International Conference on Mathematics, Computational Methods & Reactor Physics (M&C 2009)
Saratoga Springs, New York, May 3-7, 2009, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2009)

STUDY OF THE ACCELERATION OF NUCLIDE BURNUP CALCULATION USING GPU WITH CUDA

Shota Okui
Research Laboratory for Nuclear Reactors, Tokyo Institute of Technology
2-12-1-N1-17, O-okayama, Meguro-ku, Tokyo, 152-8550, Japan
okui.s.aa@m.titech.ac.jp

Yasunori Ohoka and Masahiro Tatsumi
Nuclear Fuel Industries, Ltd Fuel Engineering and Development Department
1-950, Asahiro-Nishi, Kumatori-cho, Sennan-gun, Osaka, 590-0481 Japan
ya-ohoka@nfi.co.jp; tatsumi@nfi.co.jp

ABSTRACT

The computation costs of neutronics calculation code become higher as physics models and methods are complicated. The degree of them in neutronics calculation tends to be limited due to available computing power. In order to open a door to the new world, use of GPU for general purpose computing, called GPGPU, has been studied [1]. GPU has multi-threads computing mechanism enabled with multi-processors which realize mush higher performance than CPUs. NVIDIA recently released the CUDA language for general purpose computation which is a C-like programming language. It is relatively easy to learn compared to the conventional ones used for GPGPU, such as OpenGL or CG. Therefore application of GPU to the numerical calculation became much easier. In this paper, we tried to accelerate nuclide burnup calculation, which is important to predict nuclides time dependence in the core, using GPU with CUDA. We chose the 4th-order Runge-Kutta method to solve the nuclide burnup equation. The nuclide burnup calculation and the 4th-order Runge-Kutta method were suitable to the first step of introduction CUDA into numerical calculation because these consist of simple operations of matrices and vectors of single precision where actual codes were written in the C++ language. Our experimental results showed that nuclide burnup calculations with GPU have possibility of speedup by factor of 100 compared to that with CPU.

Key Words: Graphics processors, GPU, GPGPU, CUDA, nuclide burnup

1. INTRODUCTION

The computation time has been increasing for reactor physics calculations where more detailed models are employed. In order to overcome high computation costs, parallel computing on a PC cluster is one of easy options to select. In this paper, however, we discuss a different solution, namely General Purpose computation using on Graphics Processing Unit (GPGPU). Graphics Processing Unit (GPU) has many multi-processors and allows implicit parallel computing in the manner of data-parallel paradigm. A comparison of performance of GPU and CPU is shown in Fig. 1; GPU has great potential for scientific calculations and hardware developing speed is surprisingly fast. Software development to support GPGPU is also active. In a few years ago, GPGPU was usually used in the combination with the OpenGL which is relatively difficult since it, required high programming skills and hardware insights [1]. However GPU programming for general purpose has been changed after the release of NVIDIA's CUDA. CUDA is a C-like language environment that enables programmers to write programs easily to solve complex numerical problems. CUDA have been applied for neutronics analysis and acceleration of that was achieved [2]. Therefore one can enjoy GPGPU powered by CUDA with much more gentle learning curves.

In this work, we tried to accelerate nuclide burnup calculation using the 4th-order Runge-Kutta method with CUDA as the first step. Its method is easily referred to parallel programming because equations can be divided in simple operations of matrices and vectors.

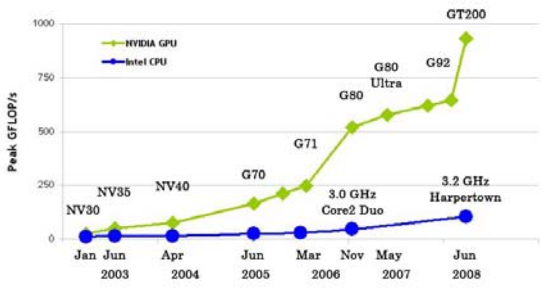

Figure 1. Development of GPU and CPU performance [3]

