COMP 429/529:
Parallel Programming
Assignment 2

Project (Image Denoising)
Report

Group members:

Mustafa Saraç

Mustafa Mert Ögetürk

Introduction

In this project, we completed all the desired versions of the assignment. First, we built the naive version of the project, it consists all the parallelization operations on the nested loops in the project. It provided a huge performance increase to us. Then, we optimized the code by using temporary variables to take the advantage of data reuse. This technic resulted a considerable performance increase with the parallelization. Because elements of the array which will be used in the computation over and over, is taken from the register all the time not from the memory. For the last version, we created different two dimensional arrays with the size of block-size in both dimension. Then all the threads started using the shared data for theirs block. This technic did not result with an observable performance increase. It is hard to say that using shared data provides a significant speed up for this project. We thought that the data, which we do the operations on them, are relatively small. Technically, we could see a speed up but it was not observed. We started by analyzing the code and determining the parts which can be parallelized. In the image denoising project, there are three computation parts which include different nested for loops, this indicates that we can divide the work that is done for each index of the pixels array to the Cuda blocks and threads.

Version 1 (Naive implementation)

In this version, we aimed to parallelize the algorithms using a single GPU by making all the references through global memory. First, we started by setting the computation methods as it will use the global memory by adding "__global__" command according to Cuda. The second step was creating new arrays and variables for the device, because we will make these computations with these new arrays and variables which will point to the memory in the global device. Then, we allocated memories for them by using the command "cudaMalloc". After that, we copied these arrays from the host to the allocated memory in the device. In the computation parts, index numbers are replaced by the global thread IDs. To do that, we use the fundamental total data computation of Cuda which is "blockIdx.a * blockDim.a + threadIdx.a" where "a" is a dimension. Each of these threads will not wait for others to start because they will work on their

own part when the computation is started. Therefore, this naive implementation speeded up the computation process when it compared to the serial implementation.

Reduction part is quite problematic and to reach the correct solution we use NVidia's final optimized version as reference. To adapt the algorithm for the summation part in the project, first we create a shared segment array. Then, after the computations, we wrote it back to the sum_d and sum2_d arrays in the device.

Please view the graphs and examine the performance differences between version_1 and serial version at the end of the text.

Version 2 (Using Temporary Variables to Eliminate Global Memory References)

In computation parts, normally algorithm uses the global memory to reach the array values, and this causes a lot of unnecessary memory transfers and a lot of time waste. To solve this problem, we define some temporary variables and write the array values on it. Therefore, the temporary value's memory was allocated in the register and whenever the algorithm uses it, takes the value from the register not from the memory. This optimization resulted with a considerable speed-up.

Please view the graphs and examine the performance differences between version_1 and version_2 at the end of the text.

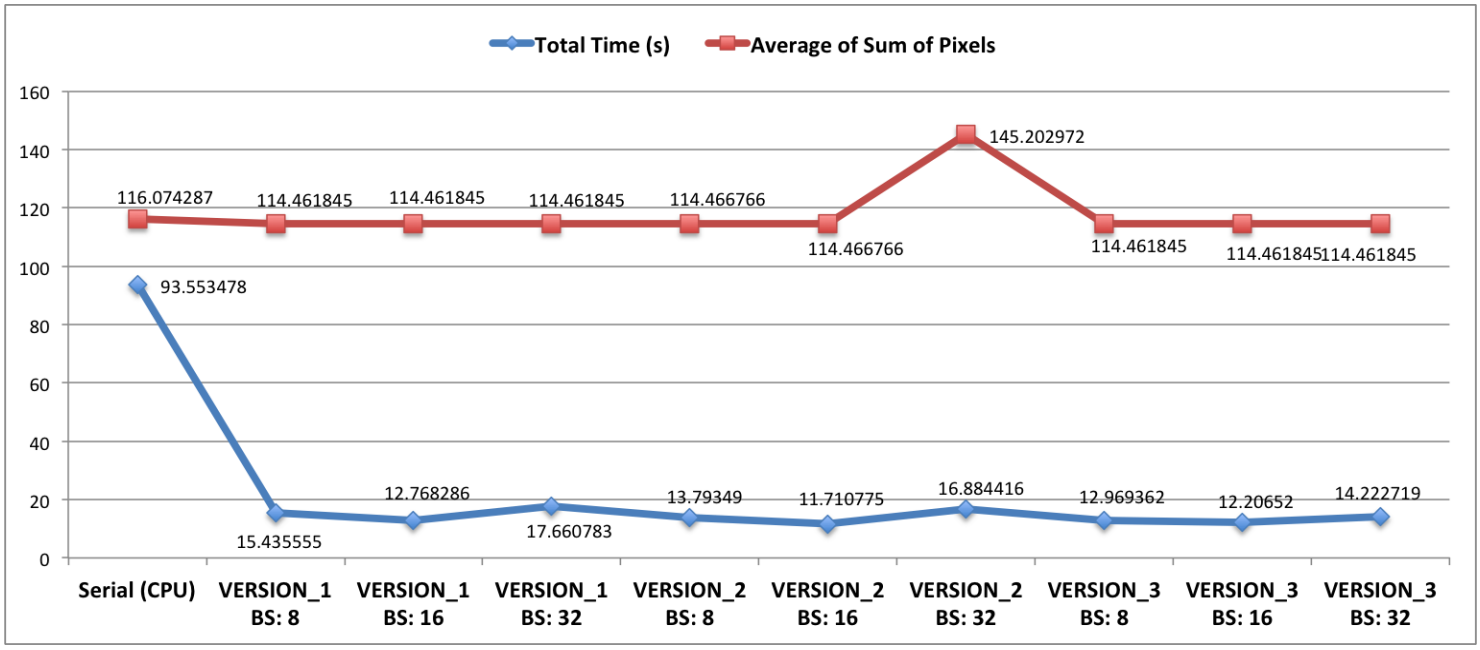
Version 3 (Using Shared Memory on the GPU)

In this part, we aimed that using the shared memory of each Cuda block for each thread in it instead of using the global memory for all the threads and we expected a speed-up because each thread will take the data from their blocks shared memory and this was going to prevent the time lose. To do that, first we created new shared two dimensional arrays for each array in the computation because each thread will work with data in both dimension. After that, we transfer the global arrays to the shared arrays according to the algorithm which is already given in the

computation parts. And with the same logic in the version_2, we wrote the array values to the related temporary values which we define and apply the computations. Lastly, when each thread finishes their work, we wrote the results to the defined arrays in the device back.

