
        time ./bin/task1 $n $p $m $blockx $blocky $Grid_Dim_x $Grid_Dim_y
    done
done
```

The performance table for the best performing (`block_x, block_y`) pair across the verying matrix sizes is:

Table 1: GPU Kernel Performance Float (ms)

(n, m, p)	(block x, blocky)	(grid x, grid y)	gpu time (ms)	cpu time (ms)
(1024,1024,1024)	(64, 16)	(16, 64)	3.628693	253.5439
(1024,1024,8192)	(128, 8)	(8, 128)	42.841162	2218.2016
(8192,8192,1024)	(16, 64)	(512, 128)	271.195058	28123.4727
(8192,1024,8192)	(64, 16)	(16, 512)	290.632660	25230.9069
(8192,8192,8192)	(16, 64)	(512, 128)	2179.101807	224239.7760

We observe that the GPU offers significant speedup, up to about 100× in some cases

4.2 GPU Kernel Performance: double

We assess the performance of the GPU on doubles for square matrix $n = m = p = 8192$. The results were:

Table 2: GPU Kernel Performance Double (ms)

(n,m,p)	(block x block y)	(grid x, grid y)	gpu time
(8192,8192,8192)	(8,128)	(1024,64)	3462.878
(8192,8192,8192)	(16,64)	(512,128)	2870.899
(8192,8192,8192)	(32,32)	(256,256)	2826.502
(8192,8192,8192)	(64,16)	(128,512)	2944.203
(8192,8192,8192)	(128,8)	(64,1024)	3005.359
(8192,8192,8192)	(256,4)	(32,2048)	3231.829

We see that the best performing kernel was for block dimensions $(32, 32)$ and grid dimensions $(256, 256)$.

This is a change from the best performing block dimensions of the float kernel, which was $(16, 64)$. The gpu execution time increased from 2179.101807 to 2826.502, a 29.7% increase.

4.3 Maximum Matrix Size

The RTX 2080 Ti has 11,012 MegaBytes of memory. That is

$$11,012MB \times \frac{1024\ KB}{MB} \times \frac{1024\ B}{KB} = 11546918912\ \mathsf{Bytes} := C.$$

Each float carries 4 bytes and thus the maximum matrix size that can be allocated is subject to the constraint

$$4\big(n \cdot p + p \cdot m + n \cdot m\big) \leq C.$$

For square matrices $n = p = m$, this reduces to

$$4 \cdot 3 \cdot n^2 \leq C \implies n \leq \sqrt{\frac{C}{12}} = 31020$$

Of course, this is the maximum size that can be allocated, but the actual maximum size is subject to the memory required by other processes. The maximum size that was able to be allocated was $n = 30536$. The timing result table is:

Table 3: Max Size GPU Kernel Performance Float (ms)

(n,m,p)	gpu time
(30536,30536,30536)	121486.5

5 Task 2: CUDA Matrix Multiplication with Shared Memory

6 Task 3: Reducing Tile Loads

7 Task 4: Handling All Tile Sizes

8 Conclusion