Analysis of the Execution Time Variation of CUDA with Canny Edge Detection

Marco Lin
University of the Pacific

Hight Performance Computing
m_lin23@u.pacific.edu

***Abstract*—The purpose of this report is to compare the difference of execution time between serial execution and GPUs execution. In order to analyze this, we will implement Canny Edge Detection first, and then using CUDA method to run on GPUs device.**

Keywords—C, Canny Edge Detector, CUDA, GPU

I. Introduction

The purpose of this report is to analyze the accelerating by designing a serial code and then using CUDA method to speedup. In Canny Edge Detector, there are several kernels, such as convolution, Suppressed, and Hysteresis. In those kernels, the program needs to get and calculate the value of each pixels. They take up execute time the most, thus by executing on GPUs device, we can effectively to observe the reducing of execution time.

II. Implementation

A. Canny edge detector with CUDA

We used this function to detect the edge of image. In order to effectively reduce our execution time. We used GPUs to implement our accelerating. On GPUs device, we can use global memory and shared memory to speed up our program.

In convolution part, especially, we compared the execution time between 2 kernels. One was only using global memory. Another was using shared memory and global memory. We only used shared memory for convolution because during the calculation, some data were out of the bound that the threads need to access global memory to get the data what it needs. We can restore our data to shared memory to enhance their speed.

For other part of functions, like Suppressed and Hysteresis, we only use global memory for implementation. This is due to there is not calculation which needs the data from different block.

In the Figures 1 to 6, it appears that the correlation between speedup and different number of threads per block. For example, when we used 32x32 threads per block with shared memory to performance 12800x12800 size image with 1.1 sigma, the speed up is up to 300. The range of speed up is from 8 (8x8 threads per block) to 300 (32x32 threads per block).

B. Parallelization Methodology

By setting two time zones: end-to-end time and computation time to measure the execution time in serial code and CUDA code. By setting different numbers of threads per block: 8x8, 16x16, and 32x32, we can observe the difference of speed up. We also set two sigmas: 0.6 and 1.1, 5 different image sizes: 1024x1024, 2048x2048, 4096x4096, 10240x10240, and 12800x12800, and two kernels for convolution: shared memory and global memory.

III. Results and Analysis

We have the figures from 1 to 6 to show our Speed-up time and results in each sigma, number of thread per block, and different kernel with different size of images. The unit of time is Microsecond(MS)

Average execution time

	1024	2048	4096	10240	12800
Shared Memory					
Sigma0.6					
8	220861	253909	371463	925849	1155146
16	218932	252631	332743	872653	1130093
32	212987	250849	330533	808013	995454
Sigma1.1					
8	240670	303909	406451	951625	1344589
16	230670	272631	377132	925475	1194621
32	229755	260849	349340	917140	1034977
Global memory					
Sigma0.6					
8	256450	321803	391463	955849	1555146
16	224506	235265	382743	902653	1330093
32	214758	244586	350533	888013	1005454
Sigma1.1					
8	287562	269633	421463	975849	1744589
16	258621	240991	412743	922653	1494621
32	226540	248484	390533	908013	1234977
Serial code					
Sigma0.6	1848690	7326289	29574118	179618848	286855145
Sigma1.1	1933301	7932161	32115823	191671199	310436054

Figure 1.

Speed up

	1024	2048	4096	10240	12800
Shared					
Sigma0.6					
8	8.37	28.85	79.62	194	248.33
16	8.44	29	88.88	205.83	253.83
32	8.68	29.21	89.47	222.3	288.17
Sigma1.1					
8	8.03	26.1	79.02	201.41	230.88
16	8.38	29.09	85.16	207.11	259.86
32	8.41	30.41	91.93	208.99	299.94
Global					
Sigma0.6					
8	7.21	22.77	75.55	187.92	184.46
16	8.23	31.14	77.27	198.99	215.67
32	8.61	29.95	84.37	202.27	285.3
Sigma1.1					
8	6.72	29.42	76.2	196.41	177.94
16	7.48	32.91	77.81	207.74	207.7
32	8.53	31.92	82.24	211.09	251.37

Figure 2.

