HIGH PERFORMANCE COMPUTING WITH COMMODITY HARDWARE
LAB 3

Kshitij Srivastava
UD ID - 702359111

Abstract: The main aim of this lab is to perform matrix multiplication on GPU using CUDA and to satisfy various conditions as given in different parts.

Machine Specification: We are using university server which has two Tesla K40c GPUs. Some of its specifications are:

1. Name: Tesla K40c
2. Max threads / block: 1024
3. Max thread size: 1024 * 1024 * 64
4. Shared mem /block: 48KB
5. Constant memory: 64KB
6. Wrap size: 32
7. Registers / block: 65535
8. Clock rate: 745MHz

Compiling and running the program:

The following method explains how the codes were run on server. Method was similar for all the questions, therefore, I am explaining it for just one part.

1. After logging into mlb.acad.ece.udel.edu, we were supposed to ssh into cpeg655.ece.udel.edu
2. After this, in order to compile the code, I used: `/usr/local/cuda/bin/nvcc -o temp device.cu –lcuda`
3. In order to run the code, I used: `srun -N1 --gres=gpu:1 temp`

So the above code executes device.cu file which extracts all the properties of the GPU and runs it. Similar approach was used for all other files.

PROBLEM 1 A: To compute matrix multiplication of C = A * B where all the matrices are of size N * N. One thread computes one element; we have to use only one thread block and we were supposed to run it for N =16 and N=22.

Running Time\ N	16	22
(ms)	0.027392	0.030592

Version with higher number of elements will take more time to complete as can be seen in the table above.

PROBLEM 1B: Extension of Part 1A where one thread still computes one element of the product but a thread block computes only a tile of matrix C. N = 512 or N = 1024 and tile size = 8 or 16.

N \ TILE_SIZE	8	16
512	4.2751(ms)	2.8260(ms)
1024	35.0677(ms)	22.5628(ms)

Problem 1C: The non-tiled version is that version where every element computes one element if the product matrix and so a thread block is fully utilized in this case. In part1b, although one thread still calculates one element of product matrix but a thread block only calculates a subset or tile of matrix C, thus, a thread block is not being utilized to its full capacity and therefore, running time of tiled version will be greater than non-tiled version in most of the cases if not optimized according to the matrix.

As far as the difference between tile size is concerned, it can be seen from the table above, as soon as tile size was increased, more elements were calculated by a single thread block which increased its efficiency.

Problem 2A: We were supposed to extend version 1b where one thread computes NB*NB tile matrix. We were supposed to find the best possible combination of NB and NT such that $N^2 = NB^2 * NT^2 * NK$ for N = 1024

For this part, I ran various combinations of NT and NB and the best timing results were achieved when NT = 64 and NB = 4 with an elapsed time of 0.001005ms

Theoretically, to achieve maximum performance, if each thread computes less number of elements, then the computation time required by each thread is lesser, and hence we have a total lesser time provided the thread block contains as many threads as possible.

To achieve average running time, I ran the loop for 5 times and divided the total time elapsed by 5. I have mentioned this in the code.

A total of 21 different combinations were used to calculate the best combination.

Problem 2B: we were supposed to unroll the inner most loop by a factor of 2 and 4.

Code\Unlooping factor	2	4
NB = 4, NT = 64	0.002650ms	0.002419ms

Theoretically, unlooping helps in reducing the processing time of matrix because it reduces the overhead which comes into play when we have to check the for loop on every iteration. So unlooping with a factor of 2 reduces the overhead time to 50% whereas unlooping with a factor of 4 reduces the overhead time to 25%. Please note that this time is not calculated on the total time required to run the program but it is calculated only when the program checks for loop condition during an iteration.

We can see unlooping with a factor 4 gives us a reduced time with respect to factor 2.

Problem 2C: We were supposed to load the matrix to shared memory instead of directly accessing it from the global memory and to see the effect of memory bank conflict (half-warp write/read 4 banks)

The aim of this exercise is to see how program works with and without bank conflict. Local memory in a GPU is divided into memory banks. Each bank can only address one dataset at a time, so if a half warp tries to load/store data from/to the same bank the access has to be serialized (this is a bank conflict).

Shared memory on GPU is 48 KB. Each integer takes 4B. Total number of numbers it can have is 48KB/4B = 12,000. However, since it is a 2D array, number of rows and columns it can have is sqrt(12000) = 109.

So that's why in order to successfully load data into shared memory, I have used NB=1 and NT=64 because any other combination would take the value to above 109.

Time(ms)\Memory bank	Without conflict	With conflict
	0.008160 ms	0.008544ms

As it can be seen clearly, time taken by program with memory bank conflict is greater than time taken by the program without any conflict. This happens because whenever there is a conflict, memory access becomes sequential which affects running time.