[DD2360] Applied GPU Programming
Assignment II: CUDA Basics I

Group 8
Martin Forslund (uz6@kth.se)
Valeria Grotto (vgrotto@kth.se)

Link to github repo: `https://github.com/MoggmentuM/DD2360`

Contributions

Martin Forslund: Did exercise 2 (with some help from Valeria)
Valeria Grotto: Did exercise 1 (with some help from Martin)

1 Exercise 1 - Your first CUDA program and GPU performance metrics

1.1 Explain how the program is compiled and run.

The program is compiled by using nvcc to compile it. And in order to run it, if the program is called vectorAdd, then just enter ./vectorAdd <parameter>

where <parameter> is the size of the vector.

1.2 For a vector length of N:

1.2.1 How many floating operations are being performed in your vector add kernel?

When adding a two vectors, they have the length N and there are N amount of addition floating point operations that are being perfomed.

1.2.2 How many global memory reads are being performed by your kernel?

Since both vectors are of length N, it will be 2*N amount of reads.

1.3 For a vector length of 1024:

1.3.1 Explain how many CUDA threads and thread blocks you used.

For the amount of threads blocks used: (1024 + 32 - 1)/32 = 32. The amount of threads used were 32 * 32 = 1024 threads.

1.3.2 Profile your program with Nvidia Nsight. What Achieved Occupancy did you get?

I got 3.12% in achieved occupancy.

1.4 Now increase the vector length to 131070:

1.4.1 Did your program still work? If not, what changes did you make?

Yes, it still works.

1.4.2 Explain how many CUDA threads and thread blocks you used.

For the amount of threads blocks used: (131070 + 32 - 1)/32 = 4096. The amount of threads used were 4096 * 32 = 131072 threads.

1.4.3 Profile your program with Nvidia Nsight. What Achieved Occupancy do you get now?

The achieved occupancy received was 32.55%.

1.5 Further increase the vector length (try 6-10 different vector length), plot a stacked bar chart showing the breakdown of time including (1) data copy from host to device (2) the CUDA kernel (3) data copy from device to host. For this, you will need to add simple CPU timers to your code regions.

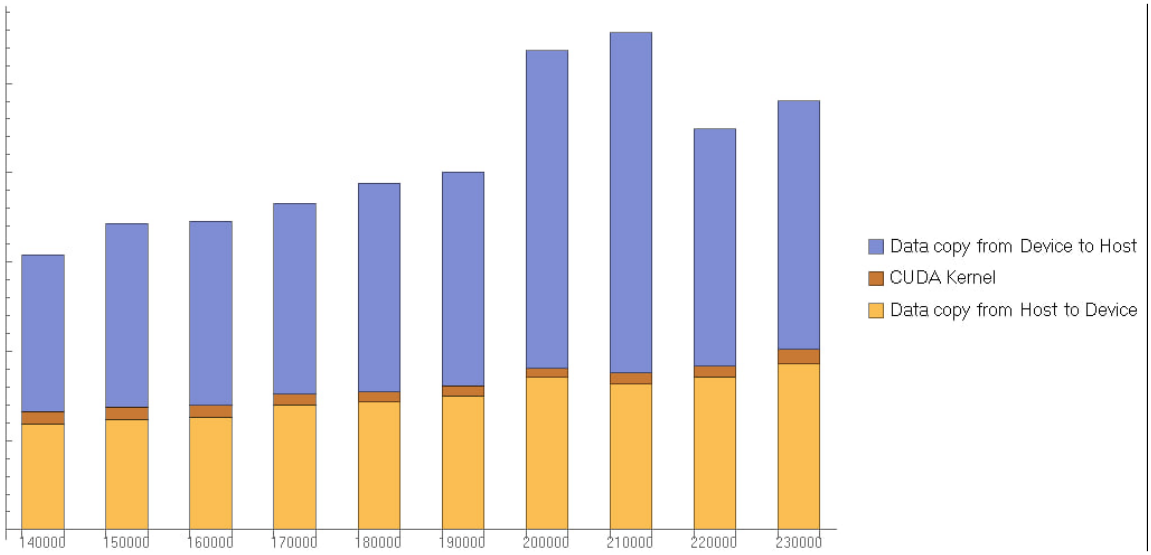

Figure 1: Stacked barchart with different lengths from 140000 to 230000

2 Exercise 2 - 2D Dense Matrix Multiplication

Name three applications domains of matrix multiplication

1. One typical application of matrix multiplication is for rendering 3D graphics.
2. Another typical application is solving linear systems.
3. The last typical application is machine learning (ML).

