Assignment 4

CSC367

Ibrahim Abdi

Justin Pham

Introduction:

To observe performance impacts from different memory accesses strategies on GPU image filtering, several workload distribution methods were tested against one another and compared for execution time.

Methods:

Analysis was performed over the different methods and differing image sizes to measure the potential overheads of the different methods in addition to overall speedup.

The general reduction algorithm for finding the min/max post application of the filter was kept constant across all methods to get consistent speedup comparisons across the access methods.

Implementation:

CPU comparisons were performed alongside for each GPU run to measure speedup.
A sharded row implementation was used for the CPU timings as it performed the best compared to other CPU parallel implementations from a previous analysis.

Prior to running any Kernels, kernel 1 was run over the data without measurements to reduce any affect cold accesses would have on the performance of Kernel 1.

For Kernel 1, every pixel was processed by its own thread and were assigned column major.

Kernel 2 like Kernel 1, processed every pixel with its own individual thread, but assigned them row major.

Kernel 3 assigned consecutive rows to each thread, processing each load row major. Should there be more threads than rows, the threads would still be assigned full rows for processing leaving some threads without work.

Kernel 4 processed consecutive pixels on different threads which then strode across the pixel array until the entire image was filtered. Since a multiprocessor can only work one one block at a time, we always run Kernel 4 with the max number of multiprocessors as the block count and the max number of threads per block. The stride is simply the product of those two.

Kernel 5 used the same implementation as Kernel 2 with two key differences. The first difference is that it uses constant memory to store the filter. The second difference, is that prior to the image being transferred into the GPU, the image matrix is first pinned to memory.

Normalization for each Kernel method made array accesses using the same patterns as their respective kernels.

Min/Max acquisition on the other hand, was constant throughout all kernels. The min and max were both found via a common reduction kernel which accessed a pixel per thread storing it into a two dimensional shared array. The rows of the array reduced com at a time and sorting them for min/max for each in block reduction. Reductions were then continually performed until the number of min/max values were below the max number of threads in a block which were then reduced a final time resulting in the global min/max.

Results and Analysis:

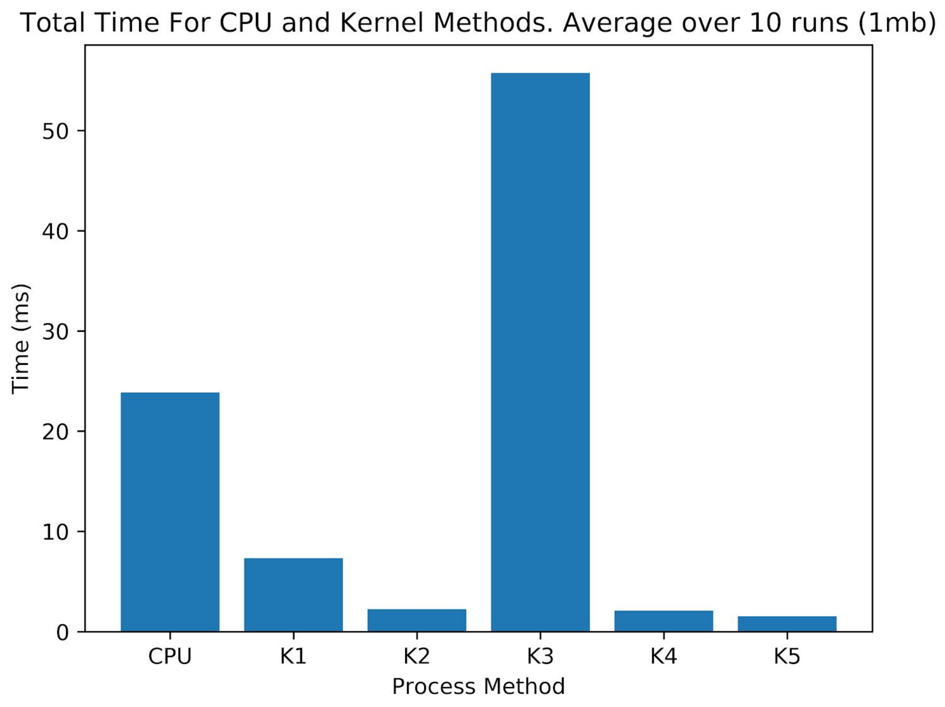

Figure 1: Execution Time (including transfer times) of the CPU and Kernel for 1mb image

