Matthew Silva

Professor Alvarez

CSC 415

3 May 2018

Assignment 3 Report

CUDA Implementation

The CUDA implementation of my assignment was quite similar to the sequential version in terms of the work completed. I create one CUDA thread for each pixel to be worked on, leaving out the edges of the grid. The raster and a copy of the raster are sent over to the GPU so that each thread can compute the single pixel of work it was assigned. The filtering operation is nearly identical to the one in the sequential version. The only difference is that the indices for the pixel to be worked on are computed using the thread block dimensions and the thread index. I used the dim3 CUDA type to define the size of my thread blocks, which were square and varied in size depending on command line arguments. The dim3 CUDA type had data members (x and y) which made it easy to compute the pixel each thread should work on. The thread blocks overflow past the edge of the grid if the size of the workspace in the image does not divide evenly by the thread block size. Threads always check to see whether they are working on a pixel that actually exists before going into the computations of their kernel. At the end of all kernel executions, the completed image gets copied back to the CPU.

My speedups were quite significant in the CUDA version, reaching over 100 in the largest problem sizes. This is with the caveat that the speedup was far less when the CUDA library was not preloaded. When the library was not preloaded, loading the library took about 2 seconds, which made all but the largest problem size just as slow or a lot slower than the sequential. I explored GPU programming briefly during my work with OpenACC, where I used the PGProf profiler to reveal that compute utilization as as low as 0.1% when running code compiled for an Nvidia GPU. The timeline

(Fig 1) gives an effective visualization of how much time it takes to get the GPU ready for computations.

Figure 1: GPU Overhead, (Matthew Silva)

By looking closer at the very small segment of the time where the GPU is actually doing work, I also found that much of the GPU's time was spent doing memory copies (Fig 2).

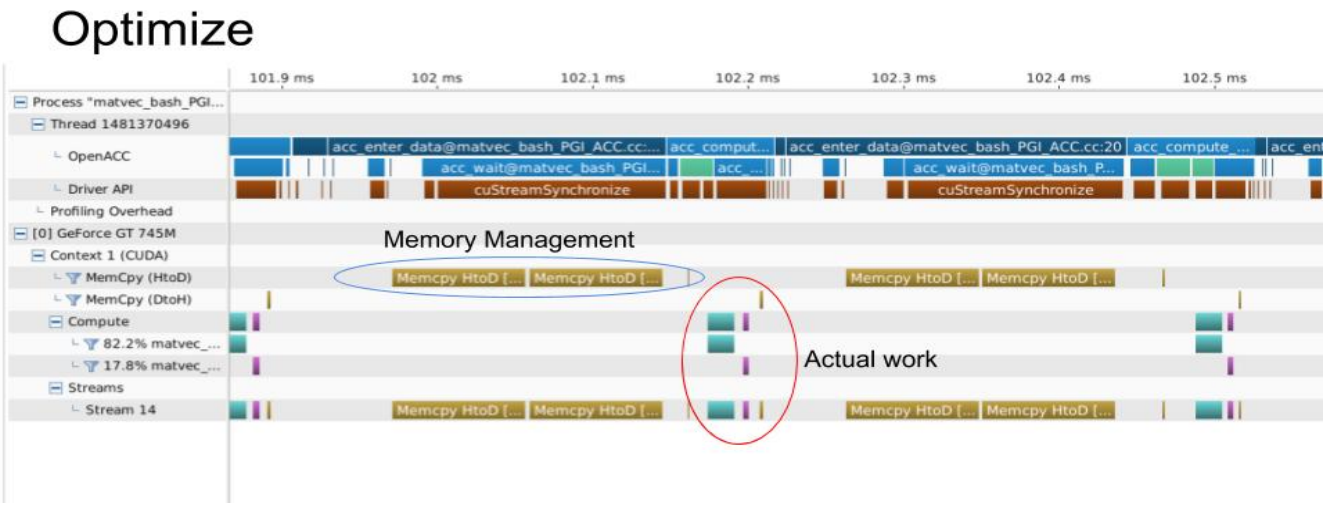

Figure 2: GPU Memory Management, (Matthew Silva)

While the above example was generated with a different program, the idea that memory management in GPUs takes a long time seemed to hold in this assignment as well, evident by my speedups from using shared memory instead of GPU global memory.

