CSC367 A4 Report

Gongzhi Wang

I. Introduction

To observe the performance impact of different memory access methods on the GPU image filtering and compare their speedup to the performance of the CPU image filtering. Different workloads were tested across all methods for better accuracy.

II. Testing Environment

The tests are done in the DH lab machines.
CPU: i7-9700
GPU: RTX 2070

III. Methods

Analysis was done over different workloads on all the methods to test out the overall performances. All tests results were averaged among 10 runs to eliminate inconsistency.

The general reduction algorithm is derived from Lab 10. All methods share the same reduction algorithm to get accurate speedups among different memory access methods.

Clarification About the Tests: The PGM images used in the actual tests are all square images. This is to prevent inaccuracy for some kernels. However, discussions about these edge cases such as a very tall image or a very long image will be covered in the analysis section.

IV. Implementations

A. CPU Horizitonal Sharding

The method spawns 8 threads to distribute the workload. Each thread gets height/8 (or (height + 8 -1)/8 if height is not a multiple of 8) rows to compute and store their own local minimum and maximum. After all the threads finish, all threads will update the global minimum and maximum accordingly and do normalization in the same distribution.

This method is chosen because it has the best performance among all other methods as tested in A2.

B. **Kernel 1**
Every pixel in the image gets assigned to one thread in column-major distribution. Below is an example from the assignment page.
T0: pixel[0][0]
T1: pixel[1][0]
T2: pixel[2][0]
T3: pixel[3][0]
T4: pixel[0][1]
T5: pixel[1][1], etc.

C. **Kernel 2**
Every pixel in the image gets assigned to one thread in row-major distribution. Below is an example from the assignment page.
T0: pixel[0][0]
T1: pixel[0][1]
T2: pixel[0][2]
T3: pixel[0][3]
T4: pixel[1][0]
T5: pixel[1][1], etc.

D. **Kernel 3**
Every row in the image gets assigned to one thread in row-major distribution. If there are more threads than rows, then the remaining threads stay idle. If there are fewer threads than rows, then some threads will have more rows to compute. Below is an example from the assignment page.
T0: pixel[0][0] pixel[0][1] pixel[0][2] pixel[0][3] pixel[1][0] pixel[1][1] ... pixel[1][3].

T1: pixel[2][0] ... pixel[3][3]

E. **Kernel 4**
Every thread gets access to multiple pixels with stride access. Consecutive pixels are being accessed by different threads. Since one multiprocessor can only handle 1 block at a time, we run at most the maximum number of multiprocessors as the block number for this Kernel to avoid overhead. In this case, this number is set to 36 as there are 36 multiprocessors on an RTX 2070. The stride is the product of the number of blocks and the number of threads needed.

F. **Kernel 5**

Same implementation as Kernel 2 with two major differences. The first one is pinned memory before the image is transferred into the GPU. The second one is that the kernel uses constant memory to store the filter being tested.

G. **Reduction on Min/Max**

All the kernels share the same reduction strategy. The minimum and maximum pixel values are computed in a common reduction kernel which accesses one pixel per thread and stores them in a 2d shared array called sdata[][]. The first dimension, which only has 0 and 1, represents the minimum value array and the maximum value array respectively. The second dimension represents the corresponding thread id in its block.

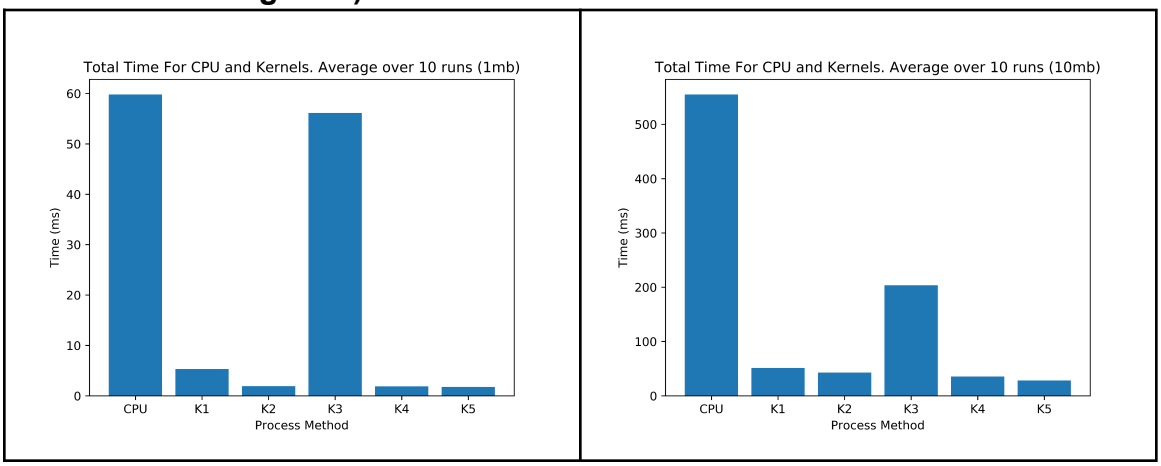
The reduction method is the same as kernel 6 in lab 10, where we unfold the for loop to get better performance. Reductions are continually being performed until the number of minimum and maximum pixel values is fewer than the max number of threads in a block. Then a final reduction will result in the global minimum and maximum value.

