600086 Lab Book

Week 5 – Lab B

Date: 3rd March 2022

Exercise 1. A simple matrix multiplication program in CUDA using one thread block

Question1 : implement a solution in C++ to the problem

Solution:

1. Created a C++ solution that iterates through the required combinations. This is done using a series of for loops. Taking the dimensions of the final array C and iterating through the x and y values as shown in the code below.

```
void ConcurrentMatrixMultiplication(int* c, int* a, int* b, int common, int W, int H)
{
    for (int x = 0; x < W; x++)
    {
        for (int y = 0; y < H; y++)
        {
            int ARow = y * common;
            int BColumn = x;
            int CIndex = x + (y * W);

            printf("Thread ID: (%d,%d)\n{\n AIndex: %d\n BIndex: %d\n CIndex: %d\n}\n", x, y, ARow, BColumn, CIndex);

            for (int i = 0; i < common; i++)
            {
                c[CIndex] += (a[ARow + i]) * (b[i * W + BColumn]);
            }
        }
    }
}
```

The commented-out printf for debugging purposes and produces the following output

```
Thread ID: (1,1)
{
 AIndex: 2
 BIndex: 1
 CIndex: 3
}
```

This allowed me to monitor the index values for finding the various matrix index value based on the threadIdx values. And ensure the offset values matched the expected coordinates.

Test data:

$$Matrix\ A \begin{Bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{Bmatrix} \quad Matrix\ B \begin{Bmatrix} 10 & 20 \\ 30 & 40 \\ 50 & 60 \end{Bmatrix}$$

Sample output:

```
Array C = {220,280,490,640,760,1000,1030,1360}
```

Question2 : implement a solution using GPU processing to solve the problem

Solution:

1. Modified the Add kernel Function to multiply the matrixes as follows

```
// represents the Common Dimension between the two arrays
__global__ void OneBlockMatrixMultiplication(int* c, int* a, int* b, int common, int cW)
{

    int ARow = threadIdx.y * common;
    int BColumn = threadIdx.x;
    int CIndex = threadIdx.x + (threadIdx.y * cW);

    //printf("Thread ID: (%d,%d)\n{\n AIndex: %d\n BIndex: %d\n CIndex: %d\n}\n", threadIdx.x, threadIdx.y, ARow, BColumn, CIndex);

    for (int i = 0; i < common; i++)
    {
        c[CIndex] += (a[ARow + i]) * (b[i* cW + BColumn]);
    }
}
```

2. Added two new passed in variables: “common” that represents the common dimension of the matrices to make calculating the indexes of the arrays easier to manage and “cW” which represents the width of the B array to allow for offset calculations.
   the for loop iterates through and calculates the dot product of the current row and column this could be extracted into a further kernel function to speed up the process.

   The commented-out printf for debugging purposes and produces the following output

```
Thread ID: (1,1)
{
 AIndex: 2
 BIndex: 1
 CIndex: 3
}
```

   This allowed me to monitor the index values for finding the various matrix index value based on the threadIdx values. And ensure the offset values matched the expected coordinates.

Test data:

$$Matrix\ A \begin{Bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{Bmatrix} \quad Matrix\ B \begin{Bmatrix} 10 & 20 \\ 30 & 40 \\ 50 & 60 \end{Bmatrix}$$

Sample output:

```
Array A = {1,2,3,4,5,6,7,8,9,10,11,12}
Array B = {10,20,30,40,50,60}
Array C = {30,70,50,110,70,150,90,190}
```

Reflection:

This was fairly simple the most difficult part was collapsing the Thread in the right way for the X and Y values as I kept confusing the two and doing them the wrong way round. However as I had looped via x and y dimensions for the concurrent method modifying the code to run on GPU in parallel allowed me to sub in the threadId.x and threadId.y for my x and y variables.

Metadata:

Further information:

Is it possible to launch a kernel from a kernel if so this could improve the process?

Exercise 2. Compare the performance of the CUDA solution against the CPU solution

Question1 : Running performance test on a range of square arrays for CPU solution and GPU solution comparing the results

Solution:

N/A

Test data:

Matrix A = Matrix B

Matrix B will be a square matrix of the following sizes

(8*8),(32*32),(256*256),(512*512),(1024*1024)

Sample output:

Matrix dimensions	CPU	GPU
8 x 8	Execution of kernel: 0.018720ms	Execution of kernel: 0.082272ms
32 x 32	Execution of kernel: 0.020480ms	Execution of kernel: 0.114464ms
256 x 256	Execution of kernel: 44.437504ms	Execution of kernel: 0.018784ms
512 x 512	Execution of kernel: 417.548157ms	Execution of kernel: 0.055744ms
1024 x 1024	Execution of kernel: 3544.644775ms	Execution of kernel: 0.020800ms

Reflection:

The CPU only solution seemed faster initially however it quickly began to take more and more time whereas the GPU solution kept a fairly consistent execution time the primary bottleneck was printing the results to the screen however another bottleneck was the iteration on each thread this could be replaced by a nested kernel call however I was unsure if this was possible.

Metadata:

Further information:

Is it possible to launch a kernel from a kernel if so this could improve the process?