Roy Liu

March . 16th, 2020

Problem 1:

Write both OpenCL and CUDA codes for performing (N*N) convolution for both direct convolution and matrix multiply over the provided sample video stream. For performing convolution implement two approaches: (1) Generic Matrix Multiply (GMM), (2) Direct Convolution. Consider four-Byte integer for convolution filter parameters. Compare overall execution time (only focus on kernel execution time), between two implementations for N=3,5,7,9. Notice that you perform the same kernel over Red, Green and Blue Channels. Make sure to plot your results. (30pts)

Answer:

I compose the programs according to the requests under CUDA10.2, details about code are illustrated in the read file in the package. The hardware been tested is NVIDIA GPU GTX1060. As a conclusion in this experiment, I found:

- Programs compiled by CUDA runs faster than what is from OpenCL, in this specific algorithm and hardware;
- Execution time grew up when the filter size increased, both in CUDA and OpenCL;
- GMM achieves better performance than direct convolution

Firstly, Since CUDA is specifically designed to support NVIDIA GPU, while OpenCL more compatibly opens to wider range of hardware such as CPU, GPU, FPGA and DSP, so CUDA has higher execution efficiency in this testing environment.

Secondly, with bigger filter size in convolution execution, there are more data movements occurred and worse locality to utilize, this makes the kernels run slower with bigger filter sizes.

Moreover, by using GMM algorithm to unroll the data from two dimensions to one dimension, both CUDA and OpenCL gain the performance improvement from matrix multiplication against direct convolution, which is beneficial from lessening memory access outweigh and wasteful storage costs.

Fig. 1 depicts the situation stated above.

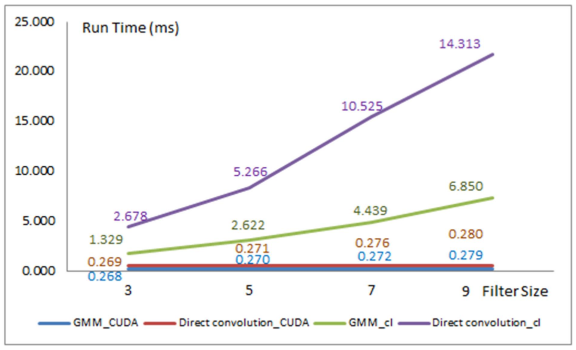


Fig. 1 Run time by CUDA and OpenCL under various filter size

Problem 2:

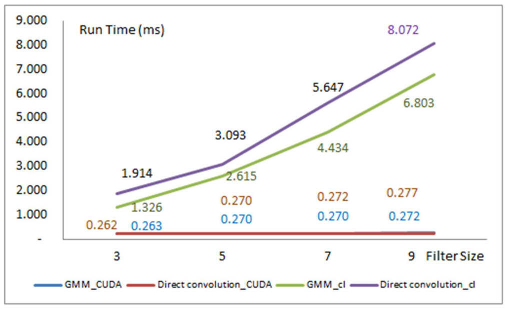
Repeat problem 1, this time try to use local (shared memory) to improve the performance. Compare and plot your results for direct convolution and GMM. (30pts)

Answer:

In this experiment, I applied the shared memory to re-do the testing. Following the same trend by using various filter sizes, the kernels spent more execution time when filter size increased. Meanwhile, further improvement was presented by using shared/local memory, such as shown in Fig 2a.

Though there is little improvement in CUDA by applying shared memory, which I think there is less improvement room due to relatively higher efficiency in CUDA, The speed up in OpenCL is significant. For instance, in direct convolution with filter size 7, the run time shorten from 10.52 ms to 5.64 ms, which is around 86% speed up.

Please refer Fig. 2b.



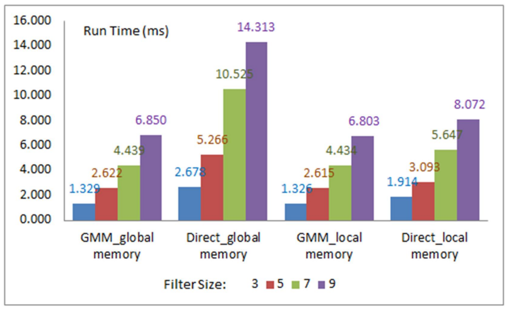


Fig. 2a Run time by various filter

Fig. 2b Global memory vs Local memory

Problem 3:

In this problem, we will try a true CNN layer (first layer of SqueezNet). In this case we will perform 96 different convolutions per each input channel which translates 288 total (7*7) different convolution filters. The convolutions will perform with stride of two. The results of convolution across three input channels will add up and finally kernel produces 96 output channels (which also called feature vector). Notice that convolution filters varied across the R, G and B channels. Consider four-Byte integer for convolution filters parameters. Implement a basic implementation of GMM for CNN layer (without local memory optimization) and compare the performance over entire stream video, only focus on kernel execution time. You need to write both Cuda and OpenCL No need to consider zero padding when performing the convolutions. Make sure to compare and plot the results (40pts)

Answer:

The code is composed in file of *gemmconv_globalMM_96ch_cuda.cu* and *gemmconv_96ch.c* + *gemmconv_globalMM_96ch_kernel.cl*.