2.1 How many floating operations are being performed in your matrix multiply kernel?

In each iteration of the innermost loop, a single floating-point multiplication and addition operation are performed. The innermost loop iterates numAColumns times. Therefore, the total number of floating-point operations is given by:

Total floating-point operations = numARows * numBColumns * numAColumns

This is because for each element in the resulting matrix, we perform one multiplication and one addition operation.

2.2 How many global memory reads are being performed by your kernel?

The global memory reads are being calculated by:
Total global memory reads = 2*numARows*numBColumns*numAColumns

2.3 For a matrix A of (128x128) and B of (128x128):

2.3.1 Explain how many CUDA threads and thread blocks you used.

For a matrix A of size (128 x 128) and matrix B of size (128 x 128), the CUDA kernel in the provided code is configured to use the following:

- Block Size: 1024 threads per block (32 * 32 threads) - Grid Size: 16 blocks in the x and y dimensions

The total number of CUDA threads is given by multiplying the block size by the grid size:

Total Threads = Block Size * Grid Size

Given that the block size is 1024 and the grid size is 16:

Total Threads = 1024 * 16 = 16384 threads

So, the kernel in the provided code is configured to use a total of 16 blocks, each containing 1024 threads, resulting in 16384 CUDA threads for the matrix multiplication of two (128 x 128) matrices.

2.3.2 Profile your program with Nvidia Nsight. What Achieved Occupancy did you get?

I got 95.89% on Achieved Occupancy.

2.4 For a matrix A of (511x1023) and B of (1023x4094):

2.4.1 Did your program still work? If not, what changes did you make?

Yes, it still worked.

2.4.2 Explain how many CUDA threads and thread blocks you used.

Number of thread blocks: 1024

Number of CUDA threads: 1024 blocks * 2048 in grid size = 2097152 threads.
Total Threads = 1024 * 2048 = 2097152 threads

2.4.3 Profile your program with Nvidia Nsight. What Achieved Occupancy do you get now?

Achieved Occupancy given was 98.01%.

2.5 Further increase the size of matrix A and B, plot a stacked bar chart showing the breakdown of time including: Data copy from host to device. The CUDA kernel execution. Data copy from device to host.

