CSC367 A4 Report

April 2018

Alex Teoh (teohshao) & Jiushan Yang (yangjius)

---

1. Implementation

Selecting parameters

- **Number of threads per block:** To find the optimal number, we experimented many times to see the impact on GPU time. For all kernels, the setting of 1024 threads per block works well. In addition, if the device has a lower limit of threads per block , our program will select that limit (*properties.maxThreadsPerBlock*) instead for all kernels.
- **Number of thread blocks**: There are some variations across the kernels. For *kernel1* and *kernel2*, we assign enough blocks such that they can contain all the elements of the input. Hence, we have 1 element per thread. Given that we are testing with input size up to 150M, our usage of blocks is within the limit as specified in the Cuda Programming Guide[1]. For *kernel3*, the number of blocks is calculated with number of rows per thread and threads per block. For *kernel4*, the number of blocks is calculated with number of pixels per thread and threads per block.
- **Others**: For *kernel3*, each thread handles one row of the input by default. For *kernel4*, each thread process 8 pixels by default. We tried several settings for these numbers. These settings usually yield the lowest GPU time with various inputs.

Reduction

We seperated reduction out as a stand-alone kernel to simplify our code structure. Immediately after filtering of image, our program performs reduction to calculate the minimum and the maximum of all pixels. Then the minimum and maximum are fed into normalization.

The reduction of minimum and maximum are done in tandem, which saves some computation time from re-iterating through pixels. In each block, we use warp reduction to compute partial results. In warp reduction, we use the shuffle down operation to optimize the calculation. After all the calculations are done in each block, we collect the results and do a final reduction by calling the kernel again. The final results are collected by thread 0 and stored as the first elements in the output arrays.

---

[1] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications

Kernel5

- We noticed that the sample filter is frequently read by all threads. The constant memory is optimized for the case where all threads in a warp read the same memory address. Therefore, we decided to use constant memory for storing the filter.
- In this implementation, we are no longer passing the filter as an argument to kernel5, as we store filter in the constant memory. In addition, we pass in a texture object instead of a pointer to the global memory, since the source image is stored in texture memory.
- Since the memory access pattern exhibit good spatial locality when we access the pixels in an image, we stored the source image in texture memory instead of global memory. However, we encountered several issues when exploring this option.
    - Problem 1: 1D Texture has size limitation (2^{27} = 134,217,728[2]). If the input image is large enough (> 135M), the texture object will not be created.
    - Solution 1: Use 2D texture, which allows much larger source images.
    - Problem 2: 2D texture has limitation on the size in each dimension. For example, an 1x1000000 image will break the limit since its height is huge.
    - Solution 2: Use either 1D or 2D Texture, depending on the situation : if width and height of image is less than the maximum width and height in 2D texture, then our program uses 2D texture; otherwise, it uses 1D texture.

2. Results

The following results are collected from Bahen Centre lab machines (BA 3175) during quieter hours. Experiments are run multiple times and there are negligible variations. We displayed the typical results from these experiments.

Results with small input (37M) and 3x3 filter

CPU_time(ms)	Kernel	GPU_time(ms)	TransferIn(ms)	TransferOut(ms)	Speedup_noTrf	Speedup
252.129929	1	217.396835	26.185535	50.562782	1.16	0.86
252.129929	2	29.712320	22.182272	49.990814	8.49	2.47
252.129929	3	226.742279	22.190752	50.129665	1.11	0.84
252.129929	4	28.062271	24.944288	54.030655	8.98	2.36
252.129929	5	19.348064	22.577761	50.376545	13.03	2.73

* Comparing to the best of kernels 1-4, *kernel5* performs 31% faster in GPU time and 9.5% faster overall.

[2] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications

Results with large input (~150M) and 3x3 filter

CPU_time(ms)	Kernel	GPU_time(ms)	TransferIn(ms)	TransferOut(ms)	Speedup_noTrf	Speedup
970.152649	1	1200.645142	86.946625	197.348007	0.81	0.65
970.152649	2	129.301987	98.685951	194.731491	7.50	2.30
970.152649	3	1107.424683	97.190529	195.293533	0.88	0.69
970.152649	4	133.395004	98.694756	193.756454	7.27	2.28
970.152649	5	87.218689	96.406944	196.798950	11.12	2.55

*Comparing to the best of kernels 1-4, *kernel5* performs 32.5% faster in GPU time and 10% faster overall.

Results with small input (37M) and 9x9 filter

CPU_time(ms)	Kernel	GPU_time(ms)	TransferIn(ms)	TransferOut(ms)	Speedup_noTrf	Speedup
1039.296021	1	652.263611	23.535072	51.284321	1.59	1.43
1039.296021	2	110.154083	22.463264	52.185825	9.43	5.62
1039.296021	3	657.308228	22.600639	51.448513	1.58	1.42
1039.296021	4	116.909027	22.447552	51.996384	8.89	5.43
1039.296021	5	76.119331	22.792896	51.588322	13.65	6.91

*Comparing to the best of kernels 1-4, *kernel5* performs 30.9% faster in GPU time and 18.5% faster overall.

Results with large input (~150M) and 9x9 filter

CPU_time(ms)	Kernel	GPU_time(ms)	TransferIn(ms)	TransferOut(ms)	Speedup_noTrf	Speedup
4650.545410	1	3606.664551	91.595970	198.289413	1.29	1.19
4650.545410	2	520.395142	101.883263	198.518661	8.94	5.67
4650.545410	3	3172.479248	90.145248	193.697861	1.47	1.35
4650.545410	4	514.465210	101.317345	192.544540	9.04	5.75
4650.545410	5	228.613480	87.939934	194.523010	20.34	9.10

*Comparing to the best of kernels 1-4, *kernel5* performs 55.5% faster in GPU time and 37.7% faster overall.

3. Findings

3.1. In summary, our most optimized kernel, kernel5, performs at least **30%** faster in GPU time (i.e. pixel processings + reduction + normalization), compared to the best of kernels 1-4. In the case of filtering large inputs with a 9 by 9 filter, the optimization on kernel5 shines even more. *kernel5* outperforms *kernel4*, the best among kernels 1-4, with **55.5%** faster in GPU time and **37.7%** in total time.

3.2. The benefit of using constant memory to store filter is not immediately obvious, when the filter size is 3 by 3. But when we switched to the 9 by 9 filter, constant memory has significantly improved the performance of *kernel5*. This makes sense because there are 81 elements in 9 by 9 filter, which is a magnitude larger than 3 by 3 filter. Each thread of our kernel accesses the 81 elements multiple times in a read-only fashion. This access pattern fits naturally with the constant memory paradigm.

3.3. Programming cuda codes requires us to be aware on the various limitation of cuda APIs. While testing *kernel5*, we ran into an issue where GPU time is reported as 0.0. Upon close reading on the specifications, we found out that linear texture (1D) has a size limitation. Inputs, that are around 135 - 150 MB in size, exceed this limitation. Therefore, we had to read further and figure out ways (e.g. 2D pitch) to work around cuda’s limitations.