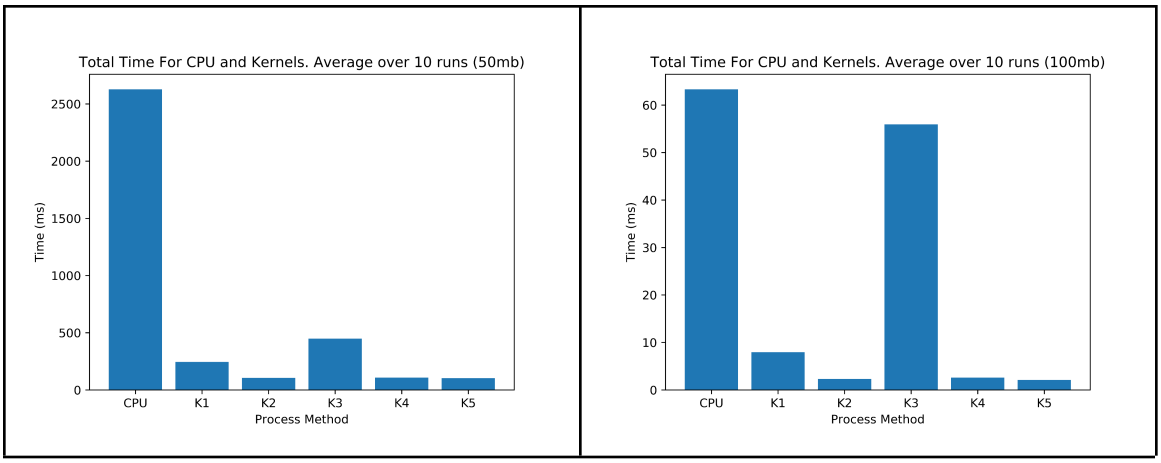
H. **Normalization**

After finding the global minimum and maximum value, every kernel will perform normalization using the same workload distribution as they have before.

