Investigating GPU Optimization to Decrease Processing Runtime

Alexander Ayerdi

My goal for the summer was to test if whether or not using a GPU processor would improve performance of a particular algorithm being used by our Astronomy Department. The GPU processor to run the tests was an NVIDIA GeForce GT 540M[1] running on Ubuntu 12.04LTS (linux). The processor was activated by implementing a layer of NVIDIA's CUDA (Compute Unified Device Architecture) on top of an already coded algorithm in C/C++. The algorithm used is called a PTMCMC Algorithm which the astrophysicists use to calculate log-likelihoods of gravitational wave data. The data is used in a separate part of the program to predict and describe the gravitational waves, of which is the focal point of the astrophysicists' investigation.

The problem with the algorithm was that the running time on the CPU processor (Intel i7 chipset at 2.70Ghz)[2], the algorithm would bottleneck on the trig-functions *sin* and *cos*. Since they were inside the in-code loop, they were slowing down the calculation at a minimum of 7872 times on one run of the algorithm. This increases proportionally as the sample rate increases. With no changes to the code and ran at 1,000,000 iterations with a 16384 sample rate, the average runtime on the CPU was **3hrs 32min 11sec**.

CUDA Method:

When the loop was converted to a CUDA algorithm (with same parameter conditions as the control) for use by the GPU, the average runtime was **2hrs 4min 11sec**. This effect is due to what is best put as "dividing and conquering" on the GPU multiprocessors. The GPU allows for the removal of the loop altogether and turns it into one line called a kernel execution. This kernel execution is the command that sends a piece of code to the GPU to be processed and returned. In essence, instead of running a line of code 7872 times with the CPU, the GPU sends the same line of code to 7872 *threads* on its multiprocessors and computes them simultaneously. Therefore, we can use parallelism to solve the trig-function bottleneck!

The only trade-off with this method is that one thread from a GPU is much less powerful than a single CPU core. Therefore, the idea is to use as many threads as possible on the largest dataset possible to create a situation where the threads end up working together faster than the single CPU core. The number of threads and data size is never obvious for the best GPU optimization strategy because each problem is different. My strategy should not necessarily be used for other code as it may end up being worse than how it may have started.

As a solution to this problem, I found that using the GPU is most effective when using sample rates of 16384 and any amount of iterations or using any sample rate (powers of 2) but at least 800,000

[1] When mentioning "the GPU" I am referring to these specifications.
[2] When mentioning "the CPU" I am referring to these specifications, also considered the control.

iterations. This increase in productivity averages at about 14% when using those parameters. There is a chance of increase in productivity if the data size increases, however the algorithm has a cap of 16384 sample rate which doesn't fully take advantage of the GPU multiprocessor.

OpenMP Method:

In addition to my research on GPU optimization my supervisor recommended that I work on researching CPU multithreading because it has the prospect of performing much better than the GPU. The reason is because, as mentioned previously, each CPU core is much more powerful than a single GPU thread. During any such run of the code, the CPU is only using one of its cores as needed. However, I was able to use OpenMP to parallelize the CPU into separate threads, depending on the amount of cores the computer has. Since the computer I ran the tests on was an Intel i7 QuadCore, the best utilization of multithreading to "divide and conquer" the code, was into four pieces. Since the CPU cores are much more powerful than the GPU threads I expected a much larger increase in productivity.

When using the same conditions as the control test, I was able to achieve an average processing time of **1hr 36min 17sec** with the OpenMP implementation. Ultimately this is a massive improvement and is scalable to larger runs.

In addition to this test I decided to combine CUDA and OpenMP but it had little improvement. With the same conditions as the control I was able to achieve an average processing time of **1hr 58min 11sec** when using the two methods in combination. In effect, the CUDA part was inhibiting OpenMP from running at its maximum potential.

I conclude that between the CUDA and OpenMP method, using OpenMP is the best for optimizing the PTMCMC algorithm because it utilizes efficient multithreading, requires little to no change of the source code, and is highly supported open source software. OpenMP is also easily implemented in any system because its code can be overlooked by the compiler if the user doesn't want to use it. Furthermore, due to its portability to any program, I plan to continue with my work in the Astronomy Department on further projects implementing OpenMP.