2. GPGPU WITH CUDA

Programmers use C or C++ language for CUDA, NVIDIA provides dedicated compiler - GPUs adapt for parallel computation because those have many multi-processors. For example, NVIDIA GeFroce 8800 GTX has 16 multi-processors. Each multi-processor is viewed as multi-core device that is enable to execute some operates for large data in parallel. Thus, CUDA treats GPUs as the SIMD processors. CUDA provides means of heterogeneous computing using both of CPU and GPU. We need to select between GPU and CPU in the each calculation. Commonly GPU is suitable for the large size calculation which can be treated as parallel computing. Other calculations with small amount of calculations are executed by CPU. For example, in the burnup calculation, GPU is used in solving differential equations. Preparation constants or some other trivial operations and calculations are executed by CPU. Manners of programming for GPU are somewhat different from one for CPU since architectures found in those processors are quite dissimilar. GPU is mounted on the video card and has a number of multi processors and four kinds of memories, these are classified as global memory, shared memory, constant cache or texture cache. A system bus is involved in the data transfer, memory transfer between CPU and GPU is necessary. The memory transfer via a system bus, which is the PCI Express x16 in this case, is much slower compared with calculation speed. Mostly, global and shared memory units are used for numerical calculation. The global memory can be referred from all multi processors in a GPU, where access costs quite high because of low bandwidth between the global memory and processors. Each multi-processor has a shared memory unit which cannot be referred from other multi-processors; the shared memory is much faster than global memory and can be recognized as somewhat similar to L1 or L2 cache of CPU. Since the shared memory is not automatically used in CUDA programming, the programmers need to write control of them in order to retrieve performance of GPU, which make it difficult for programmers to select right approach and "true" parameters in algorithms.

3. FEASIBILITY STUDY

Comparison of computational performance was done between GPU and CPU; NVIDIA 8800GTX was used for GPU hardware and Core 2 Duo E6600 for CPU. Coding for both of GPU and CPU was done in C++ where numerical operations were treated in single precision. At first, product of matrix and vector was explored. The result is shown in Fig. 2. GPU performance is lower than CPU one because computation load for GPU cannot compensate the cost for memory transfer. Calculation time of GPU case contains two memory transfer time between CPU and GPU side before and after its actual computation; firstly, data of matrix and vector are transferred from CPU to GPU to load into VRAM that is a local memory managed by GPU. The GPU is initiated to perform matrix-vector product operations with a pre-loaded code fragment running on GPU within the framework of CUDA. Secondly, vectors calculated from matrix-vector product are extracted from VRAM to be transferred to CPU via the system bus, which is PCI-Express x 16 in this case. Since these memory transfer costs much higher than calculations, we cannot get good performance in this case.

Next, the product of two matrices, **AB**=**C**, was explored in order to make clear the relationship between calculation and memory transfer costs. The result of that is shown in Fig. 3. In this calculation, two matrices of **A** and **B** are transferred to GPU memory before calculation and a matrix of **C** are transferred to CPU side after calculation. The GPU performance of this calculation is better than previous calculation despite large data transfer. Hence, the relationship between the amount of calculation and memory transfer for above two cases is shown in Table I. The amount of calculation over memory transfer is about 2 in case of the product of matrix and vector. In this case, it cannot get the GPU potential because memory transfer cost is much larger than calculation cost. On the other hand, in case of the product of 2 matrices, the amount of calculation over memory transfer is 2m/3. GPU exercise its real performance if the matrix size is adequately large. When we write the CUDA programming, the amount of calculation over memory transfer is important value to estimate how performance is displayed.

One should be careful when discussing GPU performance since performance may be much poor than that by CPU with little amount of data to be processed because of high cost in memory transfer. Therefore the large amount of data set in a calculation is essential to educe high performance from GPU. Another noted point is that GPU performance becomes higher as matrix size is larger. This is because multi-thread computing is fairly and effectively performed utilizing multi-processors in GPU.

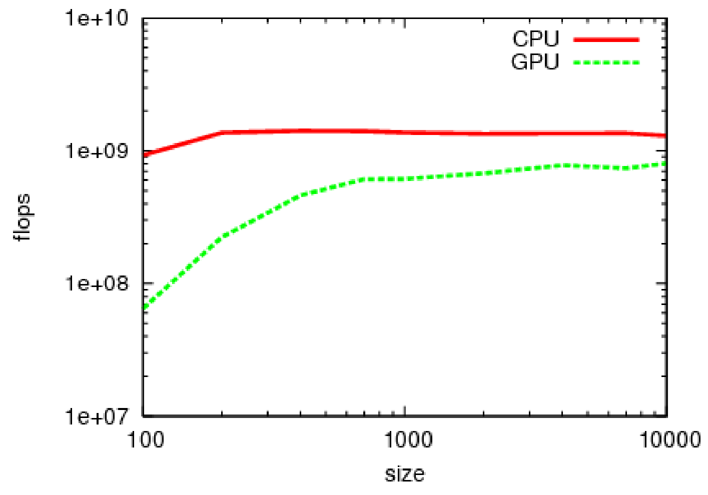

Figure 2. Product of matrix and vector.

