ECE60827 CUDA ASSIGNMENT 2 REPORT

Median Filter

a) CPU implementation

The implementation of Median Filter in the CPU had a black border around it, so I assumed that the implementation required the boundary conditions to be added. After adding the boundary conditions I tested the implementation with filter sizes 4*4 and 7*7. The execution time was 1.06172 and 4.129112 seconds respectively. The execution time is checked only for median filter function. It does not involve time taken to read the file.

The results of the implementation tested with 4*4 and 7*7 filter can be seen below:

4*4 filter results – Execution Time: 1.06172 seconds

7* 7 filter results – Execution Time: 4.12912 seconds

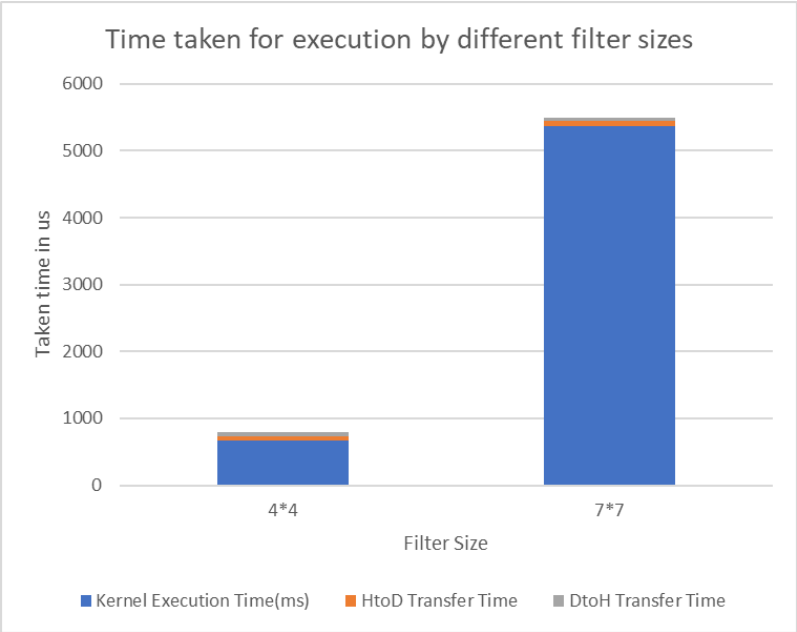
b) GPU implementation

In order to improve the execution time of the filter I implemented the median filter using thread block dimensions (16,16, 3). The reason why I chose (16, 16, 3) was because after querying the device I found out that the device cannot handle more than 1024 threads in total. The grid of blocks completely covers the image, ie, there is one thread for each of the pixels in the image. Each of these threads puts its neighboring pixel value in an array called window, sorts the array using bubble sort and computes the median of the window. The implementation is again tested with a 4*4 filter size and 7*7 filter size. The execution times were seen to be 660.51us and 5.3710ms for median filter kernel.

A few artifacts are observed on the edges of the images.The resultant images can be seen below:

