GPU acceleration of SAMURAI particle tracking simulation

J. Gao[*1,*2]

The possibility of graphical processing unit(GPU) acceleration of the trajectory simulation of particles passing through SAMURAI was evaluated. To obtain A/Z in particle identification plot, information such as the flight path and rigidity obtained from this type of simulation is necessary. Usually, this tracking simulation is the most time-consuming step in the analysis of SAMURAI data.

As with most other steps in nuclear physics data analysis, this particle tracking simulation is performed event by event. Therefore the simulation could be accelerated by distributing the events to large amount of threads of a GPU to process in parallel. In principle, all the event-by-event analysis could be accelerated by parallel computing and could take advantage of a GPU.

Another advantage of a GPU is its special cache structure. In contrast to the linear structured cache in a CPU, the cache in a GPU could have a higher-dimensional layout.[1] Therefore, when interpolating in the magnet field map, the GPU could fetch the neighboring mesh point with fewer cache misses.

In this work, a simplified task is designed to evaluate the performance of trajectory simulation on a CPU and GPU. The standard program used in data analysis takes the position and angle before and after the magnet and provides the rigidity as the output. It iterates several times to obtain a certain rigidity that reproduces the position and angle measured in the experiment. This simplified version takes the position, angle, and rigidity before the magnet and outputs the position and angle after the megnet without any iterations. The CPU version is modified from the code used for the particle identification of SAMURAI11 data.[2]

The program uses a fourth-order Runge-Kutta method to simulate the trajectory of a particle in the SAMURAI magnet. Both the CPU and GPU versions of the program were developed. The test was performed on a server with 4 Intel Xeon Gold 6136 CPUs (each has 12 cores/24 threads), 128 GB memory, and one Nvidia Titan V GPU. The CPU version code is written in Go programming language and optimized using the AVX2 assembly.[3,4] The GPU version is written in C++ and CUDA. 1024000 events were fed into the programs, and each version ran 3 times.

The results are summarized in Table 1. The timer starts immediately after the field map is loaded into the main or GPU memory and the input data are loaded into the main memory, and it stops as soon as the calculation finishes, which means the hard disk I/O time is excluded so that the time of calculation is isolated.

Table 1. Test results of CPU and GPU versions of the simplified simulation. The GPU could accelerate the simulation by a factor of 5 to 13 times approximately.

code	CPU 1	CPU 2	GPU
configuration	32 coroutines	126 coroutines	320×32
test #1	13.292 s	5.277 s	0.999 s
test #2	13.309 s	5.319 s	1.004 s
test #3	13.317 s	5.378 s	1.011 s
average	13.306 s	5.324 s	1.004 s

In Table 1, the CPU 1 code enabled 34 threads and allocated 32 coroutines for simulation. The CPU 2 code enabled all the 128 threads and allocated 126 coroutines for simulation. The GPU version divides the input data into 100 groups, each of which is passed to a device function using a configuration of 320 thread blocks and 32 threads per block. Different groups are processed asynchronously to hide the time of data transfer between the main memory and device memory.

For SAMURAI11 data, it takes 3.15 iterations on average to obtain the rigidity and a maxmium of 5 iterations. For the CPU1 and CPU 2 configurations, the time for simulation should be $13.306 \times 3.15 = 41.91$ s and $5.324 \times 3.15 = 16.77$ s, respectively. If we consider the worst case for the GPU, where each thread block contains an event that need 5 iterations, then the simulation time is $1.004 \times 5 = 5.02$ s, which is 8 or 3 times faster than the CPU version, depending on the number of threads used in the CPU code.

In conclusion, with a reasonable GPU cost and coding effort, the simulation could be made significantly faster.

References
1) CUDA C++ Programming Guide, `docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`
2) J. Gao *et al.*, in this report.
3) Intel 64 and IA-32 Architectures Software Developer's Manual, `software.intel.com/content/www/us/en/develop/articles/intel-sdm.html`
4) A Quick Guide to Go's Assembler, `golang.org/doc/asm`

[*1] School of Physics, Peking University
[*2] RIKEN Nishina Center