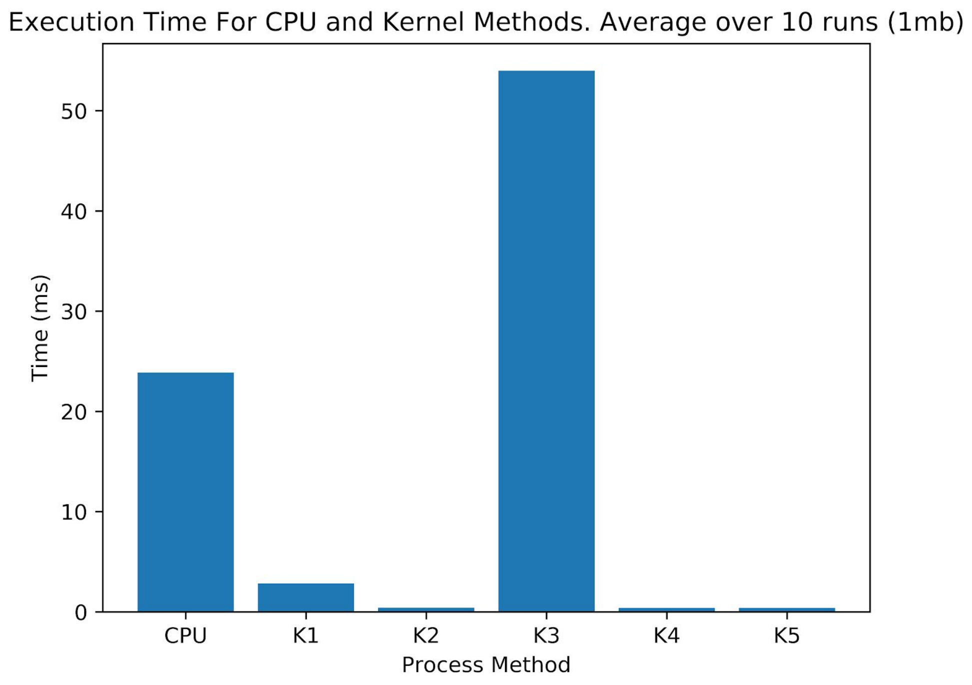

Figure 2: Computation Time of the CPU and Kernel for 1mb image

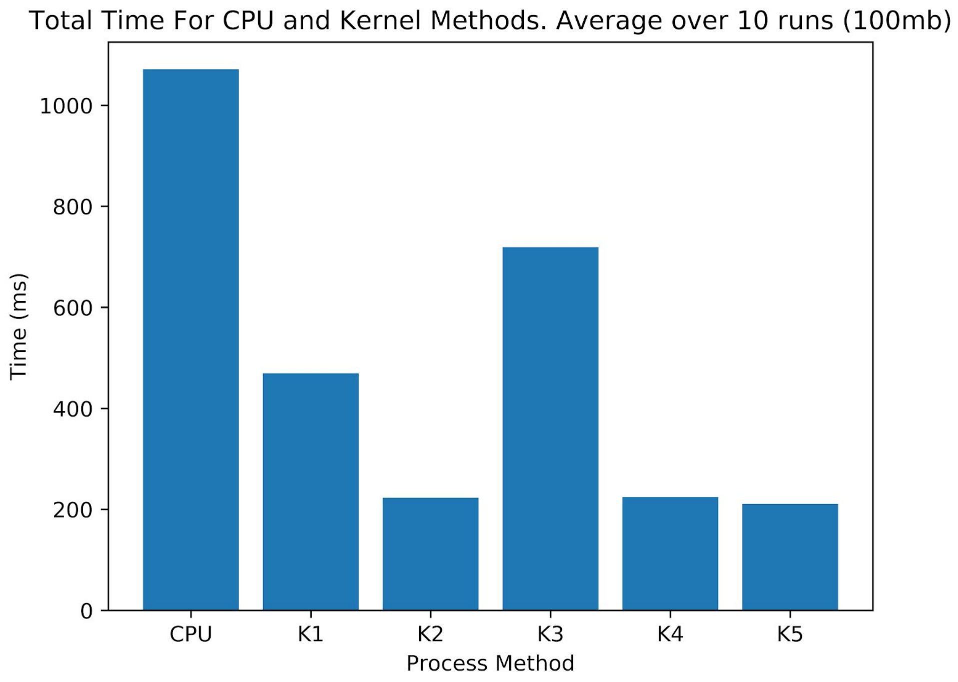

Figure 3: Execution Time (including transfer times) of the CPU and Kernel for 100mb image

