CSC367 report 4

hongmao.wu

December 2023

1 cpu implementation details

This implementation is taken from assignment 2. The program divides the entire image into rows and assigns each thread consecutive rows based on their thread ID. This approach is fastest because it leverages cache locality. Note that, regardless of the image size, this implementation only uses four threads. This choice simplifies the assessment of GPU speedup, making it easier to estimate performance improvements as the number of participating threads in the parallelization task increases.

1.1 analysis

The primary drawback of this method is the fact that it executes on the CPU, which is inherently less efficient compared to utilizing the GPU. The GPU owns a significantly higher number of cores in each of its streaming processors. Each streaming processor typically houses around 32 cores, enabling the concurrent execution of 32 threads in a warp. These 32 threads execute same instruction. Multiple streaming processors collectively form the GPU unit. Consequently, the GPU is better suited for handling inherently parallelizable tasks, and it is anticipated that a higher speedup appears as the scale of the image increases.

2 GPU implementation

The processing of image pixels on the GPU involves a sequence of steps. Initially, the program allocates memory on the device, this process requires calculating the total memory required. After initializing the device memory space, the program transfers the original image pixel values to the device. Subsequently, a kernel is launched, given the number of blocks and threads per block, apply the same filter on each pixel value and storing them on global memory. These number of block and number of thread per block vary depending on the specific kernel being executed. Following the launch, the program halts CPU execution until the GPU completes all computations. Finally, the processed image values are transfered from the device memory back to the host memory.

For each kernel, the program records execution times for three key events: transfer-in, transfer-out, and computation. Notably, both computation and transfer-out events are further divided into two parts in all kernels except kernel 5. The reasoning behind this division is that in kernels 1, 2, 3, and 4, the calculation of global maximum and minimum takes place on the CPU using a for loop. Also it's important to mention that, for simplicity, every kernel only generates 1-dimensional grid and block. Also note that some threads may be created but not actively participating in computation. This is due to the fact that the total number of pixels to be processed might not evenly align with the number of threads generated.

2.1 kernel 1 and kernel 2 implementation

Kernel 1 and Kernel 2 both operate on a one-pixel-per-thread basis, processing pixels in column-major and row-major order, respectively. Hence, the maximum size of the image that can be processed by these two kernels is constrained by the maximum number of blocks multiplied by the number of threads per block.

Kernel 1 and Kernel 2 are common in many ways, and one way is that each thread stores its value in an array based on its thread ID within a block. Each block owns its array and the array is allocated on shared memory so every thread have access to it. At the end of the kernel, all threads in a block (i.e., 1024 threads) engage in a reduction process to determine the local maximum and local minimum within that array. The reduction method used is based on the thread ID within a block. In each iteration, threads with IDs that are multiples of 2 perform comparisons, followed by threads with IDs that are multiples of 4, and so forth. The local maximum and local minimum are then stored in other two arrays, each slot on these two arrays is reserved for one block.

2.2 kernel 3 and kernel 4 implementation

Kernel 3 divides the total number of pixels by a specific number of blocks and threads per block. In this program, it selects 32,768 blocks and 1024 threads per block. This choice will be discussed in the analysis section. Similarly, Kernel 4 also opts for 32,768 blocks and 1024 threads. However, these two kernels differ in their access patterns.

In Kernel 3, each thread accesses a set of consecutive pixels, while in Kernel 4, each thread traverses the entire image with a stride size equal to the total number of threads. Notably, compared to Kernel 1 and Kernel 2, there is a significant difference in computation time. Since each thread in Kernel 3 and Kernel 4 does not handle just one pixel value, it is required to find the maximum and minimum values within the processed pixel values assigned as it processes through the pixel values. This distinction in access patterns introduces a unique aspect to the computation within these kernels.

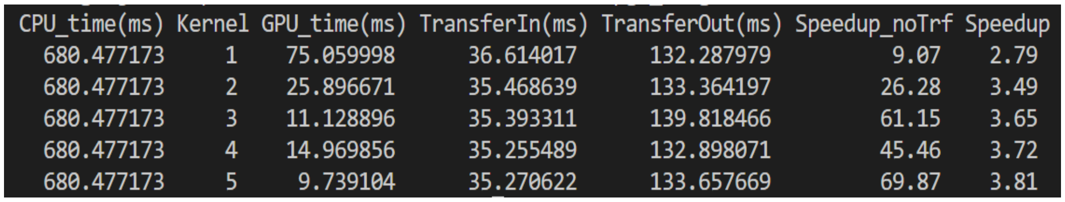
2.3 kernel 5 implementation

Kernel 5 incorporates several optimizations based on the implementation of Kernel 3, which is the fastest out of the four. These optimizations are crucial for achieving a 10 percent speed-up. Two major improvements have been implemented, and their impact will be discussed in the subsequent analysis.