Shared Memory

My regular CUDA implementation suffered from slow memory copies because of its use of the global memory of the GPU. The global memory is large but fairly slow. Each thread block also has a smaller memory shared among every thread in the block. This memory cannot fit the entire image like global memory can, but it can fit all of the pixels that a reasonably sized thread block must use for computations. My thread blocks varied in size from 4 by 4 to 48 by 48. Loading the relevant part of the image into shared memory also has the advantage of reducing the overall number of loads from the GPU global memory. Almost all of the pixels get used by threads multiple times, so loading all of the useful pixels into a thread block's shared memory in parallel will reduce that number of loads significantly, bringing them all into the cache-like shared memory where threads can quickly access both the pixels they loaded and the pixels other threads loaded.

My implementation for shared memory was essentially that every pixel was responsible for loading the pixel it would compute into the shared thread block memory. Threads on the edges and corners of the area their thread block was responsible (not necessarily the edges of the image) would also load the pixels on the edges and corner pixels, scaling these operations to account for the filter size. The threads of the block all synchronize after loading the relevant part of the image so that they do not begin computations before a complete copy of the image section is stored in the thread block's shared memory. I had a brief discussion with my classmate Alex, who informed me that there was a way to explicitly require that certain variables of constant size get stored into the memory of the thread block. We thought that this type of operation would be useful for the filter, since it would be difficult to decide which thread in the thread block would be the one to load the filter (the first thread could always be the one to do this but this could damage the parallelism that is important to the kernel). Another implementation that greatly preserves the parallelism of the kernel would be creating one thread for every pixel instead of every pixel that is going to be worked on. Loading the image into shared memory

would then be as easy as computing the coordinates each thread is assigned to and loading that pixel from the image into the shared memory. However, this implementation would have suffered from creating excess threads, as well as many threads being totally useless during the parallel execution of the kernel (many would simply sit and watch as the other threads actually do work).

Speedups

My CUDA speedups seemed to vary in size greatly between executions. I got between 123 and 219-fold speedups for my shared memory version, and between 80 and 119-fold speedups for my global memory version. I have provided my most recent version here (the last time I ran the code before submitting). Some code motion in my global memory version seemed to speed things up significantly. It is interesting that a GPU running over 10,000 threads at once can only achieve a 200-fold speedup. I wonder if the overhead is much lower for devices like ASICS that run all day on the same job. Overall, the abstraction and speed provided by the CUDA library was quite impressive.

Image Filter Combination	4*4 Thread Block	8*8 Thread Block	16*16 Thread Bloc	32*32 Thread Block	48*48 Thread Block
earth-2048.tif / Gaussian-blur	34.49	34.27	34.61	33.20	17.67
earth-2048.tif / Identity-7	69.54	69.64	69.67	118.20	119.24
earth-8192.tif / Gaussian-blur	49.08	48.58	48.60	49.62	49.04
earth-8192.tif / Identity-7	123.98	123.65	123.52	123.41	123.28

Figure 4: Table of Speedups With Shared Memory, (Matthew Silva)

Image Filter Combination	4*4 Thread Block	8*8 Thread Block	16*16 Thread Bloc	32*32 Thread Block	48*48 Thread Block
earth-2048.tif / Gaussian-blur	15.847	18.441	18.901	20.124	21.788
earth-2048.tif / Identity-7	43.814	61.842	63.469	64.499	99.911
earth-8192.tif / Gaussian-blur	18.164	21.843	22.183	22.400	26.104
earth-8192.tif / Identity-7	47.547	69.700	72.665	73.547	119.133

Figure 3: Table of Speedups Without Shared Memory, (Matthew Silva)