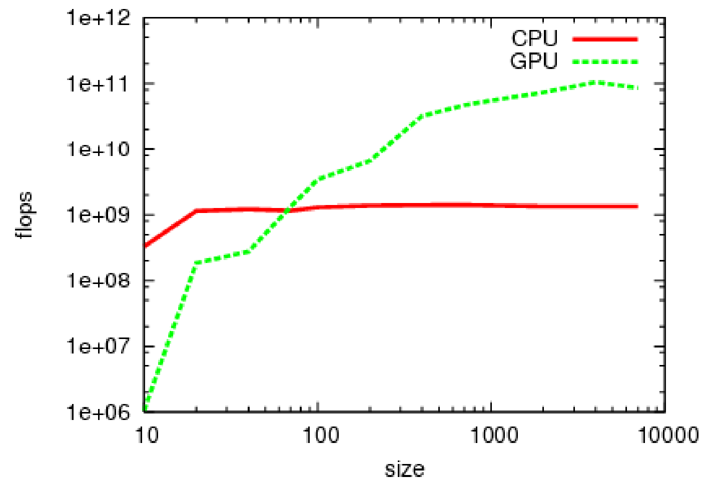

Figure 3. Product of two matrices.

Table I. Relationship between amount of calculation and memory transfer.

Product	Amount of calculation	Amount of memory transfer (input, output)	Calculation / memory transfer
Matrix and Vector	$2m^2$	m^2+m, m	$\cong$ 2
Two Matrices	$2m^3$	$2m^2$, m^2	$2m/3$

*m is the vector length.

4. APPICATION OF CUDA TO NUCLIDE BURNUP CALCULATION

Next, as a realistic application, we applied CUDA to nuclide burnup calculation shown in Eq. (1) which analyzes time dependence of number densities of nuclides, with decay, capture, fission and (n,2n) reactions. The 4th-order Runge-Kutta method was chosen to solve the differential equation. The creation of matrix M in the Eq. (2) and resolution of Runge-Kutta method were computed on GPU. The size of matrix M corresponds to number of nuclides which is about one hundred or so. However, from the viewpoint of the feasibility study, it needs to enlarge matrix size to derive high performance from GPUs. Hence we introduced another dimension in the matrix for spatial meshes. Numbers of meshes in which burnup calculation is performed are considered in a computation. With enough amount of data set, multi-processors in GPU can be effectively used.

From the feasibility study, we have to be careful about memory transfer between CPU side and VRAM. The flow chart of this burnup calculation with CUDA is shown in Fig. 4. At first, The four kinds of yield matrices, which are decay γ_d, capture γ_c, fission γ_f and (n,2n) reaction γ_{n2n}, and decay constants λ are transferred from CPU to GPU before burnup calculation loop. At the same time, initial number densities of all nuclides are transferred from CPU to GPU. The number densities calculated at the each step are stored on the VRAM and stored number densities are extracted from VRAM to CPU at the end in order to reduce memory transfer latencies. In burnup calculation, three kinds of reaction rates must be updated by transferring them from CPU to GPU via PCI-Express bus at every step because neutron fluxes change as materials are depleted. Since transferred data sets are not matrices but vectors, that cost is relatively small. In this calculation, these reaction rates treated as constant for the simplified model were transferred to the VRAM at each step. The amount of calculation over memory transfer become large by the design as mentioned above.

$$\frac{d}{dt}N_i = -\left(\lambda_i + R_a^i\right)N_i + \sum_j \left(\gamma_d^{j\to i}\lambda_j + \gamma_c^{j\to i}R_c^j + \gamma_{n2n}^{j\to i}R_{n2n}^j + \gamma_f^{j\to i}R_f^j\right)N_j \tag{1}$$

