A4 Report

Azan Ghazi and Zhuolun Jia

Part 1: Kernel Implementation

1. Best CPU:

The best CPU implementation that we observed in Assignment two was the Work Queue implementation with 8 threads and work chunks 4. Work queue is a parallel algorithm where the image is divided into multiple small chunks and deal with them. Counters are setup for the amount of chunks, chunks in filter and chunks in normalization to manage the data dependency. After that we build up chunks by signing every chunks' column start, row start, column end and row end, then insert the chunk into the queue.

In the queue_work function, the first thing is we apply the filter to chunks, we use a while loop to make sure every chunk has gone though the counter and use mutex lock to order the queue. We then have a barrier to make sure every thread computes the local max and min, then the global max and min will be determined. The last step is to normalize the chunks.

2. Kernel1 :

kernel1 is requiring us to have one pixel in each thread and column major. For column major behavior, first we set the index of thread called idx, which is blockId * blockDim + threadIdx as usual. Then checking the height of the image. If the height is 1, then variable "row" will always be 0, otherwise, it will be the remainder of idx divides height. Similarly, the index of column will be the result of idx divides height. So the index of current pixel will be row * width of image + column. After checking neither idx and index overflow the size of image, we call apply2d with the row and column we got just now. As for assigning one pixel in each thread, we just deal one pixel in every kernel1 function. As for normalization, it is pretty same as kernel1, using same method to find the index of row and column, then call the formula on the "index" computed by row and column.

3. Kernel2

The requirement for Kernel 2 is to have one pixel per thread, and moving in a row major order. This is done by first using cudaMalloc and cudaMemcpy to load in the filter, input array and output array into device memory. We then pass these values to the kernal2 global function and it simply uses the idx (based on the threadId.x and blockidx) to calculate the row and column of the pixel it needs to modify. We then feed these values into the apply2d function we have created which returns the filter output when applied to that pixel. Next the correct pixel is given its new output and __syncthreads is called to ensure that all threads reach the step of writing to output. Once this is done, we use the reduction function to find the global min and max, and a normalize function with the same lookup structure as kernel2 to normalize the output.

4. Kernel3

The requirement given for Kernel 3 is Multiple pixels per thread, consecutive rows, row major. This requirement was satisfied by allowing each thread to handle one row of data, and staying in the constraints of a block as if we have more rows than available threads, more than 1024. Then we can satisfy the consecutive rows condition and have each thread handle consecutive rows. This is done by first using cudaMalloc and cudaMemcpy to pass the filter, input and output values to the device memory. We then also calculate how many rows each thread should handle and pass that value to the kernal3 global function.

The global function then uses its threadid.x to determine its assigned rows and handles each pixel in those rows according to the kernel 2 setup. The normalize also echoes this behavior and follows the access pattern of kernel 3 to decide what needs to be normalized.

5. Kernel4:

Kernel4 wants us to have multiple pixel per thread and sequential access with a stride equal to the number of threads. This kernel's idea is from kernel7 in the lecture's code of Week10. Every thread can access to multiple pixels by stride access, using gridSize to jump to the thread we want to filt. And use a while loop to filt as many pixel as possible. Normalization is same as function kernel4, using while loop to access every pixels. What is more, the computation of gridSize is related to the block's size, so the current thread will deal with the pixel which is stride equal to the number of threads, just like the example on the assignment handout.

6. Kernel5

From the data we noticed from kernels 1-4 we noticed that kernel 2 seemed to have the best performance. Especially when dealing with larger images. We then based kernel 5 off kernel’s strategy and tried to use various speedup strategies to try and improve time.

The first strategy we tried was to use pinned memory to try and reduce the amount of cudaMemCpy commands we would need to use to transfer memory to the kernel device functions. We noticed that each time we declared pinned memory and tried to use it, it would be a lot faster to write data and greatly improve the transfer out time. We also noticed that the transfer in time also changed and increased by a large amount. And there was a negative bump in performance when passing pinned memory to the device functions, as they would run somewhat slower than the pure device memory. The overall speedup seemed negative and so we abandoned this tool.

