Programming Assignment 4: Report

Rajarshi Biswas
biswas.91@osu.edu
section: ab

Table of Contents

Part1 (Multiply transpose of a Matrix with itself)

GigaFlops:

The matrix size used in my program was 1024 x 1024.
Serial and CUDA both performed a total of (1024 x 1024 x 1204 x 2) = 2,147,483,648 number of floating point operations.
Gigaflops for serial version is = 2147483648 / (10^9 x 6.24352) = 0.347
Gigaflops for CUDA version is = 2147483648 / (10^9 x 0.321237) = 6.685

Performance:

Size of the matrix: 1024 x 1024
Time taken by the serial version: 6.24352 seconds
Time taken by the CUDA version: 0.321237 seconds
Hence I observed approximately 19 times performance improvement when the program ran on GPU.

CUDA compute structure:

In my code each thread processes a single cell in the matrix. We can spawn maximum 1024 threads in each block. Therefore, in my program each block has (32,32) threads in 2 dimensions. Hence, the total number of

threads in each block is 32 x 32 = 1024. In addition, each grid has the input matrix dimension (in this example 1024) divided by the thread number (in one dimension, i.e. 32), of blocks in both x and y plane. Hence blocks per grid (1024/32, 1024/32). Below is the code snippet of the CUDA compute stricture.

```
#define SIZE 1024
#define THREADS_X 32
#define THREADS_Y 32
dim3 threadsPerBlock(THREADS_X, THREADS_Y);
dim3 blocksPerGrid(SIZE/THREADS_X, SIZE/THREADS_Y);
```

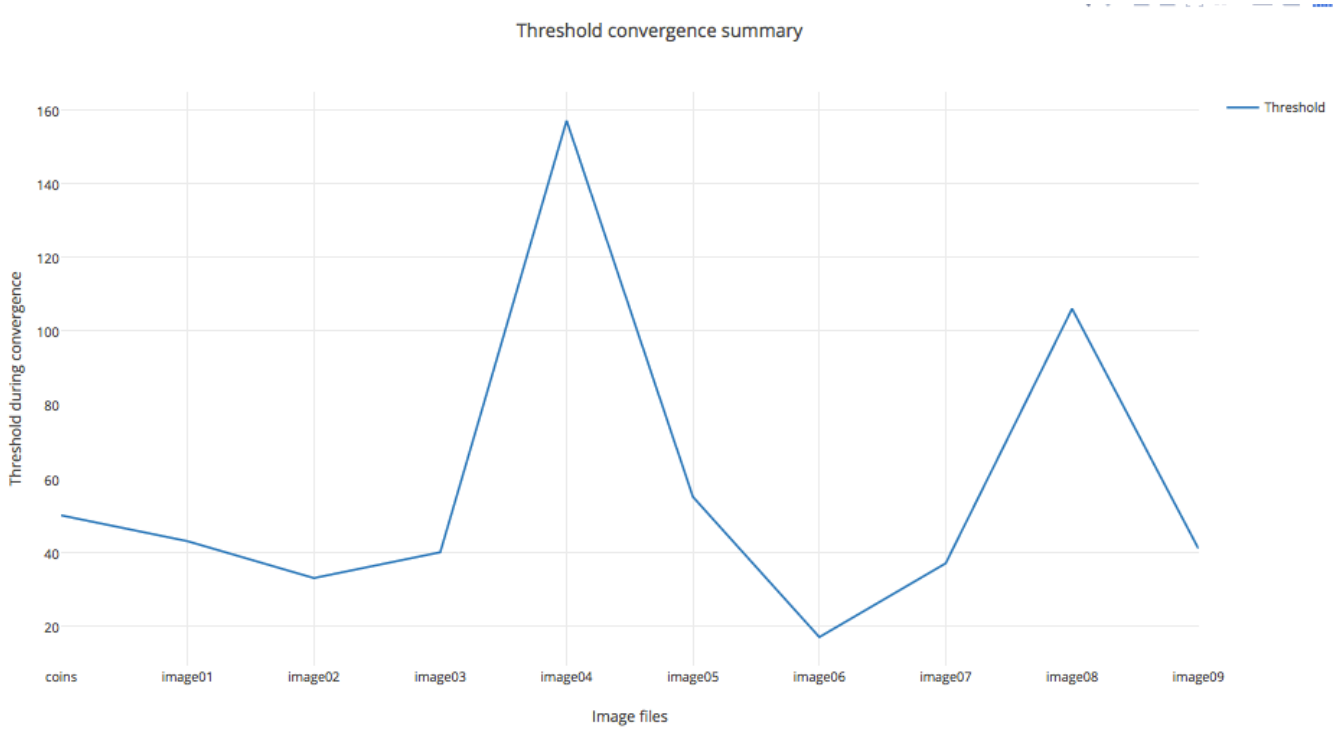
Hence, for matrix size of (1024 x 1024) my program will have:
Threads per block = (32,32)
Blocks per grid = (32, 32)
Number of grid = 1

Therefore, total number of threads spawned is 32 x 32 x 32 x 32 = 1,048,576. i.e. same as the total number of elements (1024 x 1024) in the matrix.