4*4 filter results – Execution Time: 660.51us

7*7 filter – Execution Time: 5.3710ms

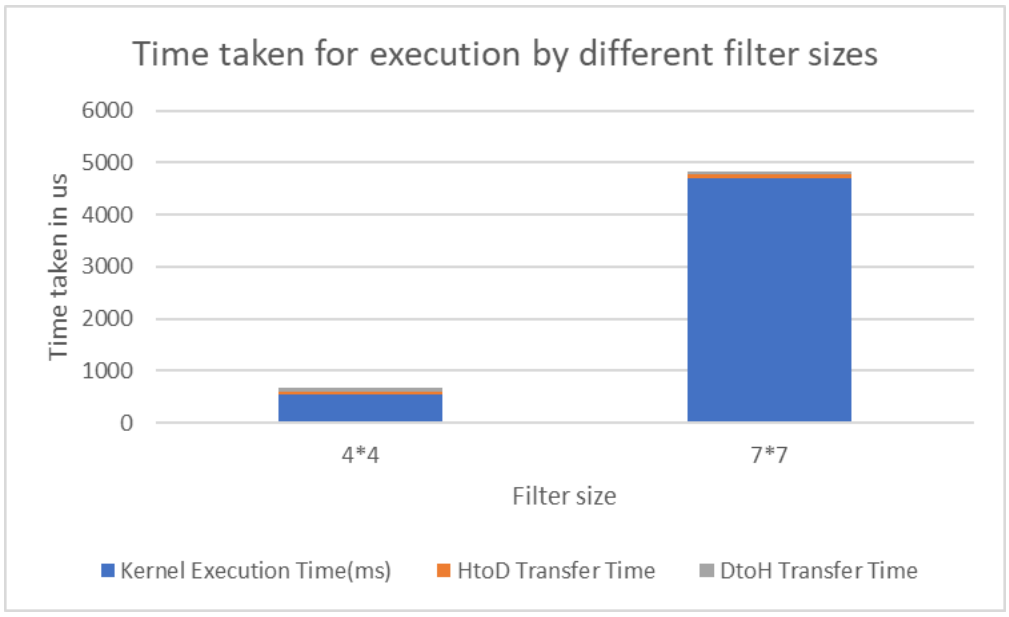


The graph above shows the execution and memory transfer time taken in case of 4*4 and 7* 7 filter. It can be clearly observed that the bottleneck is the median filter kernel itself and not the data transfer between host and device.

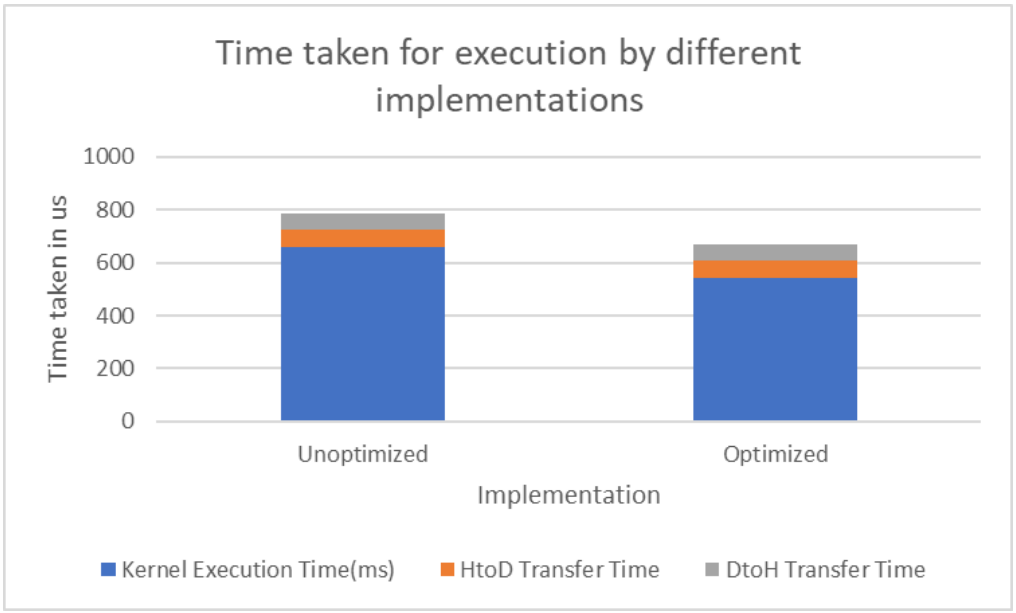
c) Optimized GPU Implementation

My first approach to the optimization of the Median Filter was to load the image into the shared memory in Tiles. The idea was to load BLOCK_SIZE + MAX_WINDOW_WIDTH -1 * BLOCK_SIZE + MAX_WINDOW_WIDTH -1 input image pixels using BLOCK_SIZE * BLOCK_SIZE threads for each of the 3 channels. This implementation can be seen in the shared_mem branch. The implementation, however, failed because I kept getting artifacts around the edges of individual blocks. The output for 4*4 filter can be seen below:

The execution times for this implementation can be seen in the graph below:



The graph below shows the execution time taken by unoptimized implementation and the shared memory implementation. It can be observed that the execution time is comparable to the unoptimized version, so I figured there is a need for another approach.