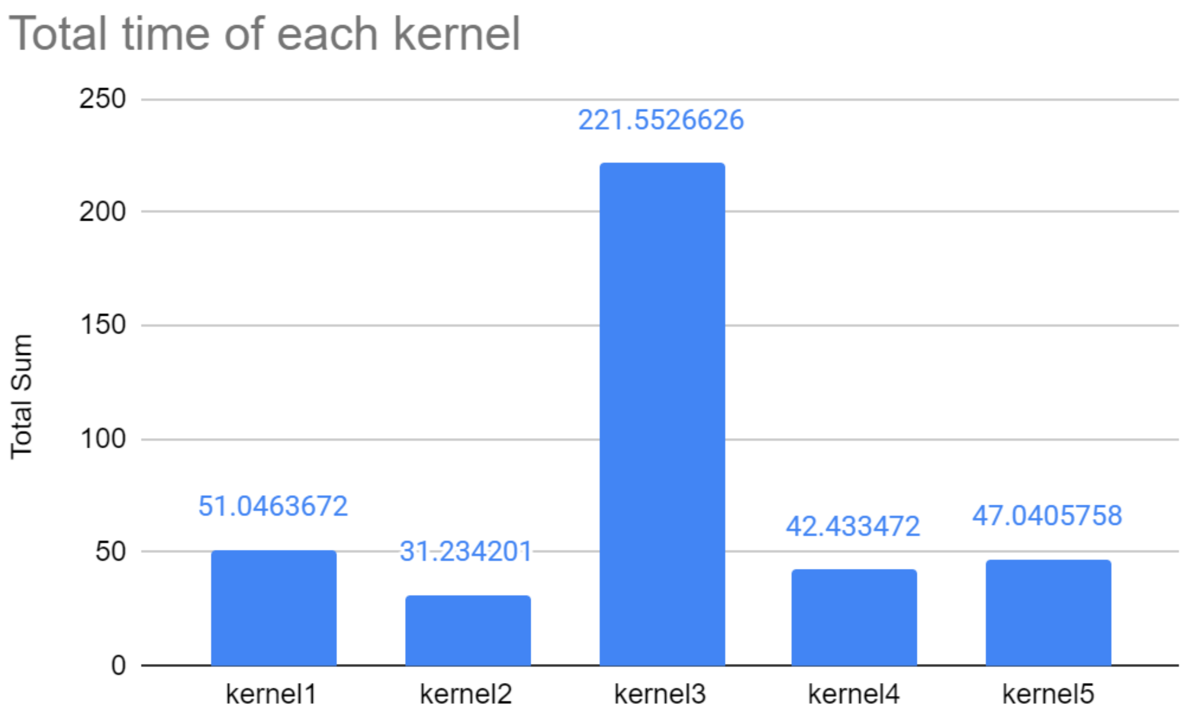
The next strategy we used was to try and use streams, we changed all the cudaMemCpy functions into cudaMemCpyAsync functions and noticed a small jump in performance in the reads, and didn't seem to have negative costs, especially since we were loading different data into different memory. Thus we chose to keep this strategy.

The next strategy we tried to use was batched memory. We tried to batch all the variables into pinned memory and deliver this load device functions. We noticed that the functions seemed to run slower than with normal device allocated memory, and there was still an overhead cost to using pinned memory, thus we also did not continue to use this moving forward. Overall we were able to achieve a 3-4% improvement over kernel 2, but were not able to reach the 10% goal.

Part 2: Reduction Implementation

The reduction mainly used lab10’s algorithm, all kernel share same reduction algorithm, slightly different on normalization. The big picture of the algorithm is let every thread takes at least one pixel in the image, and using the amount of block to decide if the algorithm should be run again, during the reduction, the amount of block will become smaller and smaller, that is saying the algorithm is reaching the global maximum and minimum. If the block amount reaches 1, that means every pixel inside the image have been all compared, the normalization function could be ran for every pixels. As for finding the maximum and minimum, we have a gpu_switch_thread function, which can switch function reduction_max and reduction_min with 10 possible block sizes. For both reduction_max and reduction_min, we use shared memory which can dynamically changed the size of array. What is more, we also use completely unrolling to saves all unnecessary work in every warps, that will gives the fastest running time.

