Spring 2018 ECE 8823: GPU Architectures **Moumita Dey**

Assignment 2

Project Description

In this project, 2D convolution to perform image blur and edge detection on RGB images has been performed. As convolution of large images involves many repetitive computations, performing them on the CPU takes up a lot of time. Launching a kernel on a GPU to perform the same highly parallelizes convolution and reduces the compute time. Different approaches have been used to finally arrive at the algorithm that has the minimum kernel execution time.

Implementation

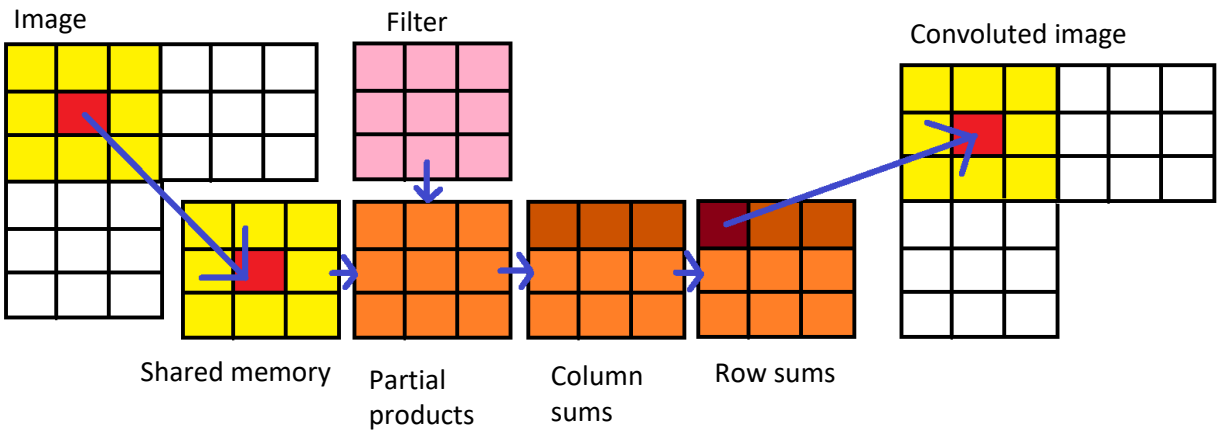
- Kernel description

  In a broad sense, the kernel receives the image and filter inputs from the host, performs the convolution and returns the convoluted image to the host. For the computation of each output pixel, the input pixel at the corresponding location, along with all its neighboring pixels (apron) is loaded, partial multiplications are performed with the corresponding filter values, and the sum of all these partial products is stored in the output pixel. Shared memory has been used majorly for storing the image pixels for each thread block and since the filter is accessed by all threads, it was stored in the constant memory. The techniques used within the kernel have been described below.

- Optimization techniques

  1. Multi-thread single-output

     Since each output pixel requires 9 individual partial products to be summed, it was hypothesized that if each output pixel has 9 threads to compute the final convoluted value, it would remove the serialization on each thread and hence this method would be faster. Each thread loads the pixel values into the shared memory and performs partial multiplication with its own filter value. Now, the first three threads perform additions of its own columns, followed by the first thread adding up the results from the previous stage. This has been illustrated below.



However, probably because of a lot of threads spawned, there was a bottleneck and this approach instead took a lot of time. It might be concluded that more threads might be harmful in terms of kernel execution time.

2. Output stationary approach

   Each thread is responsible for performing all computations required to compute the convoluted value for that pixel, thereby reducing the total number of threads needed. To reduce the number of global memory references, thread blocks of size of 32*32 were defined each having a shared memory size of 34*34 due to apron. First, each thread brings its own image pixel. After this the left, right, top and bottom aprons are loaded. Special cases are used to load the corners of the apron. Once all the loads are done, each thread computes the partial products, adds them up and

stores to the output matrix. This approach even though has more thread divergence, was much faster than the first approach.

3. Output stationary with precomputed apron
   If the host computes and stores the input matrix along with the apron containing all zeros and passes this to the kernel, the kernel doesn't have to perform checks of image boundary and can simply copy the contents from the input image. This will reduce some thread divergence from the previous case and an improvement in the kernel execution time was observed.

4. Output stationary with precomputed apron and direct index references
   In the previous approach, a for loop with pragma unroll was used during the partial product and addition computation. However, if the entire for loop was manually broken down and written with direct index references, it was observed that this further reduced the kernel execution time.

Result Analysis and Observations

The timings of all the cases discussed above have been tabulated below. The timings of the output stationary approaches have been compared against an application using just the global memory to compare the effect of usage of constant and shared memory. For the all the approaches, a maximum possible size of thread block was used. Since 1024 is the maximum possible number of threads in each thread block, for the first approach, a thread block size of 30*30 was used [since each output pixel requires a 3*3 thread usage], and for all output stationary approaches, a thread block size of 32*32 was used. All times listed are in microseconds.

Convolution images	Multi-thread Single-output	Output-stationary Global memory	Output-stationary Apron in kernel	Output-stationary Apron in host with loop unroll	Output stationary Apron in host with direct index
Bird-Filter1	503.875	70.8082	66.6585	52.2045	47.0244
Bird-Filter2	504.062	71.0216	66.6603	52.1767	47.0841
Cat-Filter1	504.016	70.7783	66.7022	52.3176	47.1274
Cat-Filter2	504.002	70.8843	66.6204	52.2887	46.8384
Soccer-Filter1	504.096	70.533	66.7051	52.3714	47.05144444
Soccer-Filter2	503.942	71.0725	66.6823	52.1851	47.0038
Tiger-Filter1	504.215	70.9268	66.7382	52.3049	46.9677
Tiger-Filter2	504.068	70.6372	66.7907	52.2917	47.0494