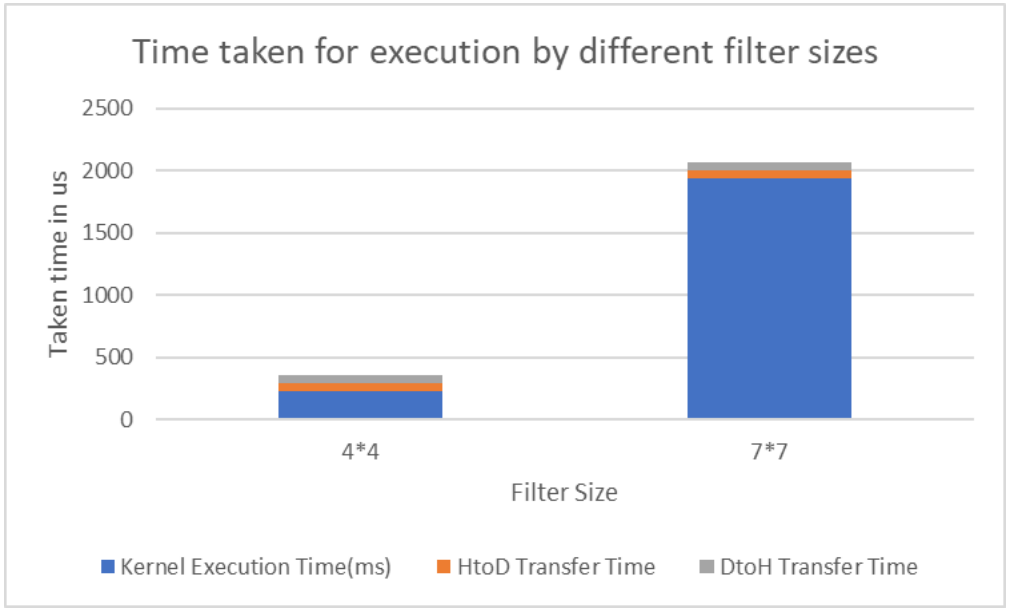
The median filter kernel is later finally optimized by storing windows in the shared memory. Each thread in a block stores its window array in the scratch which facilitates reuse when it comes to sorting. The sorting requires the elements of the window array to be accessed again and again. Therefore, putting the window in the shared memory helped execution a lot. The execution time taken in case of 4*4 and 7*7 filters decreased from the unoptimized implementation to be 230.14us and 1.9365ms respectively.

The optimization uses 36.750KB of the static shared memory with Max width and height of filter as 7. Allowing larger filter sizes than 7*7 causes out of shared memory issues. Hence, the implementation is limited to the maximum of 7*7 filter.

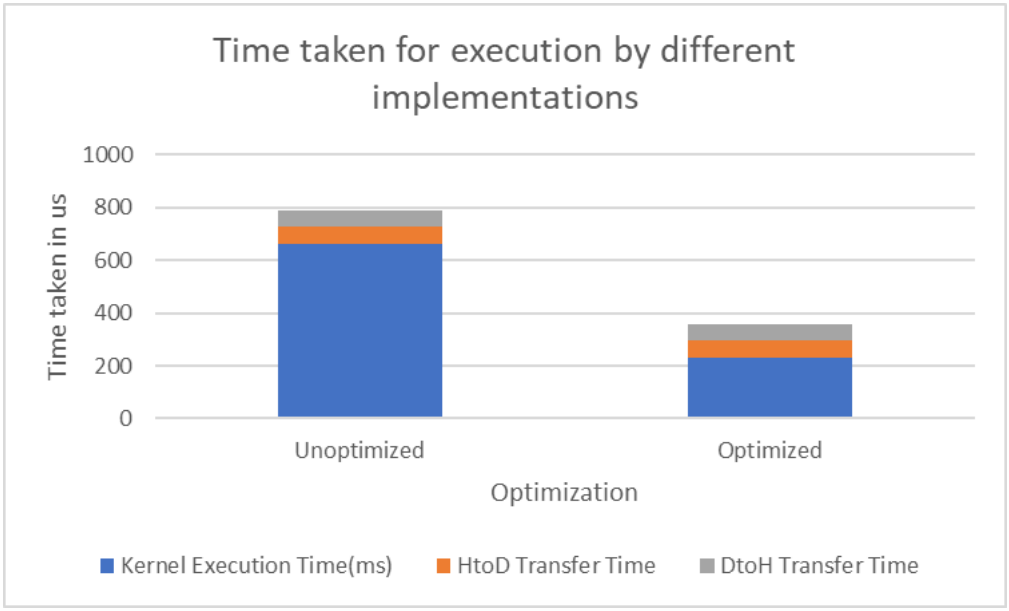
4*4 filter – Execution Time: 230.14us

7*7 filter – Execution Time: 1.9365ms

The graph depicts the time taken by the optimized version of the median filter for different filter sizes.

