CSC367 A4 Report

Farzaneh Tabandeh Sitao Wang

April 2021

1 Applying Laplacian filters on GPU using CUDA

Applying a Laplacian filter on an image involves using a filter of a certain size (for example 3 * 3) over all of the pixels of the image and multiplying the corresponding element of the filter by the one of the image. This task has a great amount of parallelization and in this assignment we experimented parallelization of this task using different implementations of CUDA kernels on different images. Overall, the task of applying a Laplacian filter on an image consists of 3 sequential parts:

1. Applying the filter on each pixel of the image
2. Finding the max and min value of all pixels after applying the filter
3. Normalizing the value of each pixel to be a value between 0-255 using the max and min found

Although these 3 main parts need to be done sequentially, each part by itself is a great parallel task. In particular, we parallelized these 3 parts by implementing specific kernels for them to be run on the GPU. For steps 1 and 3 which are applying the filter and normalization we have 5 different kernel implementations. For step 2 which is finding max/min of pixels after the applying the filter, we have used a warp reduction implementation using CUDA's atomic max operation(whose implementation is in the file kernels.cu). We also implemented a block reduction which also used CUDA's atomic max operation, but it is commented out in our code as its performance was nearly the same as the warp reduction one.

1.1 Kernel 1

Kernel 1's filter application and normalization implementations distributes the work between threads such that each pixel is processed by one single thread in a column-major manner.

1.2 Kernel 2

Kernel 2's filter application and normalization implementations distributes the work between threads such that each pixel is processed by one single thread in a row-major manner. This is similar to having a stride equal to the number of threads, but each thread is assigned only one pixel.

1.3 Kernel 3

Kernel 3's filter application and normalization implementations distributes the work between threads such that multiple rows of pixels are processed by one thread in a row-major manner.

1.4 Kernel 4

Kernel 4's filter application and normalization implementations distributes the work between threads such that "multiple" pixels are processed by each thread with a stride equal to the number of threads.

1.5 Kernel 5

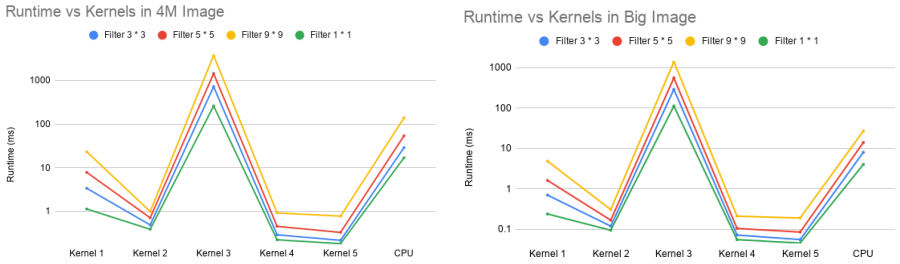
Kernel 5's filter application and normalization implementations distributes the work between threads such that "multiple" pixels are processed by each thread with a stride equal to the number of threads. This is partly similar to the kernel 4, however, since we use vector types(int2, and int4) of CUDA, we use N / 4 number of threads overall to process the whole image of size N.

2 Experiments and Results

Please note in all of the discussions in this report, we refer to N to be $N = width \times height$ of the image. Also the timing of the kernels are averaged over 5 runs of the kernel. We use the filter 3*3 by default in our code, so you could change the filter type by changing the variable FILTER at the top of main.cu.

2.1 Runtime

Figure 1 shows the results of processing the 4M and big image using different kernel implementations and our row major pthreads implementation on CPU(called *apply_sharded_rows* and *normalize_sharded_rows* in filters.cu file). As we can see, the kernel 3 has the worst time complexity compared to all implementations in both images. On the other hand, all other kernels 1, 2, 4, 5 have better time complexities than the one of the CPU parallelization implementation. We can also see that kernel 5 is the fastest among all, in particular it has around 15 percent improvement in runtime compared to the second best kernel which is kernel 4. These results are consistent on all filter types.



(a) 4M Image (b) Big Image

Figure 1: Runtime vs Kernels & CPU

In order to find out why the GPU kernels' implementations have different time improvements, we can look further at the threads and blocks configurations used in them.

In Kernel 1, we have requested to have the same number of threads as we have pixels in our image. Then we distribute the pixels among the threads in column major. When pixels are assigned to each thread column by column, we will lose the ability to fetch the data from the global memory in a coalesced way. That is because memory access of each thread of a warp is not consecutive in away that can be coalesced. That causes the kernel 2 to have a better performance compared to kernel 1 quite greatly, since in accessing by row major coalescing is possible.

In kernel 3, multiple rows of the image are distributed to each thread. In our implementation, we used 4 rows per thread(but the lack of performance is persistent regardless of the number of shared rows). We have sharded the image among the height/4 number of threads, and based on this number of threads we have calculated the required number of blocks. We have filled the blocks with at most 1024 threads depending on the size of threads we would need. So certainly we have used less number of blocks in this kernel compared to other previous two kernels. However, since each thread is assigned to process multiple rows, there is a high chance of warp divergence. Since each thread of a warp are bound to each other, this reduces parallelization and hence having a performance loss quite extensively. Also assigning multiple rows to one thread might increase its usage of registers and that reduces the simultaneous blocks that can be scheduled and that decreases parallelization as well.

In kernel 4, we assign multiple pixels to each thread with a stride of the whole grid's threads. We configured the threads and blocks sizes such that each thread is assigned 2 pixels, so we used N/2 number of threads and then calculated the total number of blocks needed. As we can see, the performance of this kernel is much better than any of the previous 4 kernels. The reason is that using a grid size stride increases coalesced memory accesses and reduces warp divergence. One might argue that we have a similar scenario in case of the kernel 2 where we still have a stride of the whole grid but we have used

more number of blocks such that each thread will be assigned one pixel of work. In kernel 2 we assign each pixel to one thread, and that might seem to have more parallelization, but why kernel 1's performance is worse than kernel 4's ? The reason is the limit of number of CUDA cores available to us at the same time. As we saw in lab 6, the underlying GPU specifications that we use here has 64 CUDA cores per multiprocessor(MP) and we have 68 MPs. That means, overall we have $68 \times 64 = 4,352$ CUDA cores in our GPU. Each CUDA core is in fact one single warp of threads. So that means we can potentially have $4,352$ number of warps to simultaneously do SIMD operations on the GPU. Now in both kernel 2 and 4, we used blocks of size 1024 threads, that means:

$$warps\_per\_block = \frac{1024}{32} = 32$$

$$\frac{4,352}{warps\_per\_block} = 136$$

which means we will have the possibility to schedule 136 simultaneous blocks at once(given all other limitations are satisfied) and any other extra blocks will be queued.

Suppose we are processing 4M image and so let $N = 4 \times 1024 \times 1024$. Then we calculated the number of blocks needed. Specifically in kernel 2, we calculated the number of blocks of size 1024 needed such that we could assign each thread one single pixel. Whereas for kernel 4, we needed N / 2 threads and so calculated the number of blocks that are needed to assign two pixels to each thread. In both kernels we used blocks of size 1024. That means:
in kernel 2:

$$required\_threads = N = 4 \times 1024 \times 1024 = 4,194,304$$

$$required\_blocks = \frac{required\_threads}{1024} = \frac{4 \times 1024 \times 1024}{1024} = 4 \times 1024 = 4096 \text{ blocks}$$

While in kernel 4:

$$required\_threads = \frac{N}{2} = \frac{4 \times 1024 \times 1024}{2} = 2,097,152$$

$$required\_blocks = \frac{required\_threads}{1024} = 2048 \text{ blocks}$$

So we can see we have more number of blocks in kernel 2. Since the max number of active blocks of size 1024 is potentially 136, that means kernel 2 will have to wait much more than kernel 4 so that all of its blocks can be scheduled and processed. While in kernel 4, we have smaller number of blocks and do double work on each block, which means we have much more parallelization compared to kernel 2.

In kernel 5, we have used the idea of the kernel 4 for thread configuration but we used N/4 number of threads to process the whole array of size N. The difference in implementation of kernel 5 is that we used vector types int2 and int4 in the filter application and normalization functions, respectively. Specifically, we used int2 in kernel 5's applying filter function so that each thread can process

4M Image

	Filter 3 * 3	Filter 5 * 5	Filter 9 * 9	Filter 1 * 1
Kernel 4	0.290925	0.454131	0.922298	0.223642
Kernel 5	0.217139	0.330362	0.780326	0.182195

Big Image

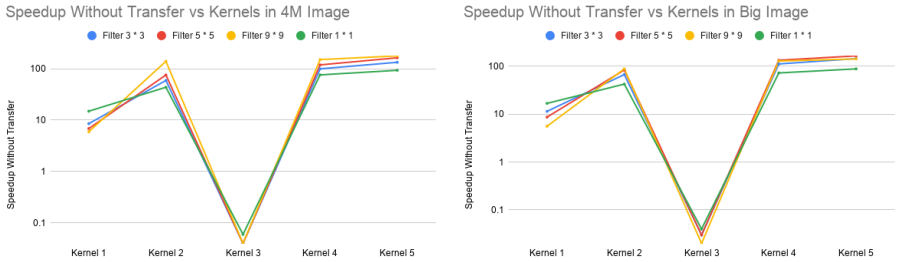
	Filter 3 * 3	Filter 5 * 5	Filter 9 * 9	Filter 1 * 1
Kernel 4	0.071315	0.104166	0.208883	0.054867
Kernel 5	0.055085	0.084525	0.188928	0.045171

Figure 2: Runtime(ms) of Kernel 4 and 5

2 pixels in a vectorized manner. Also we used int4 in normalization function, so that each thread can normalize 4 pixels simultaneously. Using vectorized operations increases the coalesced fetching of data from global memory and using N/4 number of threads means more simultaneous blocks that can do more number of works per unit of time, hence increasing performance. We can also observe that the kernel 5's runtime is mostly improved even more than 15% compared to our kernel 4 which is our second best kernel. Specifically, on 4M image and using filter $3 * 3$, kernel 5's runtime was around 0.21 compared to 0.29 in kernel 4, which is around 28% improvement. More detailed runtime in Figure 2.

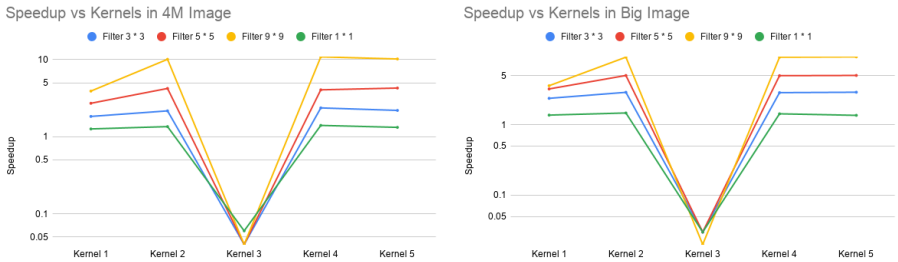
One other overall observation is that using GPU to process the images are independent of the value of width and height of the image. That is because we use the image as a 1-d array and using our best kernel, we can process image pixels regardless of the size of the width vs height. We can see that in our results the average time per pixel for both type of images is proportional to the size $N = width \times height$ of the image, and not width or height independently. However, in the case of processing the image using CPU, since our best implementation is row major sharding, the performance gain is much higher if we have an image with smaller height value. This is because less number of rows are assigned to each thread and hence more parallelization.

Also bigger filters increase the runtime as more number of elements of the array much be fetched which might not be able to be coalesced among the threads of the warps.



(a) 4M Image (b) Big Image

Figure 3: Speedup Without Transfer Time



(a) 4M Image (b) Big Image

Figure 4: Speedup

2.2 Speedup Without Transfer Time

As seen in Figure 3, without considering the time of transferring the data to GPU, the speedup of GPU kernels is very great. Specifically, using kernel 5 on 4M image, we get an speedup of around 133 on filter 3*3 and 178 on filter 9*9. Also as explained, the kernel 3 has the worst speedup of less than 0.1.

2.3 Speedup

As shown in 4, while we can achieve very high speedup values if we only consider the calculation time on GPU, taking the transfer time into consideration drops the speedup value intensively. Specifically, using kernel 5 on 4M image, we get an speedup of around 2.19 on filter 3*3 and 10.18 on filter 9*9, which has decreased so much compared to the speedups we got without transfer time.

Overall, in our kernel implementation we tried to have max occupancy by using max number of possible warps in a block and in some of the kernels distributing small amount of work to each thread so that the registers needed per thread be manageable. However, not always having a 100% occupancy is the best and we experienced that when by changing the number of needed threads

we got different run-times even for the same kernel implementation. Also we can conclude using GPU is worthwhile for the type of works that has enough computational parallelism that can compensate the heavy cost of transferring the data.