ECE569 -Homework 3
Matrix Multiplication

This assignment will involve profiling with nvvp/nvprof. You need to consider the availability of Ocelote/ElGato when you need them. Accessing Ocelote using virtual desktop to run interactive jobs for an hour depending on the time of the day varies. If you start the assignment three days before the deadline, probability of late submission will be higher. Deadline is firm as we will move onto hw4 next.

A. Objective

The purpose of this lab is to implement two versions of the dense matrix multiplication routine.

B. Logistics

In the Assignment3 package you will find the "hw3" folder along with the "CmakeLists.txt". Place them into your "labs" folder.

C. Task 1: Instructions for basic matrix multiplication

You will implement a basic dense matrix multiplication routine for arbitrary but legitimate size inputs. Optimizations such as tiling and usage of shared memory are not required for this version. You will find the temple source code named "solution.cu" in the folder named: hw3/BasicMatrixMultiplication
Edit this file to perform the following:

- Allocate device memory
- Copy host memory to device
- Initialize thread block and kernel grid dimensions
- Invoke CUDA kernel
- Copy results from device to host
- Free device memory
- Write the CUDA kernel

The code handles the import and export as well as the checking of the solution. Instructions about where to place each part of the code is demarcated by the **//@@**. You should keep all other parts of the template code unchanged.

Compiling and Execution Instructions

You will need to compile your code from the "build-dir" folder similar to the process described earlier assignments. In your "build_dir" type the following command to compile:

```
$make
```

The makefile compiles and generates the executable "BasicMatrixMultiplication_Solution" in the "build-dir" directory. The executable generated as a result of compiling the lab can be run using the following command:

```
./ BasicMatrixMultiplication_Solution -e <expected.raw> \
       -i <input0.raw>,<input1.raw> -o <output.raw> -t matrix
```

where <expected.raw> is the expected output, <input0.raw>,<input1.raw> is the input dataset, and <output.raw> is an optional path to store the results.

The datasets can be generated using the dataset generator built as part of the compilation process.
BasicMatrixMultiplication_DatasetGenerator in the build-dir generates the data set.

```
$./BasicMatrixMultiplication_DatasetGenerator
```

then go into the MatrixMultiplication folder to access the test files.

- Dataset generator has routines to print transposed matrix for convenience.
    - If you want to test your code with small non-square matrix, printing the first one in transpose is convenient. You can then copy paste into excel and do column wise multiplication of the two inputs to check your results.
    - Refer to lines #107 and #110 in MatrixMultiplication/dataset_generator.cpp

- **When grading square and non-square matrices will be used. Refer to grading criteria.**

Example test run:
Before running your pbs script (run_hw3.pbs provided in Assignment 3) ,
create the following folder in your "build_dir" directory:

```
$mkdir BasicMatrixMultiplication_output
```

In the pbs script, you should comment out the test runs related to the TiledMatrixMultiplicaiton (lines 60-76)

Then launch your jobs
```
$qsub run_hw3.pbs
```

Outputs for each test case will be wrriten into the "BasicMatrixMultiplication_output" folder.

D. Task 2: Instructions for tiled matrix multiplication

You will implement a tiled dense matrix multiplication routine using shared memory. Before starting this task, make sure that you have completed "Basic Matrix Multiplication".

Instructions
You will find the temple source code named "solution.cu" in hw3/TiledMatrixMultiplication
Edit the code to perform the following:

- allocate device memory

- copy host memory to device
- initialize thread block and kernel grid dimensions
- invoke CUDA kernel
- copy results from device to host
- deallocate device memory

Instructions about where to place each part of the code is demarcated by the **//@@** lines.

You will need to compile your code from the "build-dir" folder. In your "build_dir" type the following command to compile:

$make

The executable generated as a result of compiling the lab can be run using the following command:

```
./ TiledMatrixMultiplication_Solution -e <expected.raw> \
  -i <input0.raw>,<input1.raw> -o <output.raw> -t matrix
```

where <expected.raw> is the expected output, <input0.raw>,<input1.raw> is the input dataset, and <output.raw> is an optional path to store the results.

The datasets can be generated using the dataset generator built as part of the compilation process.
TiledMatrixMultiplication_DatasetGenerator in the build-dir generates the data set.

$./TiledMatrixMultiplication_DatasetGenerator

then go into the TiledMatrixMultiplication folder to access the test files.

- You should test your code with all the files in the data set. You can modify the "dataset_generator.cpp" modules to test your code for various sizes.

- Dataset generator has routines to print transposed matrix for convenience.
  - If you want to test your code with small non-square matrix, printing the first one in transpose is convenient. You can then copy paste into excel and do column wise multiplication of the two inputs to check your results.
  - Refer to line #99 in TiledMatrixMultiplication/dataset_generator.cpp

- **When grading square and non-square matrices will be used. Refer to grading criteria.**

<u>Example test run:</u>
Before running your pbs script (run_hw3.pbs provided in Assignment 3) ,
create the following folder in your "build_dir" directory:

$mkdir TiledMatrixMultiplication_output

In the pbs script, make sure that lines 60-76 are uncommented from task1 before running the jobs.

Then launch your jobs
`$qsub run_hw3.pbs`

Outputs for each test case will be written into the "TiledMatrixMultiplication_output" folder.