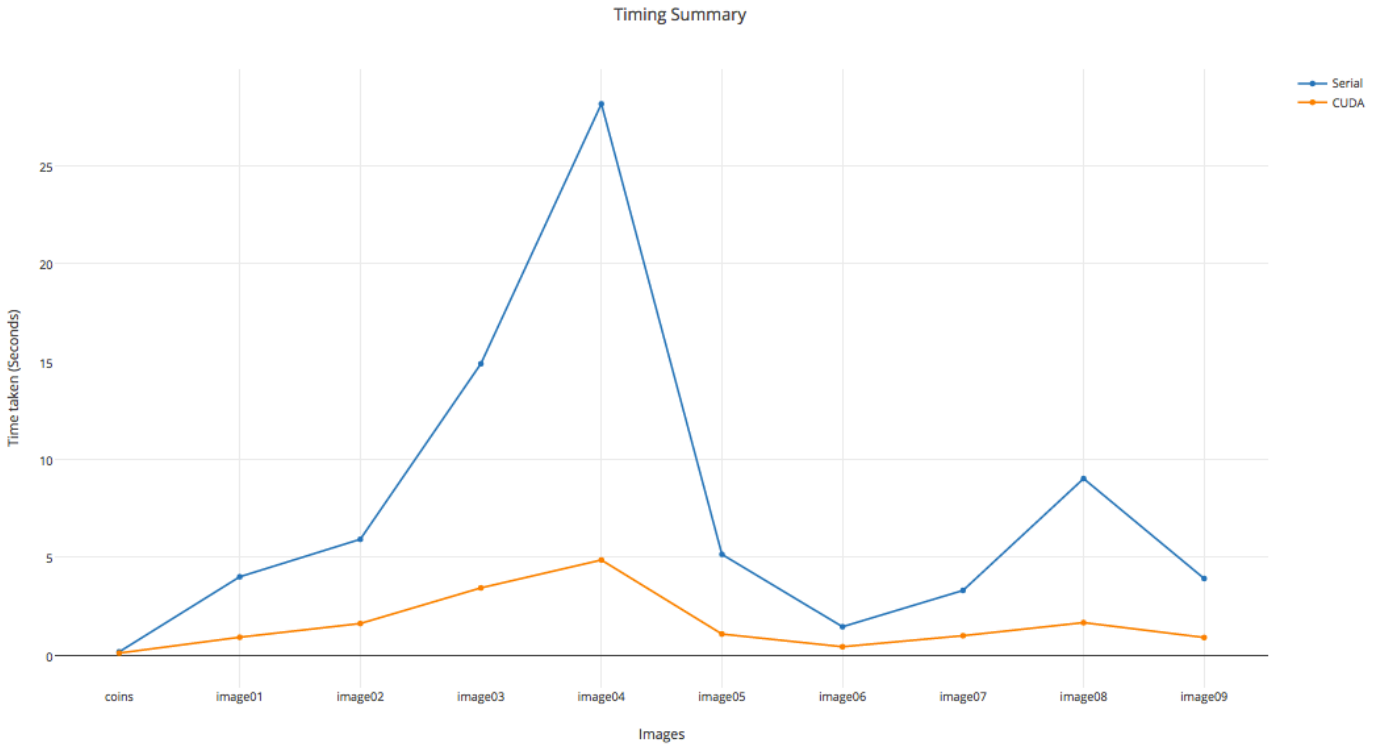
Part2 (Sobel operation)

Threshold convergence summary

The below graph shows the threshold convergence summary for each image. Both the serial and CUDA version obtained same threshold during convergence for all the images.



Timing summary



CUDA organization

Every thread computes for one pixel. Each block has threads (32,32) in 2 dimensions. Hence, the total number of threads in each block is 32 x 32 = 1024. The blocks in a grid are also organized in 2 dimension. Now, to calculate the required number of blocks in a grid, I have divided the width and height of an image by the thread number (32) in one plane + 1. Added one, to compensate if the image's width and height is smaller than 32. In case of more number of threads than pixels, the kernel function simply returns without any computation for the additional threads. Below is the code snippet from my program:

```
#define THREADS_X 32
#define THREADS_Y 32
dim3 threadsPerBlock(THREADS_X, THREADS_Y);
dim3 blocksPerGrid((ht/THREADS_X) + 1, (wd/THREADS_Y) + 1);
```

Performance improvement

Yes, I have seen a significant performance improvement when using GPU. For example, for processing Image04.bmp, the serial version takes approximately 28 seconds, whereas CUDA version takes only around 5 seconds. Hence, around 5.6 times performance improvement observed.