I found the run time in OpenCL increased significantly after using 96 filters. I think this inflation comes from great data latency increase. CUDA's increased relatively less, I think the reason is CUDA compiler did the optimization which lead the improvement space squeezed.

Run time comparison please refer Fig. 3 (Filter size =7).

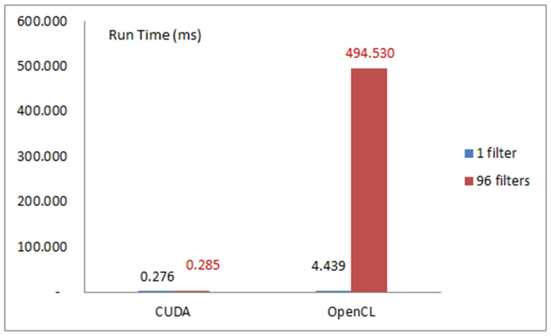


Fig. 3 Run time comparison between 1 filter and 96 filters by CUDA and OpenCL

Problem 4:

Repeat problem 3, this time try to use local (shared memory) to improve the performance. Make sure to write your code both at CUDA and OpenCL abstractions. Compare and plot your results over entire video stream, only focus on kernel execution time. (50pts)

Answer:

In this problem, I applied the local/shared memory in OpenCL and CUDA programs. It shows the execution time improvement by using local/shared memory, which result aligns what has be observed in problem 2.   Fig. 4 illustrates the comparison (Filter size =7).

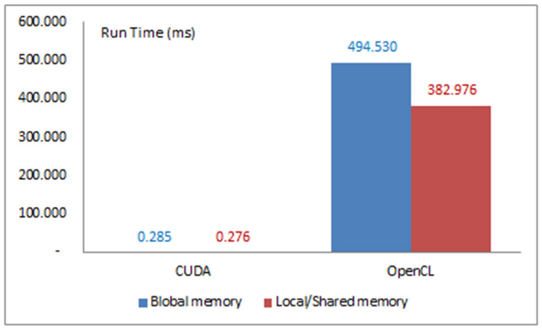


Fig. 4 Run time comparison between global vs local/shared memory by CUDA / OpenCL

Problem 5 (bonus, extra 100 points):

Repeat problem 5, this time try to use batching idea across the video frames. Make sure to write your code both at CUDA and OpenCL abstractions. Compare and plot your results over increasing batching size (at least 5 different batching sizes), only focus on kernel execution time. (100pts)

Answer:
In this experiment, I composed the batch function in both CUDA and OpenCL, with GMM and local/shared memory configured as well. Various of batch size from 2 , 4, 6, to 120 was set up and run, Fig. 5 presents the result.
It was found the run time drops down when increased the batch size. The reason is more batch of data was feed, higher usage rate kernel got, which increased kernel's efficiency.
On the other hand, this run time testing is only counted what occurs in kernel, if add CPU's performance, some overhead from batch of data transposing will offset overall improvement.
Also, from the figure, it illustrates the speed up rate is dropping down. With my hardware, it is expected if feed more images to test, we will get a bigger batch size which reflects the best performance from kernel execution perspective.

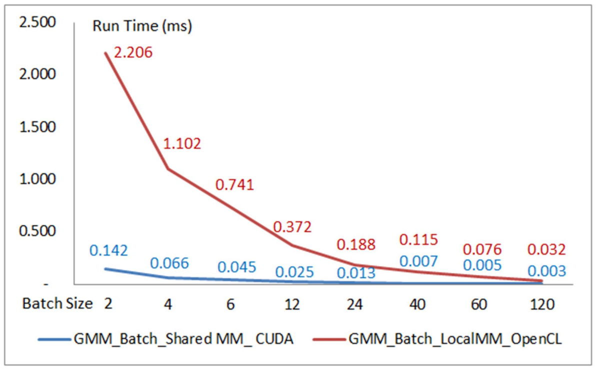


Fig. 5 Performance with various batch size