Griffin Maxwell
11/21/17
CSE 5441
12:45

Programming Assignment 4 – CUDA and Ohio Supercomputer

1) Part 1: Matrix Multiplication

Part 1 of this experiment compared two implementations of an algorithm that multiplies a large 1024x1024 matrix of floats with its transpose: one serial, and one parallelized with CUDA. A sample of the console output of this program is at the end of this section. In order to compare the performance of these two implementations, timing results were gathered over 5 trials with the `clock_gettime()` utility in the `<time.h>` library. This timing information was then used to calculate the GFlops per second benchmark for each implementation. Since each iteration of the does a single calculation composed of two floating point operations (one multiplication and one sum), the number of total floating-point operations required for the matrix multiplication is:

$$GFlops = \frac{iterations_{i\ loop} * iterations_{j\ loop} * iterations_{k\ loop} * Flops_{per\ iteration}}{1G}$$

$$= \frac{1024 * 1024 * 1024 * 2}{1G} = \frac{1G * 2}{1G} = 2\ GFlops$$

The table below shows the timing results and the GFlops benchmark for all of the trials.

	Time to execute using `clock_gettime()` (seconds)		**Benchmark (GFlops per second)**	
	Serial	CUDA	Serial	CUDA
Trial 1	8.5806	0.3492	0.2331	5.7274
Trial 2	8.5715	0.2839	0.2333	7.0447
Trial 3	8.6382	0.2971	0.2315	6.7317
Trial 4	8.5818	0.3430	0.2331	5.8309
Trial 5	8.6010	0.2944	0.2325	6.7935
Average	**8.5946**	**0.3135**	**0.2327**	**6.4256**

The CUDA parallel version of this program ran significantly faster than the serial version. On average, the CUDA version executed in 1/27 the time, so it was able to do over 6 GFlops/second of work versus the serial version, which could only do less than 0.25 GFlops/second on average. The program has built in checks after the matrix multiplications occur to make sure that both the serial and parallel versions computed the same result matrix. Based on the console output from each trial, the versions were verified to compute the exact same matrix.

The CUDA compute structure used in this experiment had 1 grid of 1024 blocks each having 1024 threads. When the code was first being tested, the compute structure was 1 grid and 1 block with 1024 threads, but this was modified once preliminary testing showed that increasing the number of blocks decreases the time. This makes sense because having more blocks means more of the GPU's hardware is being utilized at one time to do these calculations in parallel. The specific values of 1024 threads per block was chosen because the matrix dimensions are 1024x1024. Each block of threads could do an entire row's worth of calculations, meaning that the innermost (k) loop of the serial implementation could be taken out of the kernel function and only two levels of looping were required. Furthermore, having 1024 blocks means that even though the code sets up the kernel to use cyclic distribution of the inner (j) loop, there are enough blocks that each block only executes one iteration of this loop. However, while having 1024 threads per block is intentional and makes the code simpler by eliminating an entire nested loop, having 1024 blocks is somewhat arbitrary. In testing, having 2 blocks was almost twice as fast as having only one block, but having 1024 blocks was not significantly faster than having 10. The main reason that 1024 was finally chosen as the number of blocks in the compute structure was because the compiler didn't stop me from choosing such a large grid, and 1024 is a round number that matches the parameters of the problem, even if it is not the most ideal grid dimension.

Sample Console output:

```
-bash-4.2$ ./lab4p1
********************************************************************************
lab4p1: serial vs. CUDA matrix multiplication.

Matrix dimensions: 1024x1024
CUDA compute structure:
|-- with 1 grid
    |-- with 1024 blocks
        |-- with 1024 threads per block

Performing serial matrix multiplication.
Performing parallel matrix multiplication.
Checking that the two resulting matrices are equivalent.
The two resulting matrices are equivalent!

Timing results:
|-- The serial algorithm took    8.6382 seconds
|-- The parallel algorithm took    0.2971 seconds
********************************************************************************
```

2) Part 2: Sobel Edge Detection

Part 2 of this experiment involved two implementations of a Sobel edge detection algorithm: one serial, and one parallelized with CUDA. Because both algorithms executed in nearly the same time with the small provided image coins.bmp, a larger (1920x1200) bitmap named larger.bmp was used to get a better idea of the how the execution times of the two algorithms differ. As in Part 1, the `clock_gettime()` utility was how time was tracked and five trials were done with each algorithm. The timing and convergence results from a test trial with coins.bmp and the five trials with larger.bmp are on the next page, and sample console output from this program is located at the end of this section.

		Time to execute using `clock_gettime()` (seconds)		Convergence Threshold	
		Serial	CUDA	Serial	CUDA
coins.bmp	Trial 0	0.2369	0.2788	51	51
larger.bmp	Trial 1	3.1291	0.4997	27	27
	Trial 2	3.1194	0.4984	27	27
	Trial 3	3.1281	0.4996	27	27
	Trial 4	3.1145	0.5007	27	27
	Trial 5	3.1554	0.4975	27	27
	Average	**3.1293**	**0.4992**	**27**	**27**

The convergence threshold reached by both implementation was the same in all trials and for all images, indicating that the serial and CUDA implementations of the Sobel edge detector likely produce the same results. Additionally, visual comparison of the output bmp files showed the same black and white image, lending further evidence that the two versions produce identical output.

The CUDA compute structure used in this experiment had 1 grid of 32 blocks each having 128 threads. This CUDA organization was determined mostly by trial and error. The first tests had 4 blocks and 4 threads per block, just to verify that the code worked correctly. The execution time for this first guess was a little over a second, about twice as much as the final version of the code. Then, the number of blocks and number of threads per block were slowly increased back and forth, and the execution time started to go down. There was a point where small number of blocks and a huge number of threads per block (2x2048) were tested, and this showed to be no faster, and at times slower, than the final version of the code. This is likely because each block had to schedule many more warps, which caused overhead in completing the edge detection. After much tinkering, it showed that having 32 blocks and 128 threads per block made the program run about as consistently quick as it could.

In conclusion, there was a very noticeable improvement when using the GPU and CUDA as opposed to writing a single threaded serial program. While Part 1 showed a 27x increase in speed,

Part 2 demonstrated that the CUDA version ran on average 6.27x faster than the serial version. Considering that the small test image, coins.bmp, showed almost no speedup compared to the larger image, the maximum speedup could be increased with even larger test images that fully utilize the dimensions of the GPU cores. Additionally, the times for the CUDA implementation were very consistent, with all trials falling within about 0.1 seconds of each other. This is fairly different than parallelization with pthreads, where consecutive executions of the same program could be 30 seconds to a minute different. However, this timing difference is likely exaggerated by the fact that the pthread applications were run on a server with many users vying for computing time, while these applications were run on a dedicated supercomputer node. Still, it is easy to see that the potential for speeding up parallelizable sections of code is very large with executing device code on a GPU with CUDA.

Sample Console output:

```
-bash-4.2$ ./lab4p2 larger.bmp s.bmp p.bmp
Image Info ::
        Height=1200 Width=1920
********************************************************************************
lab4p2: serial vs. CUDA Sobel edge detection.

Input image: larger.bmp   (Height: 1200 pixels, width: 1920 pixels)
Serial output image:      s.bmp
CUDA output image:  p.bmp

CUDA compute structure:
|-- with 1 grid
    |-- with 32 blocks
        |-- with 128 threads per block

Performing serial Sobel edge detection.
Performing CUDA parallel Sobel edge detection.
Time taken for serial Sobel edge detection: 3.129137
Convergence Threshold: 27

Time taken for CUDA Sobel edge detection: 0.499651
Convergence Threshold: 27
********************************************************************************
```