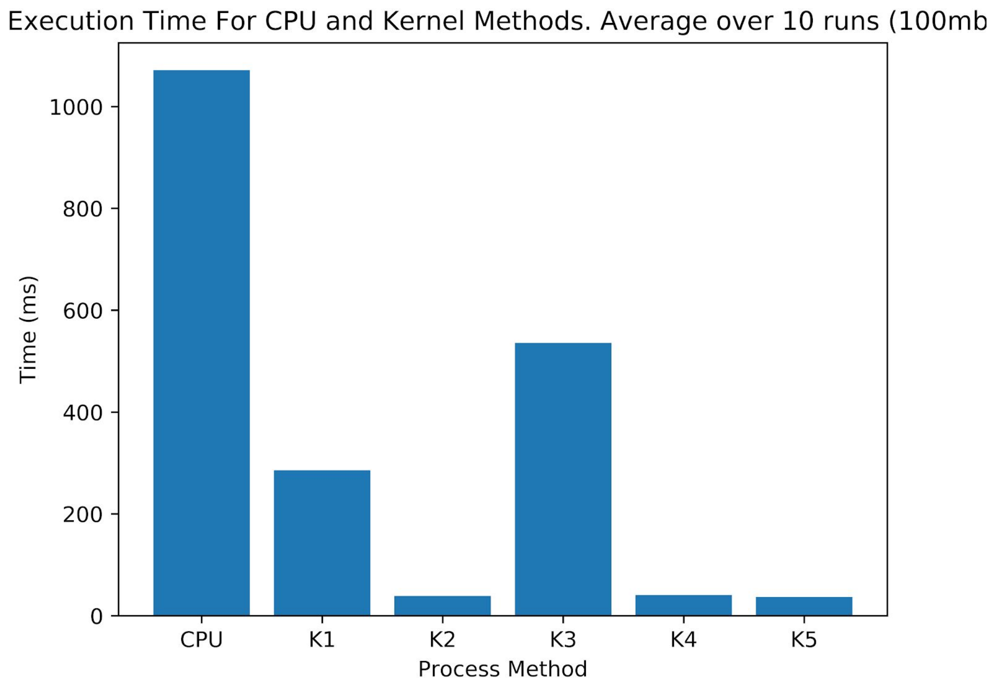

Figure 4: Computation Time of the CPU and Kernel for 100mb image

Kernel 1 showed relatively good speedup compared to the CPU implementation. Given the distribution of work over the large amount of threads facilitated by the GPU, some level of speedup was expected.

Kernel 2 also showed speedup, but to levels far beyond that of Kernel 1. Although Kernel 2 took a similar approach to task distribution over the threads, the fact that consecutive threads processed consecutive pixels of the image allowed for the method to take advantage of memory coalescing whose speedup is reflected in the greatly improved performance.

Kernel 3 was by far the slowest method even when compared to the CPU implementation. Its distribution of work not only did not take advantage of memory coalescing like Kernel 2, but also did not take advantage of the processing resources of the GPU opting to assign rows to threads reducing the methods concurrency.

Kernel 4 showed significant speedup even when compared to Kernel 2. Striding done by this method allowed it to take advantage of memory convalescing like Kernel 2, but limiting the number of blocks taking into consideration the number of SMs on the device allowed the method to reduce the overheads caused when distributing the workload reducing the overall compute time.

It was found that for exceptionally large images, that Kernel 2 exceeded Kernel 4 in performance, but for most other images Kernel 4 achieved better performance. This may have been due to the fact that Kernel 2 assigns more blocks than Kernel 4. With an extremely large image, the number of registers available to Kernel 4 may become a limitation as the number of threads is constrained by the total number of blocks lowering the potential for hiding memory latency.

Although compute time was a major consideration and showed a lot of variance across the methods, with each improvement the bottleneck in performance caused by the transfer of data to the device became more apparent for most tests barring the extremely large image case.

One of the takeaways from the overall data is that the methods are comparatively similar with or without the inclusion of transfer time relative to one another. The transfer methods were consistent across all kernels barring Kernel 5, which explains a major difference when comparing execution times with transfers when observing Kernel 5's execution to the other methods. Transfer time became the main bottleneck in terms of speedup as decreasing compute time showed diminishing returns for overall speedup.

So for Kernel 5 the bottleneck was addressed and its effect reduced the execution times for both the computation and the transfer. In terms of the computation, the methods use of constant memory allowed data elements ubiquitous across all threads to make use of the constant cache. Due to the limited resources of constant memory, the data element stored there was the filter. Constant memory is known to be fast as long as threads in the same half-warp are reading the same address. Since, the filter is used by all methods and the operations on each warp is performed in lock step, it enables the method to make full use of the constant cache's capabilities. Although use of constant memory was effective in improving compute time, its effect was relatively small as compared to the transfer time. Thus, the major speedup consideration for Kernel 5 was its use of pinned memory. By pinning the image array to the RAM, the method avoided any calls and transfers from pageable memory relying only on the calls to the memory pinned. This produced significant speedup in the transfer in time for the kernel making it the fastest kernel overall.