$$\frac{d}{dt}\vec{N} = \mathbf{M}\vec{N} \tag{2}$$

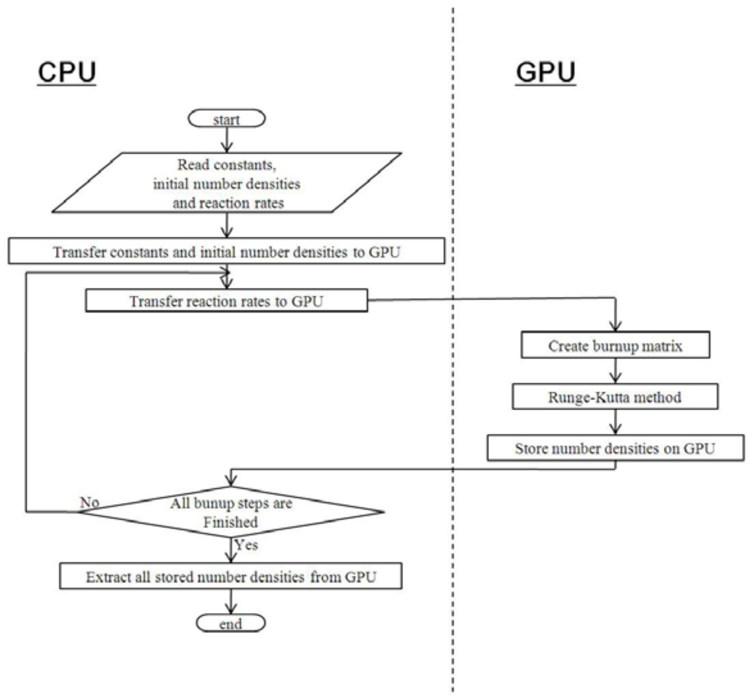
CPU
GPU
start
Read constants, initial number densities and reaction rates
Transfer constants and initial number densities to GPU
Transfer reaction rates to GPU
Create burnup matrix
Runge-Kutta method
Store number densities on GPU
No
All bunup steps are Finished
Yes
Extract all stored number densities from GPU
end

Figure 4. Flow chart of the burnup calculation with CUDA.

Comparison on computation time GPU and CPU is shown in Table II where the number of nuclides is 96. Both codes were written in C++ and treated in single precision. The calculation time of GPU contains memory transfer time. As the mesh size is larger, GPU performance becomes higher, up to about 100 times faster.

GPU calculation treats single precision. Hence we compared the difference of GPU single precision calculation and CPU double precision calculation, in which total time steps is 1000 and time intervals are 0.01 day. That difference is very small and the absolute value of the error of the major nuclides is shown in Fig. 5. The error is less than or comparable to 0.01%.

Table II. Processing time on CPU and GPU

Mesh size	**200**	**1000**	**4000**	**8000**
CPU [sec]	4.63e+1	2.32e+2	1.01e+2	2.03e+2
GPU [sec]	1.15	3.41	10.6	20.0
CPU / GPU	40.3	68.0	95.7	102

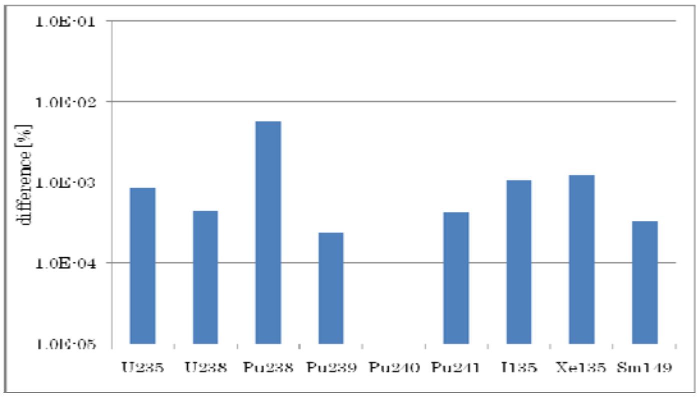

Figure 5. Difference of number densities with GPU and CPU.

5. CONCLUSIONS

GPGPU would become a remedy for performance in terms of computation speed and its cost. Future development on hardware and software for GPGPU would consolidate its position for great success. In this study, computation performance of the basic operations of matrix and vector using CUDA were verified. And acceleration of burnup calculation was also studied. It was confirmed appropriate treatment of large number of threads and small amount of memory transfer between CPU and GPU is indispensable for good performance. In the nuclide burnup calculation, GPUs performance was accelerated by the factor of 100 compared to that by CPUs. If you put in several hundred dollars for GPUs, there is a possibility that numerical computation is one hundred times faster. Furthermore, by using the GeForce GTX280 which is the latest product of the NVIDIA and has about twice calculation performance than 8800GTX used in this paper, more efficient is expected.

REFERENCES

1. Y. OHOKA and M. TATSUMI, “Study on the Acceleration of the Neutronics Calculation Based on GPGPU,” *Proc. Int. Topical Meeting on Mathematics & Computation and Supercomputing in Nuclear Applications (M&C + SNA 2007)*, Monterey, California, April 15-19, 2007, American Nuclear Society (2007) (CD-ROM).
2. Yasuhiro Kodama, Akio Yamamoto, Yoshihiro, Yamane, et. al."Fast Computation of the Neutron Transport Calculation with a Graphic Processing Unit (GPU)," *Trans. Am. Nucl. Soc.* vol. 99(2008)
3. “NVIDIA CUDA Compute Unified Device Architecture-Programming Guide (Version 2.0)”, http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf (2008)