Programming Project #2

Due Date: 12/6, Tue., 11:59 p.m. Please submit via Blackboard. Late submissions are accepted till 12/9, Friday 11:59 p.m., with 10% penalty each day.

Please name your submission file starting with "LastName_FirstName_PP2".

Many problems in computer science require performing fast matrix multiplication. Modern machine learning workloads rely heavily on computations involving matrices, and GPUs are proven to accelerate such workloads. In this programming project, you are asked to develop an **optimal matrix multiplication** using CUDA. This programming project assists you in some understanding of parallel computing using the CUDA framework. This programming project intends to reinforce our discussions in class and your knowledge of lectures related to GPU architecture and CUDA programming.

Problem: Given two `N*N` matrices `X` and `Y` of floating-point numbers, perform matrix multiplication and store the result on an `N*N` matrix `ans`. First, to explain why this is a well-suited problem for parallel programming, we show you a sequential version of matrix multiplication in the `matrixMul_cpu.cpp` file. After that, we move on to the GPU version and its performance evaluation. However, we notice that the given CUDA code does not take advantage of any parallel programming principles. Therefore, please optimize this code as we did for vector addition during the lecture. Finally, compare the results.

Part 1: Getting Started

In this part of the programming project, we have provided a functional sample code. To begin, we ask you to run both the source code's CPU and GPU versions.

```
$ git clone https://github.com/mertside/CS5375_GPU_Lecture
```

To compile these source codes, use a command like the one below. This command compiles the `matrixMul_cpu.cpp` source file and generates a binary executable file, `matrixMul_cpu.exe`. Or you can use the `make` utility.

```
$ g++ matrixMul_cpu.cpp -o matrixMul_gpu.exe
```

The CUDA code provided for your GPU program is incomplete! Please address the comments marked as `TODO` and complete the program. Then, compile the CUDA code using a command like the one below. This command compiles the `matrixMul_gpu.cu` source file and generates a binary executable file, `matrixMul_gpu.exe`. Or you can use the `make` utility.

```
$ nvcc matrixMul_gpu.cu -o matrixMul_gpu.exe
```

To run the code, please use a command like the one below.

```
$ ./matrixMul_cpu.exe
```

And

```
$ ./matrixMul_gpu.exe
```

Although the output already includes some timing, you can also use `time` or `nvprof` for timing.

Requirements of Part 1:

1. Complete the GPU version of the program and write a `Makefile` to compile your programs. Finally, please compile and run the codes and report the outputs.
2. How do these two versions of the code compare? Report how long it takes to execute each version of the matrix multiplication.
3. Is the GPU version optimal? Report why or why not?

Part 2. Multiple Threads
Modify the loop to stride through the array with parallel threads. Remember that `threadIdx` contains the current thread index within its block, and `blockDim` contains the number of threads in the block.

Profile this version of your code using a command like the one below.

```
$ nvprof ./matrixMul_gpu_v2.exe
```

Requirements of Part 2:

1. Please compile and run your modified code and report the outputs.
2. Profile this code. How does this version compare to your GPU code in Part 1? Please report it.

Part 3. Multiple Blocks
Update the kernel code to consider the entire grid of thread blocks. CUDA provides `gridDim`, which contains the number of blocks in the grid, and `blockIdx`, which contains the index of the current thread block in the grid.

If,

`t` is the thread number of a thread inside a particular block,
`b` is the block number of a block inside the grid,
`B` is the total number of blocks,
`T` is the total number of threads per block,
Let's define the number of the assigned cell to a thread as `AC`.

Then

```
AC = (N*N)/(T*B) = (N/T)*(N/B).
```

We calculate this since the thread IDs are given in CUDA as 3D tuple, but we needed a single integer. So, with `t` and `b`, we can identify each thread uniquely. After that, each thread iterates over only the cells they were assigned and do `O(N)` computation for that cell. So in total, a thread does `O(N*AC)` computation.

Requirements of Part 3:

1. Each cell in the matrix should be assigned to a different thread. Each thread should do `O(N *` *`number of assigned cell`*`)` computation. Assigned cells of different threads does not overlap with each other. And so, no need for synchronization. Please compile and run your modified code and report the outputs.
2. Then, profile this code and report the output. How does this version compare to your GPU code in part 2? Report.

Part 4. Optimize

The migration overhead in this simple code is caused by the CPU initializing the data, and the GPU only uses it once.

Let's try to achieve better throughput by changing our kernel's execution configuration. Then, try taking advantage of the unified memory and prefetching.

Requirements of Part 4:

1. Can you optimize the number of threads and blocks you use? Report your effort.
2. Move the data initialization to the GPU in another CUDA kernel and prefetch the data to GPU memory before running the kernel. Did the page faults decrease? Report why or why not?

Expected Submission:

You should submit a single tarball/zipped file through the Blackboard containing the following:

- All your source codes.
- Output files for each part's result/test-cases (in plain ASCII format, NOT screenshots).
- 2-3 pages report summarizing all your test results in a table format or charts (i.e., plotting these data via a tool like Excel, gnuplot, MatLab, Python, etc.) and discussing your findings/insights. There is no required format/template, so please be creative.

Grading Criteria:

Please find the detailed grading criteria in the table below. **Please note that we may require an in-person demo for grading this project.**

Part 1	
15%	Requirement 1
10%	Requirement 2
5%	Requirement 3

Part 2	
5%	Requirement 1
5%	Requirement 2
Part 3	
15%	Requirement 1
10%	Requirement 2
Part 4	
5%	Requirement 1
10%	Requirement 2
Report	
20%	Quality of the report and level of effort.

THE END.