E. Task 3: Performance Analysis

Objective: Practice running configuration sweeping experiments, data collection and profiler based performance analysis

Use the following 4 test cases generated by the "dataset_generator.cpp"
Case 1: A is 128x128, B is 128x128
Case 2: A is 100x100, B is 100x100
Case 3: A is 134x130, B is 130x150
Case4: A is 417x210, B is 210x519

For each **version** (basic and tiled matrix multiplication)
For each **test case** (4 cases)
For each **thread block** configuration of 4x4, 8x8, 16x16 and 32x32

- Collect computation time (time spent on kernels excluding time spent on memory allocations, data transfer from host to device and device to host) data.
- Fill the timing table below for each test case and thread block configuration.

Execution Time Data					
		Thread block configuration			
	Test case	4x4	8x8	16x16	32x32
Basic Matrix Multiply Kernel	Case 1				
	Case 2				
	Case 3				
	Case 4				
Tiled Matrix Multiply Kernel	Case 1				
	Case 2				
	Case 3				
	Case 4				

Suggested approach:
In the CMakeLists.txt you should turn off all debugging related flags. You may observe noisy timing for the same testcase for various runs. For this, running each test case for 10 times and taking the average should help. You will need to modify your pbs script.

For this experiment it might save you time to run through all configurations. You may also prefer to use your own timer function for computation time analysis. For both needs, there is a

template in “BasicMatrixMultiplication” folder “solution_template_with_timer_utility_without_wb_library”. The main function has all wb calls commented out except functional verification. For a given test case, once filled in with kernel code, it will run for all four thread block configurations and generate an output similar to the following:

Total execution time (ms) ??? for block size 4 x 4 matrix size of 128 x 128 and 128 x 128
{"data": {"correctq": true, "message": "The solution is correct"}, "type": "solution"}
Total execution time (ms) ??? for block size 8 x 8 matrix size of 128 x 128 and 128 x 128
{"data": {"correctq": true, "message": "The solution is correct"}, "type": "solution"}
Total execution time (ms) ???for block size 16 x 16 matrix size of 128 x 128 and 128 x 128
{"data": {"correctq": true, "message": "The solution is correct"}, "type": "solution"}
Total execution time (ms) ??? for block size 32 x 32 matrix size of 128 x 128 and 128 x 128
{"data": {"correctq": true, "message": "The solution is correct"}, "type": "solution"}

Analysis

There are two parts to cover for grading:

1. Timing trend analysis, performance comparison based on timing table given above
2. Performance analysis of basic and tiled implementations with nvvp

Analysis Part-1 Timing Trend Analysis:

Just to get you started here are some suggestions:

- Fix input size, discuss execution time trends with respect to change in thread block configuration for basic and tiled solutions individually
- Discuss execution time trends among cases 1 and 2 (two square matrices)
- Discuss execution time trends among square and non-square matrices
- Discuss performance of basic and tiled solution versions

The list above is a starting point only. You should expand on any other trend you see relevant.

Analysis Part-2 Profiling based analysis:

Using only Case4 (A is 417x210, B is 210x519 or larger matrix size of your choice not divisible by tile size) expose inefficiencies in basic and tiled solutions. What are the improvements observed based on your profiling data for the tiled solution compared to the basic solution? What are the performance issues reported by nvvp/nvprof regarding the execution of the tiled solution? Note that, some of those issues will be covered later but it is important to go through the guided analysis using nvvp. for profiling.

<u>Perform memory, compute and latency analysis</u>

a. Execution time
b. Memory bandwidth utilization: Traffic to/from each memory subsystem relative to peak (L1/shared memory, L2, device memory)
c. Compute resource utilization: divergent branches, warp execution efficiency,
    i. Occupancy analysis

ii. upper bound analysis based on threads/block, shared memory usage/block, register usage per thread,
iii. Achieved occupancy – (active warps/active cycles)/ Maximum warps per SM
iv. Warps per SM with respect to varying block size (threads per block)
v. Instruction and memory latency

Support your claims referring to several performance metrics related to memory, thread behavior, utilization and efficiency factors. You should connect your arguments with the nature of the input data (square, non-square, matrix dimension). Refer to the template given in the "Content➔Demo➔Profiling" for setting up hw3 for profiling.

INCLUDE YOUR ANALYSYS HERE
Analysis Part-1 Timing Trend Analysis:

Analysis Part-2 Profiling based analysis:

F. Submission Instructions

It is critical that you follow the following steps for submission. Please refer to penalty points.

1. Rename your solution.cu files from parts C and D as **basic_solution.cu** and **tiled_solution.cu** for global and tiled versions respectively**.**

    a. **Make sure that these are the versions implemented using the solution templates given for each in the BasicMatrixMultiplication (solution.cu) and TiledMatrixMultiplication (solution.cu) folders. Testing and validation process relies on the wb library.**
    b. **"**solution_template_with_timer_utility_without_wb_library.cu" is provided for helping you with automating the testing of block size variations. It is a suggested method. You can use any method you prefer. **Do not** submit any code built based on this template.

2. Create a folder and name it as your "net id".
    a. Include **basic_solution.cu** and **tiled_solution.cu** in the **"netid"** folder

3. Zip your **netid** folder **(netid.zip)**

4. Create a **hw3_netid** folder
    a. Include **netid.zip**
    b. Include **ECE569_hw3.docx** or **ECE_hw3.pdf** file with part E (analysis)

5. Zip **hw3_netid** folder

6. Submit **hw3_netid.zip** to the designated D2L hw3 folde**r.**

G. Grading

1.	basic_solution.cu coding style and commenting	10pts
2.	tiled_solution.cu coding style and commenting	20pts
3.	ECE569_hw3.docx file with part E	100pts
4.	basic_solution.cu (private test cases)	50 pts
5.	tiled_solution.cu (private test cases)	120 pts

Total 300 points

Penalty Conditions:

- 20% penalty – incompatible submission (refer to F)
- 20% penalty per day for late submission