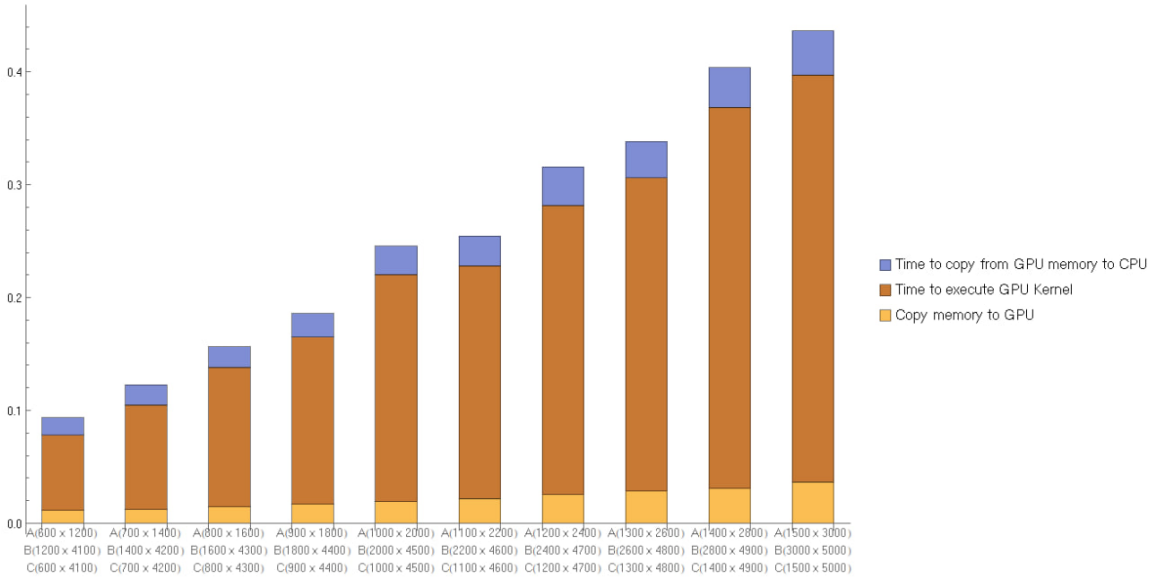

Figure 2: Stacked barchart with different sizes of matrices from 600x1200 to 1500x3000

From looking at the graph, we can observe that even if it grows linearly, it increases what more on the even iterations (iteration 2, 4, 6, 8, 10) as it looks most clear in iteration 8 compared to iteration 7. In addition, the thing that increases the most in each iteration is the GPU kernel time while the copies doesn't increase as much, which is most noticable with copying to GPU.

2.6 Now, change DataType from double to float, re-plot the stacked bar chart showing the time breakdown.

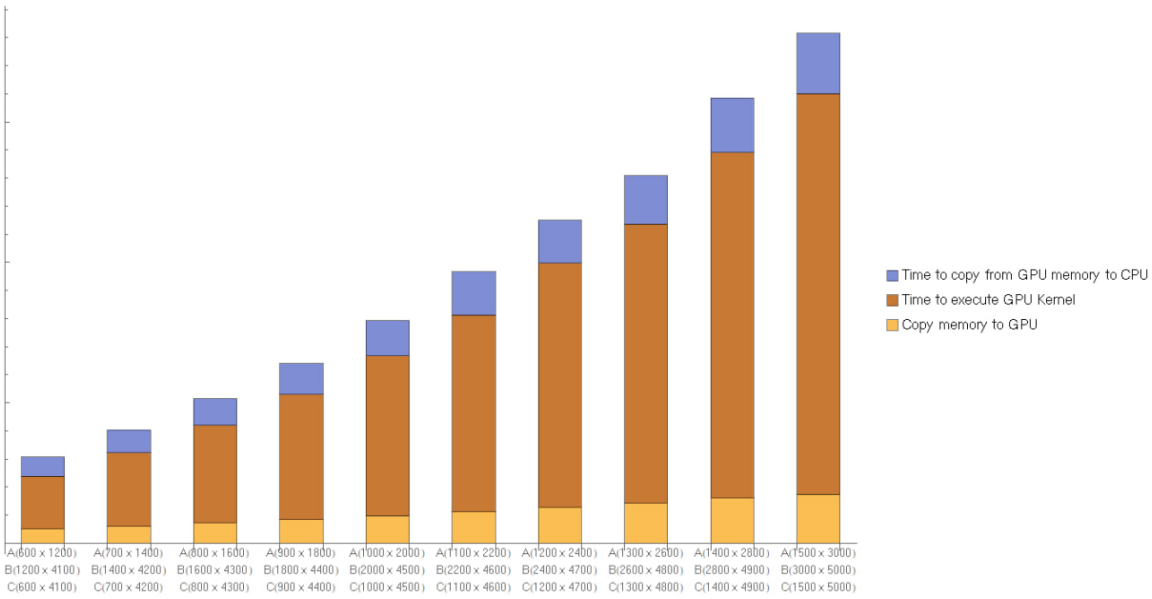

Figure 3: Stacked barchart for float datatype with different sizes of matrices from 600x1200 to 1500x3000 (increase by 100x200 in each iteration)

From observing the graph for float, we can see that the execution is way lower compared to the the graph that uses double. It can also be observed in this one that it linearly increases more compared to the double which was a bit inconsistent with the timings.