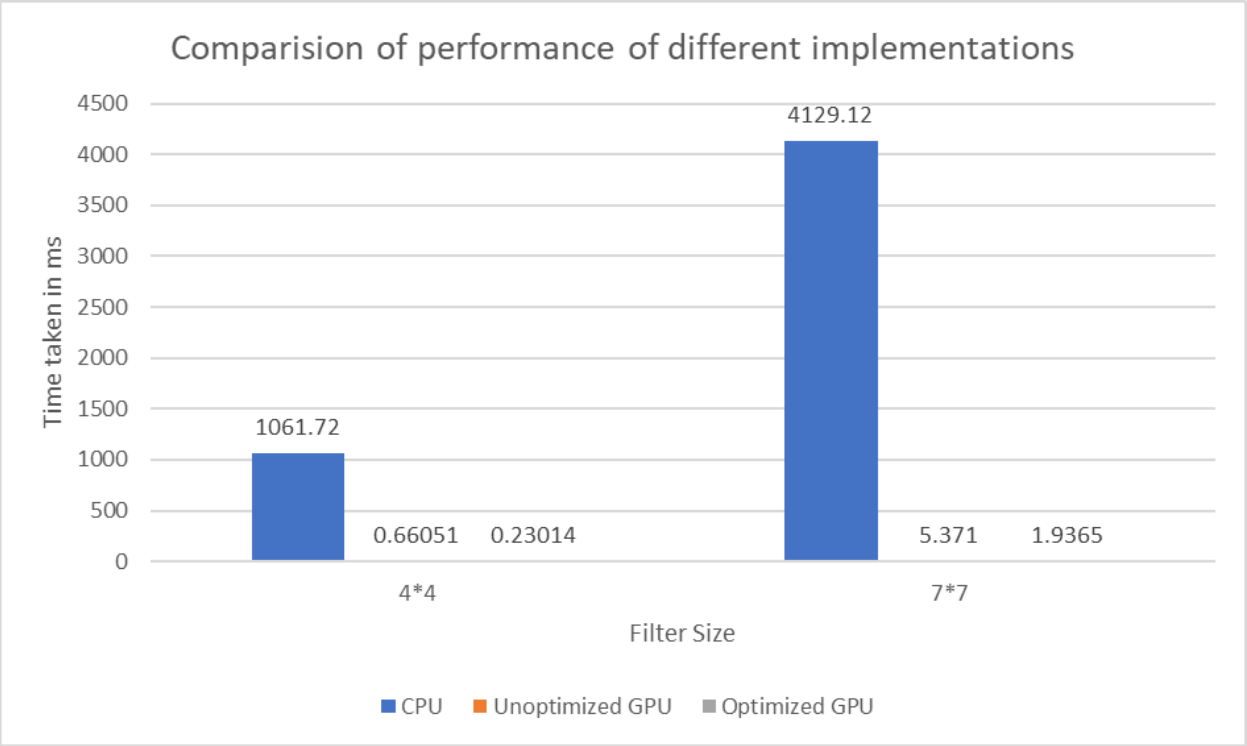


The graph depicts the time taken by the optimized and the unoptimized version of the 4*4 median filter. It can be clearly observed that the shared memory implementation performs considerably better.

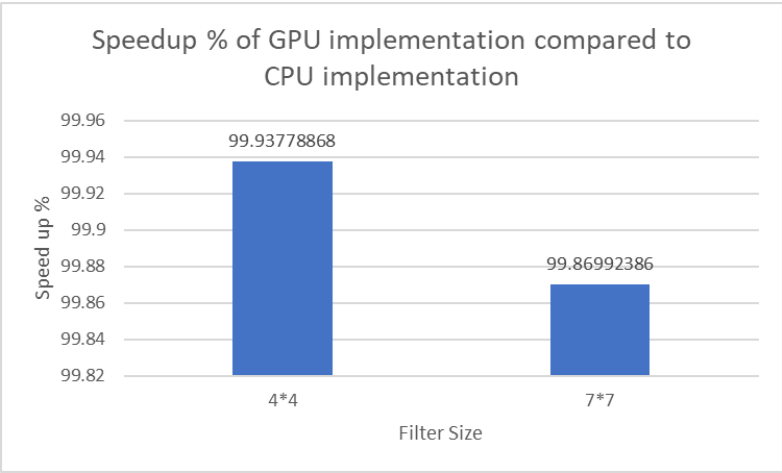


d) Conclusion

The graph below depicts the time taken for execution in the case of all three implementations. It can be clearly observed that GPU implementations perform considerably better as compared to the CPU implementation of median filter.



