Program 5:

Convolution on GPGPUs

Glenn Contreras
g_contreras@u.pacific.edu
University of the Pacific

***Abstract*—Image convolution is the process of applying a small mask or convolution matrix to an image to obtain a desired effect–i.e., sharpening, smoothing, blurring or edge detection. Depending on the size of the image, convolving can be a high-compute intensive task as it requires multiple operations for every pixel of the image. This process is intrinsically well-suited for parallelization, since each operation is independent from the rest. This assignment involves leveraging the raw power of GPGPUs using NVIDIA's CUDA framework to optimize the image convolution process, and analyze the performance gains with various GPU optimization.**

I. Introduction

The purpose of this assignment is to translate the serial code that was provided to us using the CUDA framework. The CUDA programming model created by NVIDIA provides a relatively easy way to program for GPU architectures so that we can leverage the massive computing power that GPUs have to offer.

The serial code is from program-2 that computes the horizontal and vertical gradients. The objective of program-5 is to translate the serial code to run on the GPU architecture via CUDA with various levels of optimization.

II. Implementation

A. Optimization 1

The first optimization involves solely the GPU kernel. We perform the convolution on the GPU and transfers the image into the global memory of GPU device. Each block has 16 threads and the number of thread blocks depends on the size of the image.

Size	L0	L1	L2
1024	537.5	700.5	433.6
2048	970.3	1237.3	974.9
4096	3273.9	3582.1	3307.1
7680	10939.3	11515	11257.1
10240	19261.1	19782.1	19333.3
12800	29869.2	30611.5	29902.8

Table 1: Run times for the multi level optimizations

B. Optimization 2

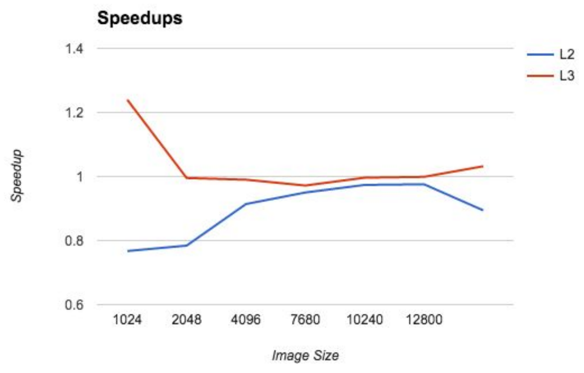
The second tier optimization took advantage of the L2 shared memory. Global memory access times can take 300 to 600 clock cycles compared to the L2 cache which only takes about 30 cycles[1]. To utilize the L2 cache we took advantage of collaborative loading from the adjacent threads in the same thread block. We allocated a 16x16 shared memory block dynamically. Instead of implementing the ghost rows to make sure each thread has its necessary values, we used the leap of faith method that takes advantage of the memory management process the GPU device automatically performs. We make the assumption that when the other threads load in a value from the global memory to perform the convolution, it automatically stores that value in the L2 cache. If a thread needs a value that resides outside of the local shared memory block, it will make a global memory access but the GPU will recognize that value is already cached. This simplifies the code tremendously while still benefiting from the speed up. To be sure that all threads load in their respective values before convolution, we sync the threads before hand.

Size	L2	L3
1024	0.767309065	1.239621771
2048	0.7842075487	0.9952815673
4096	0.9139610843	0.989960993
7680	0.9500043422	0.9717689281
10240	0.973663059	0.9962655108
12800	0.9757509433	0.9988763594
Average	0.8941493404	1.031962522

Table 2: Speedups compared with the first optimization: L0.

C. *Optimization 3*

Optimization 3 relies on the program to use the fast math libraries. We simply set the compiler flag use_fast_math and this forces the compiler to use the faster but less accurate math libraries.



III. Results

The serial code took 34463955ms to run on an image of 10240 by 10240 px size while the GPU version took only 29869.2ms to run. That is a 64000 times speedup achieved with GPGPU architecture. For the tier 2 optimization, the speedup on average was 0.894 ms for all image sizes with the highest speedup achieving around 0.98 ms. The speed up from the using the fast math libraries is about 1.03 ms.

II. Conclusion

Utilizing the GPGPU architecture to speed up convolution was a tremendous success. The multi-tiered optimizations did not lead to a major improvement in the speed up, but the overall performance from CUDA drastically improved the performance of the program.

References

[1] V. Pallipuram, "NSF/IEEE-TCPP CDER Curriculum Initiative on Parallel and Distributed Computing," 2013