The first optimization involves invoking another kernel, named global reduction. This kernel loops through the output array to find global maximum and minimum values. This modification prevent the transfer-out event and computation event from being divided into two parts.

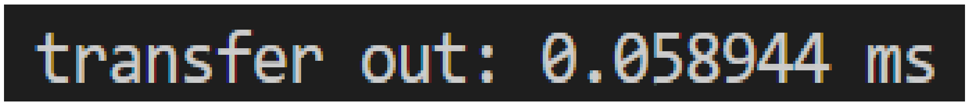
The second optimization targets the elimination of branch divergence. This enhancement is applied to both the reduction process, responsible for finding maximum and minimum values within a block, and the global maximum and global minimum calculations. The elimination of branch divergence contributes to more efficient parallel execution and is instrumental in the overall performance improvement observed in Kernel 5.

2.4 Analysis



```
CPU_time(ms) Kernel GPU_time(ms) TransferIn(ms) TransferOut(ms) Speedup_noTrf Speedup
  680.477173      1    75.059998      36.614017      132.287979          9.07    2.79
  680.477173      2    25.896671      35.468639      133.364197         26.28    3.49
  680.477173      3    11.128896      35.393311      139.818466         61.15    3.65
  680.477173      4    14.969856      35.255489      132.898071         45.46    3.72
  680.477173      5     9.739104      35.270622      133.657669         69.87    3.81
```

Figure 1: All kernel speed up



```
transfer out: 0.058944 ms
```

Figure 2: Negligible optimization in kernel 5

In Figure 1, it is evident that Kernel 1 exhibits the slowest GPU execution time and consequently the lowest speedup. Several factors contribute to this outcome. Firstly, memory coalescing could be an issue here. As GPUs store a row of pixel values in contiguous addresses, processing pixels in a column-major order prevent the exploitation of memory coalescing. In Kernel 2, threads within same warp access consecutive memory address. The GPU can spend as much time accessing 32 pixels from global memory as accessing one pixel, contributing to improved efficiency.

A second issue in Kernel 1 is related to bank conflicts. After processing a single pixel value, each thread in a block stores its computed value in an array located on shared memory. This can lead to bank conflicts, where multiple threads access memory on the same bank, causing serialization of memory access operations. The kernel creates numerous blocks to enforce one thread per pixel, amplifying this issue. Addressing this issue involves changing the memory access pattern.

Kernel 3 and Kernel 4 demonstrate significantly lower execution times compared to Kernel 1 and Kernel 2. This improvement is attributed to addressing issues introduced by the latter kernels. Kernel 1 and Kernel 2 create numerous blocks, introducing switching overhead as the streaming multiprocessor switches between thread blocks. However, each thread must process more pixels in kernel 3 and 4. Hence, it is important to find a balance that maximize speed up. After testing out with different number of block, a balance point with 32,768 blocks and 1024 threads per block optimize the execution time. Either increase the number of block or decrease it leads to increased computation time.

In theory, kernel 3 is supposed to be slower than kernel 4 because kernel 4 exploit memory coalescing such that multiple threads access consecutive memory addresses. For some reasons, figure 1 shows otherwise. There could be other factors such as the reduction method that mediate the effect of lack of memory coalescing in kernel 3.

Kernel 5 achieves the 10% speed up resulting from two optimizations. The first optimization addresses the issue of divergent branches, ensuring that all threads in a warp participate in computation and prevent serialization caused by divergent paths. Given the huge number of warps, the amount of time saved by solving this issue is important to reach the speed up. The second optimization involves creating another kernel to compute global maximum and minimum, eliminating the need to copy a specific size of values from GPU memory to CPU memory. The amount of time saved from not transferring out is shown in Figure 2, which the effect is negligible. Second optimization benefits from low cost to invoke another kernel. However, this kernel performs reduction on an fixed size array, and it does impact the speed up as much as first optimization does. In other word, the part being optimized in second optimization is not the performance bottle.