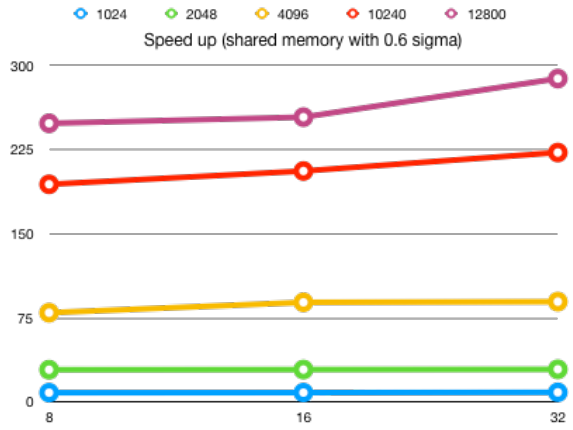

Figure 3.

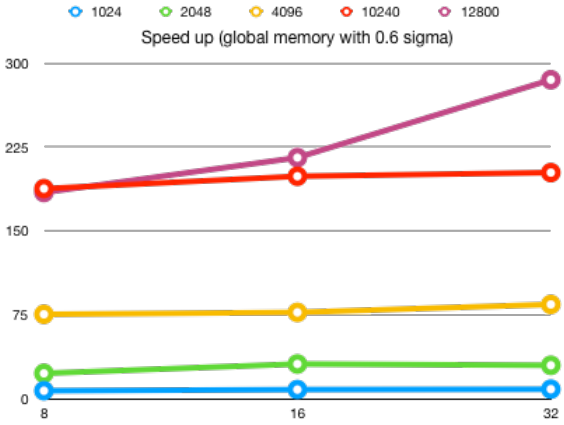

Figure 4.

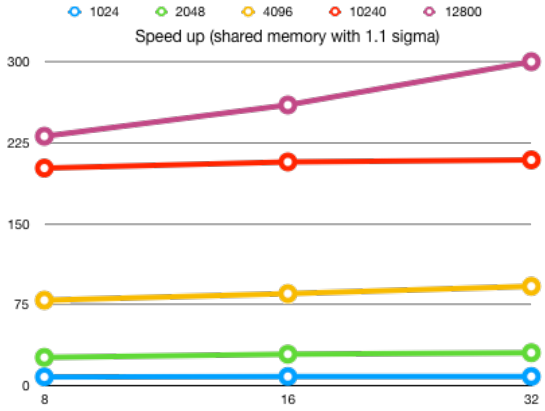

Figure 5.

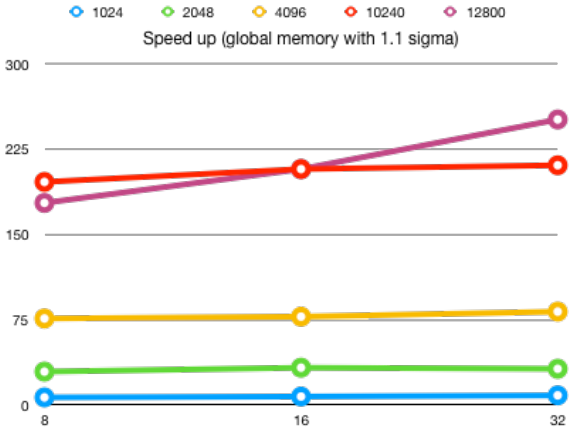

Figure 6.

Occupancy of each Multiprocessor

8x8	50%
16x16	63%
32x32	50%

Figure 7.

The execution time are affected by sigma value, size of image, number of threads per block, and different kernel using. According to the execution time we got, it shows that using shared memory is faster than only using global memory. By the different numbers of thread per block, we can calculate occupancy of each multiprocessor rate. The occupancy rates in here appear that the rates are justified.

IV. Conclusion

In this project, we could learn how to use different numbers of thread, block and shared memory to optimize our execution efficiency. The most import thing we need to be careful is the bound of access. In my project, the suppression function was the reason causing the segmentation fault. But this problem only showed up when I run with 10240x10240 and 12400x12400 size images. It was running successfully with the smaller image. But, by using the memory checking tool , we can easily to find where the problem was.