Part 3: Analysis - Best Kernel, Worst Kernel

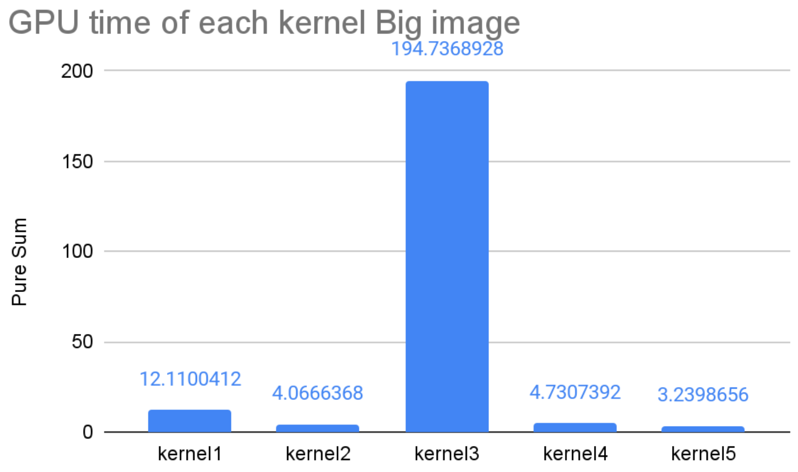


From the graph, we can see the kernel that has best performance is kernel2, and worst performance is kernel3. We are going to analyze the reason for their different performance.

For kernel2, which is 1 pixel per thread and row major, it takes advantage of maximum threads and blocks. It split up the whole image into small and compact pieces. Kernel1 also has the same feature, but the reason kernel1 was beaten by kernel2 is that memory was stored in row-major form. So kernel2 will have better memory bandwidth, which leads it to the best kernel.

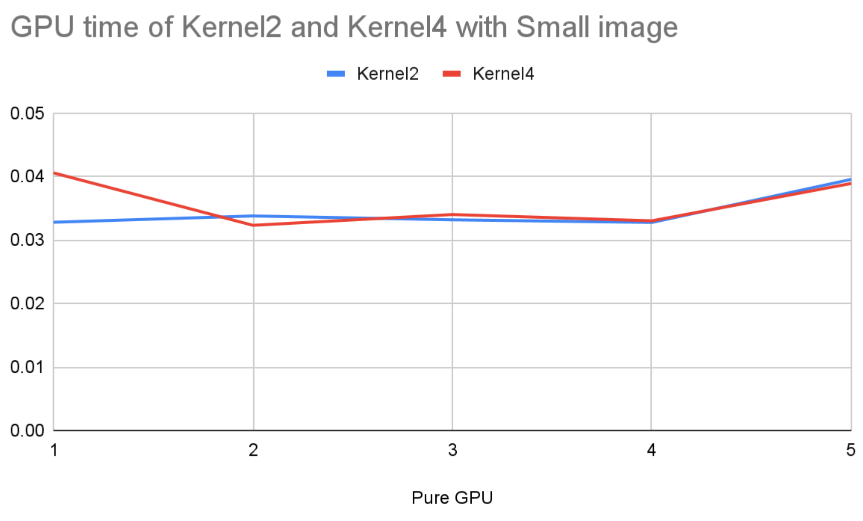
For the worst kernel, the only option is kernel 3 as it is far worse then the other kernels. Kernel 3 does not take advantage of the multiple blocks. It only focuses on one block of data. It focuses on work being distributed across threads using row-major traversal. Each thread must complete one row, only after that the thread can be assigned more work. This approach is good for taking advantage of a small number of threads available, however we are dealing with a massive GPU thread, and miss the optimization of memory coalescing. In conclusion, this approach does not work well if we have a lot of free blocks that are not being used, and so it is far worse than the other kernels.

Part 4: Analysis - Kernel 2 and Kernel4



As we discovered above, kernel 2 is benefit from the same advantage, which is memory are stored in row-major form, which is same as the way we mapping pixels into the thread. However, as the graph shows, kernel4's performance is not as good as kernel2. In this part, let's take a closer look of this situation.

We test the kernels with images that have different size, and we surprising realize that in small image, the pure GPU time of kernel4 is better than kernel2 at some time. The reason of kernel4 is better than kernel2 at some



time is that by limiting the max number of blocks to the number of multiprocessors, kernel 4 reduces the overhead of switching blocks of multiprocessors. However, with the image size become larger, the kernel4's speed is dragged because the complexity of computing the maximum and minimum exceeds the complexity of switching thread. What is more, for small image, kernel4 has fewer thread, which might lower the hidden latency in some case.