The optimized GPU implementation offers a speedup of 99% in case of both the filters as compared to the CPU as can be seen in the graph below.



Pooling Layer

a) CPU Implementation

After implementing the Pooling Layer in the CPU, I tested the accuracy of the algorithm with a max pooling with filter of 2*2 dimensions moving with stride of 2. The input was chosen as 4*4*1 initialized with random values. The output of the same can be seen below:

Output Dimensions: 2 * 2

Actual Matrix

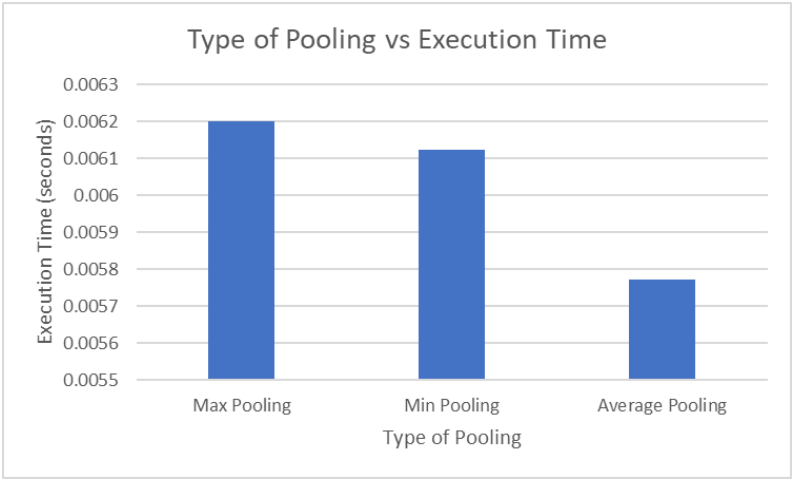
Channel: 0
89.1338 90.4747 16.7222 98.3355
44.7406 18.355 92.4943 68.5172
74.2414 41.8269 48.4691 70.9737
76.5943 66.5542 93.3152 70.6424

MaxPool : 4 x 4 with a 2 x 2 window -> 2 x 2

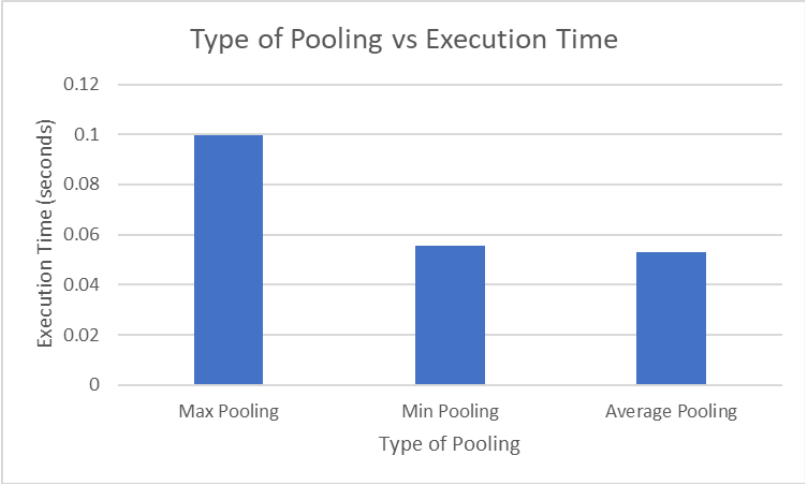
Channel: 0
90.4747 @ (0, 0)
98.3355 @ (0, 1)
76.5943 @ (1, 0)
93.3152 @ (1, 1)

It took 0.00620055 seconds.

The execution times for the MaxPooling, MinPooling and AvgPooling for the above configuration can be seen in the graph below:



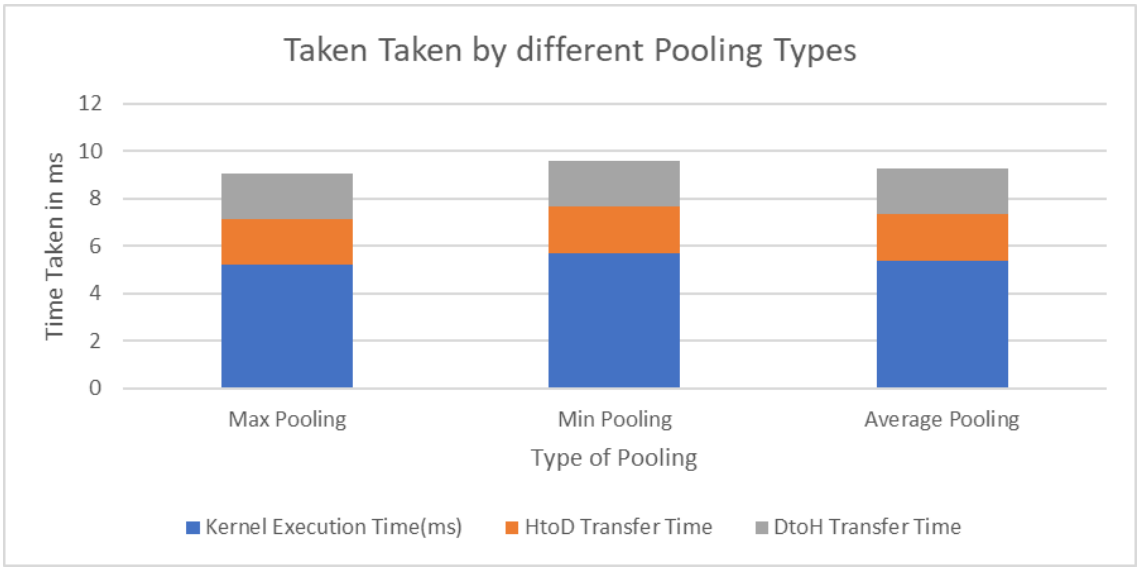
In case of an input image of 32*32*1 max pooled with a 4*4 filter with stride of 1, the execution time taken is displayed in the graph below:



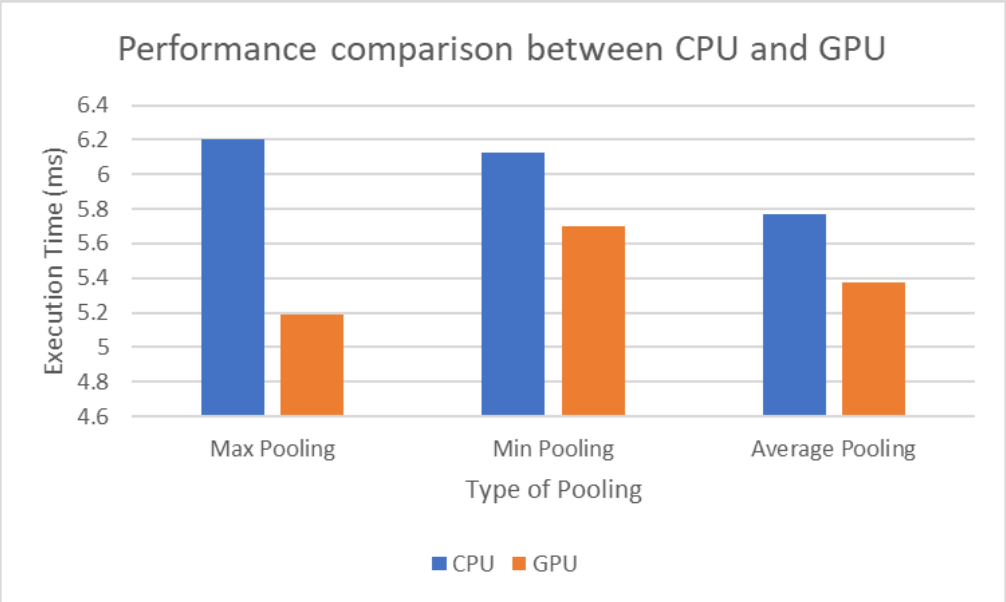
As can be seen in the graph above the Max Pooling takes the most time at 0.0996256 seconds.

b) GPU Implementation

In order to improve execution time, I implemented Pooling in GPU with a (16,16) thread block. The block has a dimension z to be able to handle more than one channel and works effectively. In case of an input image of 32*32*1 max pooled with a 4*4 filter with stride of 1, the execution time taken is displayed in the graph below:



The comparison between the performance of CPU and GPU for 32*32*1 input matrix max pooled with a 4*4 filter with stride of 1 can be seen in the graph below:



c) Optimized GPU Implementation

I attempted the shared memory optimization of the pooling layer. It can be seen in the shared_mem branch. I was getting erroneous values and couldn't figure out a way to improve upon them.

d) Conclusion

The speedups of each pooling type when implemented on GPU vs on CPU can be seen in the graph below: