CSC367 Winter 2021
Assignment 4 Report

Shuting Zhang zhan4145
Muyuan Cao caomuyua

1. Implementation

a. CPU. horizontal sharding:

Each thread processes a set of consecutive rows, such that each thread gets a roughly equal number of rows. If the total number of rows is not divisible by the number of threads, then each thread will be assigned nrows/nthreads rows, except possibly for the last thread, which may get assigned more rows.

b. GPU. Choice of parameter:

number of threads per block: After several experiments, we found that 1024 threads per block works well for all kernels. Therefore, we define the maximum threads per block as 1024, and for each kernel we choose min(1024, width * height) as the number of threads per block.

number of blocks: will be discussed separately in the following part, since we chose different numbers for each kernel.

c. Computation of min/max value

When our program finishes the filtering of the image, it will perform the reduction to calculate the minimum and the maximum of all pixels. After that the min and max will be sent to normalization. During this process, we created 2 global arrays (global_min, global_max) to store the min and max values, i.e. the first element of both of them will be the final min and max.

In each reduction call, the program creates an array named *sdata*, and it also creates m blocks that each thread will process 1 element in the arrays. All of the threads in one block will calculate the min and the max for this block.

And the program used a while loop to continually update the min and the max, by keeping calling reduction, until the number of blocks hits 1.

d. Kernel 1
   i. number of blocks:
      *(width * height + num_threads - 1) / num_threads*
      Assign enough blocks to ensure Kernel 1 is able to contain all the elements of the input.
   ii. We assigned each thread one single pixel to process, and access the input image matrix in column major method.

e. Kernel 2
   i. number of blocks: (same as Kernel 1)
      *(width * height + num_threads - 1) / num_threads*
      Assign enough blocks to ensure Kernel 2 is able to contain all the elements of the input.
   ii. We assigned each thread one single pixel to process, and access the input image matrix in row major method.

f. Kernel 3
   i. number of blocks:
      (height + num_threads - 1) / num_threads
      The number of blocks is based on the number of rows per thread and threads per block.
   ii. We assigned each thread multiple pixels, consecutive rows to process, row major.

g. Kernel 4
   i. number of blocks:
      *((width * height + pixels_per_thread - 1)/pixels_per_thread + (num_threads - 1)) / num_threads;*
      Note: We defined pixels_per_thread as 8 (default value) as specified in the handout.
      The number of blocks is based on the number of pixels per thread and threads per block.
   ii. We assigned each thread multiple pixels. Kernel 4 processed consecutive pixels on different threads which then stride across the pixel array until the entire image is filtered.

h. Kernel 5

i. number of blocks: (same as Kernel 4)
*((width * height + pixels_per_thread - 1)/pixels_per_thread + (num_threads - 1)) / num_threads;*

ii. Since Kernel 4 performs best among the first 4 kernels, we decided to implement our Kernel 5 based on it.

iii. We used a different reduction method here. In each block we used warp reduction to compute partial results. And we used the shuffle down operation to optimize the calculation. When all calculations are done, our program will collect the results and do a final reduction, whose results will be collected by the thread 0 and be stored in the output array (first element).

2. Results:

Configuration: Filter: 5*5, image size 4mb, time unit: ms

2.1 Kernel 1

CPU_time(ms)	Kernel	GPU_time(ms)	TransferIn(ms)	TransferOut(ms)	Speedup_noTrf	Speedup
75.010109	1	4.196320	6.339584	10.307456	17.88	3.60
56.726337	1	4.168032	6.452128	10.327520	13.61	2.71
71.494377	1	4.179328	6.308064	10.284064	17.11	3.44
74.931358	1	4.178336	6.338272	10.238272	17.93	3.61
77.806053	1	4.175008	6.333888	10.184608	18.64	3.76

Average:

CPU_time	Kernel	GPU_time	TransferIn	TransferOut	Speedup_noTrf	Speedup
71.193646	1	4.1794048	6.3543872	10.268384	17.034	3.424

The average speedup of Kernel 1 for GPU_time is 17.034, i.e. (CPU_time/GPU_time = 17.034) as we can see from the above table. Notice that we are not discussing the overall Speedup here since transfer time can vary sometimes due to lab machine states. More details will be discussed in part3.

2.2 Kernel 2

CPU_time(ms)	Kernel	GPU_time(ms)	TransferIn(ms)	TransferOut(ms)	Speedup_noTrf	Speedup
75.010109	2	1.473920	4.585728	1.987424	50.89	9.32
56.726337	2	1.473824	4.663808	2.032896	38.49	6.94
71.494377	2	1.480480	4.613312	2.008992	48.29	8.82
74.931358	2	1.476672	4.555648	1.988896	50.74	9.34
77.806053	2	1.479424	4.706624	2.024352	52.59	9.48

Average:

CPU_time	Kernel	GPU_time	TransferIn	TransferOut	Speedup_noTrf	Speedup

71.193646	2	1.476864	4.625024	2.008512	48.2	8.78

The average speedup of Kernel 3 for GPU_time is 48.2, i.e. (CPU_time/GPU_time = 48.2) as we can see from the above table. More details will be discussed in part3.

2.3 Kernel 3

CPU_time(ms)	Kernel	GPU_time(ms)	TransferIn(ms)	TransferOut(ms)	Speedup_noTrf	Speedup
75.010109	3	55.160671	4.583232	1.989344	1.36	1.22
56.726337	3	55.145790	4.527424	1.971328	1.03	0.92
71.494377	3	55.134720	4.789344	2.079232	1.30	1.15
74.931358	3	55.310398	4.707936	2.037472	1.35	1.21
77.806053	3	55.296322	4.709280	2.027360	1.41	1.25

Average:

CPU_time	Kernel	GPU_time	TransferIn	TransferOut	Speedup_noTrf	Speedup
71.193646	3	55.209580	4.6634432	2.0209472	1.29	1.15

The average speedup of Kernel 3 for GPU_time is 1.29, i.e. (CPU_time/GPU_time = 1.29) as we can see from the above table. More details will be discussed in part3.

2.4 Kernel 4

CPU_time(ms)	Kernel	GPU_time(ms)	TransferIn(ms)	TransferOut(ms)	Speedup_noTrf	Speedup
75.010109	4	1.478112	4.674112	2.061440	50.75	9.13
56.726337	4	1.475072	4.551552	1.970304	38.46	7.09
71.494377	4	1.482080	4.754048	2.053856	48.24	8.62
74.931358	4	1.480992	4.714528	2.009088	50.60	9.13
77.806053	4	1.481472	4.684416	2.017504	52.52	9.51

Average:

CPU_time	Kernel	GPU_time	TransferIn	TransferOut	Speedup_noTrf	Speedup
71.193646	4	1.479546	4.675731	2.022438	48.114	8.696

The average speedup of Kernel 4 for GPU_time is 48.114, i.e. (CPU_time/GPU_time = 48.114) as we can see from the above table. More details will be discussed in part3.

2.5 Kernel 5

CPU_time(ms)	Kernel	GPU_time(ms)	TransferIn(ms)	TransferOut(ms)	Speedup_noTrf	Speedup
75.010109	5	1.211456	4.717792	2.058848	61.92	9.39
56.726337	5	1.209728	4.515136	1.980256	46.89	7.36
71.494377	5	1.219296	4.694624	2.023008	58.64	9.01
74.931358	5	1.214400	4.710528	2.011200	61.70	9.44
77.806053	5	1.212864	4.682016	2.023392	64.15	9.83

Average:

CPU_time	Kernel	GPU_time	TransferIn	TransferOut	Speedup_noTrf	Speedup
71.193646	5	1.2135487	4.6640192	2.0193407	58.66	9.01

The average speedup of Kernel 5 for GPU_time is 58.66, i.e. (CPU_time/GPU_time = 58.66) as we can see from the above table. More details will be discussed in part3.

3. Analysis

Findings:

a. In summary, our most optimized kernel (kernel 5), performs 18% faster in GPU time in large image size(4mb), compared to kernel 2 (best of 1 - 4).
b. Kernel 1 shows relatively good speedup compared to the CPU implementation, which is expected since a large amount of threads facilitated by GPU.
c. Kernel 2 shows much better speedup compared to Kernel 1. Even though Kernel 2 used a similar approach, they both process single pixels by individual thread, the row major performs better. In other words, the consecutive threads process consecutive pixels of the image would help the program to take advantage of memory coalescing.
d. Kernel 3 performs worst among all kernels. This might be because its method neither has the advantage of being row-major, nor the benefit from processing resources of GPU to assign rows to threads, which reduces the concurrency of the method.
e. Kernel 4 also performs very well among the first 4 kernels. In theory, Kernel 4 should perform better than Kernel 2, since striding not only allows it to achieve memory convalescing like Kernel 2, but also limits the amount of blocks which reduce the overhead.

4. Optimization Attempts:

a. Memory types

One possible optimization is changing the memory type of some variables.
For example, we've tried to change the filter to constant variable for Kernel5. Then we don't need to pass it as a parameter to run_kernel5 and other methods.
Ideally, this should have some positive impact. But when we did it, the runtime got slower.

b. stream

We've tried to implement Kernel 5 in CUDA stream. In theory it should work, as our plan was: split the memory copy into b parts (eg. 3 parts) running on different streams. And split the kernel execution into different streams separately. Since the program spends a lot of time on transferIn and transferOut, optimizing the memory copy and writing back to disk should help the efficiency.

However, this method could only achieve a little optimization and is very inconsistent for image size 875kb - 4mb.