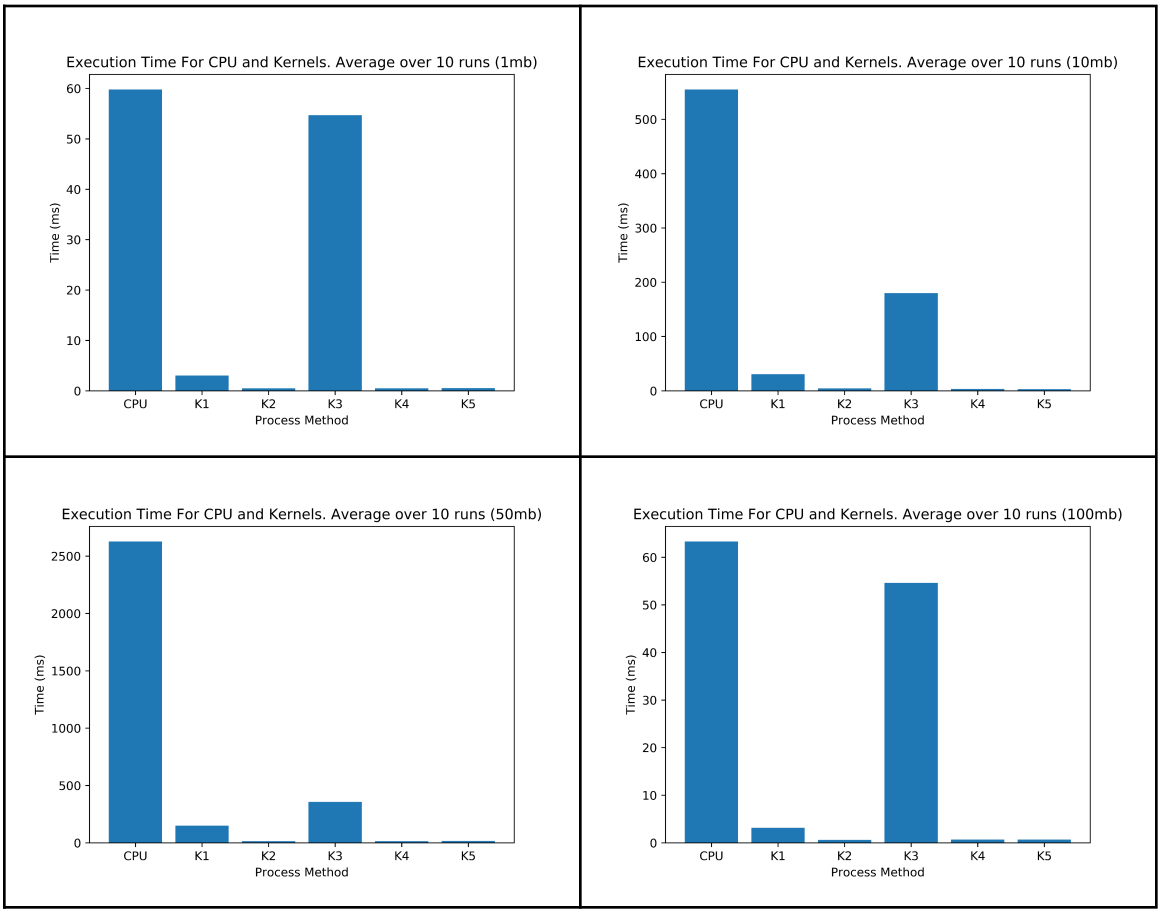
V. Results

A. **Total Time for CPU and Kernels to Execute an Image (including transferring time).**





B. **Pure Execution Time for CPU and Kernels to Process an Image**



VI. Analysis

A. Kernel 1

Although kernel 1 does not apply any optimization mentioned in lectures, it still shows relatively good performance compared to the CPU. Since it occupies a lot of the GPU resources for one image filtering, speedup is expected.

B. Kernel 2

Kernel 2 has better performance than kernel 1 with the usage of the same amount of resources. The fact that consecutive threads process consecutive pixels allow kernel 2 to take the benefits of memory coalescing during execution to have a better memory bandwidth. This optimization is reflected in the greatly improved performance.

C. Kernel 3

Kernel 3 has the worst performance among all GPU kernels. Its memory distribution strategy not only gives the advantage of massive GPU thread resources but also misses the optimization of memory coalescing. The combination of these two critical weaknesses makes kernel 3 the slowest kernel in this test.

D. Kernel 4

Kernel 4 has an even better performance than kernel 2 in these tests while using fewer GPU resources. Similar to kernel 2, kernel 4 takes the advantage of memory coalescing. On top of that, by limiting the max number of blocks to the number of multiprocessors, kernel 4 reduces the overhead of switching blocks of multiprocessors.

Interestingly, when the scale of the image increases, the performance of kernel 4 decreases and is slower than kernel 2. This might be caused by the limited resources that kernel 4 is using as the computation complexity exceeds the overhead of switching blocks. Moreover, fewer threads might lower the potential of hiding latency in these cases.

E. Kernel 5

Although the memory access method inside a kernel is a major

consideration and different methods have their own advantages and disadvantages, the transfer bandwidth becomes a bottleneck of performance.

By comparing the differences in table 1 and table 2, the major takeaway is that transferring images to the device and back to the host takes a noticeable time to finish for all kernels except kernel 5. Table 2 demonstrates the similar performance in terms of pure execution between kernel2, kernel 4, and kernel 5, which proves the only difference between kernel2, kernel 4, and kernel 5 in table 1 is the bandwidth. This makes kernel 5 at least 10% faster than kernel 2 and kernel 4.

With that being said, kernel 5 takes the same memory access method as kernel 2 but with two major differences before the execution as mentioned in the implementation section. By using pinning memory in the RAM, kernel 5 does not need to suffer the loss of data due to page replacement and other programs, which makes it take full advantage of RAM during transferring data.

Moreover, the usage of constant memory to store the filter increases the access speed of threads in the same half-warp as long as they are reading the same address. Since the same filter is used by all threads and accessed in the same pattern, the kernel is able to use the full capacity of the cache.