Biography

I am a second year nuclear engineering Ph.D. student studying computational methods and reactor physics. I have worked with large-scale parallel codes, and am familiar with MATLAB, but I have never really had to deal with the low-level technicalities of parallel computing. I hope to learn how to implement efficient algorithms and how to look for different kinds of parallelism in my problems.

Monte Carlo on GPUs

One very popular transport algorithm in nuclear engineering is Monte Carlo, since it is able to work without discretization and is able simulate the physics almost exactly. Since the method tracks individual particles, it can take a very large number of them to build reliable statistics in desired quantities. The time cost of using it on big problems means running on a large cluster. Since all the particles can be simulated independently, the method is very well suited to parallelization. This is usually done using MPI and scales linearly to hundreds of CPUs. Some people do not have access to large clusters, however, but can afford a different massively-parallel piece of hardware: GPUs. There are problems with utilizing GPUs to their full capability when using Monte Carlo, however. Divergent branches are executed serially on GPUs, making the traditional Monte Carlo scheme of tracking a particle per thread a problem. After doing a little searching, I found a paper about overcoming this divergent branch problem using some organizational overhead. This paper was written to address thread divergence in general, not specifically to Monte Carlo methods. The programs are written in CUDA, which is heterogeneous and has both CPU-executed and GPU-executed parts.

The approach taken is to remap the data the threads access so that every thread in a warp takes the same fork in conditional statements. There are two ways to do this: make a remapping vector that redirects the data calls (“reference redirection”), or rearranging the data itself (“data transformation”).

Figure 1: Towards thread coherence.
Figure 1 shows these two approaches. Building the remapping vector or doing the data rearranging is done on the CPU before the GPU is invoked. This is done in a pipelined fashion where the CPU is performing the transformation for the next step while the GPU is computing the current step. These transformations need to be done “on-the-fly” since the divergent branches cannot be known a priori. This is especially true in Monte Carlo, where the conditional statements are based on random numbers.

Benchmarks of different problems were done using a workstation with a NVIDIA Tesla 1060 GPU and dual-socket quad-core Intel Xeon E5540 CPUs. The amount of thread divergence in each benchmark varies, and ranges from 50-100%, depending on the size of the threadblocks. On average, pipelined remapping sped up an unoptimized, 100% divergent implementation by 1.1-1.4 times, showing that the overhead cost of remapping was more than paid for by the “revenue” generated by eliminating serial GPU threadblock execution. Speedup dropped from 1.4 to 1.1 in case where the number of divergent thread blocks was reduced from 100% to 50%. This is expected since the profits of remapping are reduced when there isn’t much remapping to be done.

The paper does not compare GPU to CPU implementations of Monte Carlo codes, or any Monte Carlo codes for that matter, so I cannot comment directly on the benefits of GPUs beyond their financial savings. It only discusses the problem of thread divergence, which has serious implications for using GPUs to accelerate Monte Carlo simulations. The paper shows how to make Monte Carlo more efficient on GPUs and points out the importance of coalesced memory requests and thread coherence when using GPUs. There is hope for accelerating stochastic methods like Monte Carlo with GPUs, but its takes a lot of work and careful data management routines. Once the work is done, however, the codes would be able to use top supercomputers, like Tianhe-1A, which derive a lot of their computing power from GPUs.

References