Please view the graphs and examine the performance differences between version_1 and version_2 below.

Total time and average of sum of pixel graph

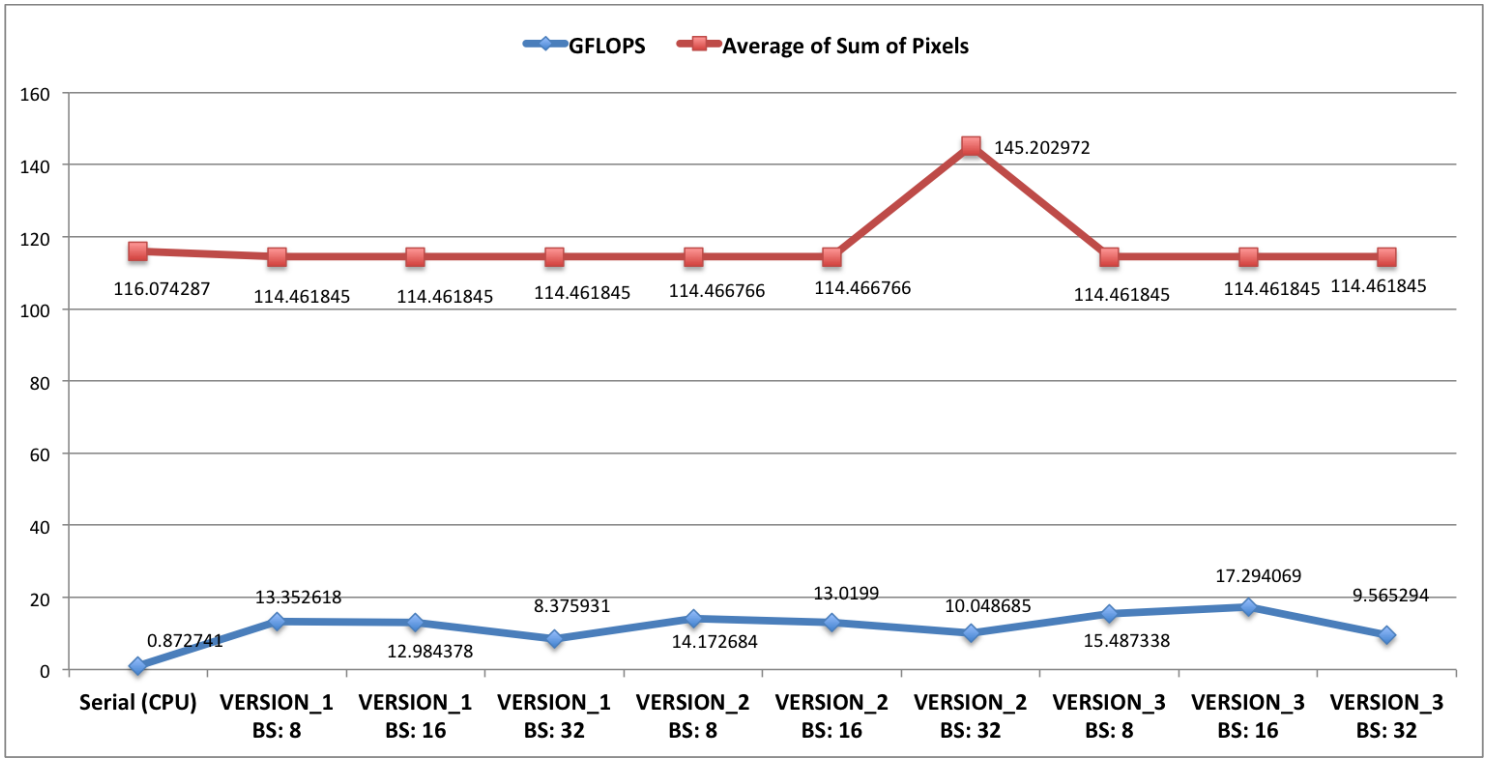


(BS refers to "BLOCK_SIZE")

When we look at the graph, it is obvious that algorithm is functioning well, when block size is equal to 16. Considering this, we have achieved;

- ~7.3x speed up between serial version and parallel version_1.
- ~1.1x speed up between parallel version_1 and parallel version_2 (optimization).
- Non-observable speed up between parallel version_2 and parallel version_3.

GFLOPS and average of sum of pixel graph



(BS refers to "BLOCK_SIZE")

Conclusion

We had a chance to experience and apply different parallelization and optimization technics in this assignment. At the beginning, our main concern was understanding the concepts of Cuda the fundamental parallelization methods over this programming model. After that, we applied these optimizations through over the image denoising calculations.

We have realized that practicing the parallelization concepts on image denoising algorithm is very suitable, because it requires high amount of calculations. We expected a performance increase for each parallelization version. Due to the amount of the data we could not observe a significant improvement in performance in the last version. Cluster mechanism is also an important reason for the situation, it works with different run times all the time.