2009 International Nuclear Atlantic Conference - INAC 2009
Rio de Janeiro,RJ, Brazil, September27 to October 2, 2009
ASSOCIAÇÃO BRASILEIRA DE ENERGIA NUCLEAR - ABEN
ISBN: 978-85-99141-03-8

GPU-Based High Performance Monte Carlo Simulation in Neutron Transport

Adino Heimlich, Antônio C. A. Mol, Cláudio M. N. A. Pereira
Laboratório de Inteligência Artificial Aplicada
Comissão Nacional de Energia Nuclear - IEN/CNEN
R. Hélio de Almeida, 75, 21941-972 - P.O.Box 68550 - Ilha do Fundão - Rio de Janeiro, Brazil
cmnap@ien.gov.br

ABSTRACT

Graphics Processing Units (GPU) are high performance co-processors intended, originally, to improve the use and quality of computer graphics applications. Since researchers and practitioners realized the potential of using GPU for general purpose, their application has been extended to other fields out of computer graphics scope. The main objective of this work is to evaluate the impact of using GPU in neutron transport simulation by Monte Carlo method. To accomplish that, GPU- and CPU-based (single and multi-core) approaches were developed and applied to a simple, but time-consuming problem. Comparisons demonstrated that the GPU-based approach is about 15 times faster than a parallel 8-core CPU-based approach also developed in this work.

1 Introduction

In order to achieve high level of graphical quality with adequate performance, computer graphics applications require very fast processing of large amount of data. To accomplish that, dedicated high performance co-processors, called Graphics Processing Units (GPUs) have been widely used. A GPU is a Single Instruction Multiple Data (SIMD) parallel architecture, in with one instruction can be performed with multiple data in a pipeline, at same time. Since researchers and practitioners realized the potential of using GPU for general purpose, their application has been extended to other fields out of computer graphics scope [17] [14] [21]. Developing GPU-based applications have been considered quite interesting, not only due to the great speedup provided, but also due to the very low cost of GPU, when compared to multiprocessor (multi-core or clusters) architectures. In the nuclear field, however, such technology is poorly explored. Such fact has, motivated the development of this work, which is aimed to show the impact of using GPU in nuclear applications. To accomplish that, a simplified, but computer-expensive neutron transport simulation by Monte Carlo method has been considered. Recently, GPU-based approaches for Monte Carlo calculations has been successfully explored in other fields of application [20] [8], demonstrating great speedup in processing time. In order to provide a comparative analysis, solutions have been implemented for both GPU and CPU. Hence, sequential and parallel (multiprocessor) CPU-based approaches have also been developed in this work. The reminder of this paper is organized as follows: Section 2 gives an overview of the GPU architecture and programming concepts. Section 3 presents the proposed problem. In sections 4 and 5 the GPU and CPU-based algorithm implementation are described. Comparative experiments and results are discussed in section 6 and, finally, section 7 provide some concluding remarks.

2 GPU Architecture

The GPU used in this work was the GeForce GTX-280, the second generation of the CUDA enabled NVIDIA GPUs. GTX-280 architecture is based on Scalable Processor Array (SPA) framework. The SPA architecture, in GTX-280, consists of 10 Thread Processing Clusters (TPCs). Each TPC comprises 3 Streaming Multiprocessors (SMs), and each SM contains 8 Streaming Processors (SPs), or thread processors; each TPC which comprises 8 Arithmetic Logic Unit (ALUs), one 64 bits Floating Point Unit (FPU) and a 24 Kbyte shared memory for communication inter SMs. In summary, 240 streaming processors are available. In order to support a possible divergence in each thread, situations caused by conditional, WHILE IF, etc; within the warp some threads may be inactive during the execution of instructions. Thus different branches in a program are serialized with respect to their threads are active or not, unless it is clear that all threads in warp are in sync. Figure 1 shows a simplified GTX-280 GPU graphics processing architecture.

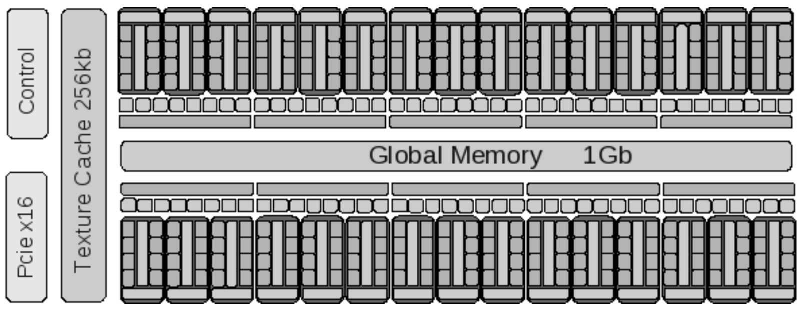

Figure 1: Simplified GTX-280 GPU graphics processing architecture.

The GTX-280 frame buffer memory interface is 512 bits wide, composed of eight 64 bit GDDR3 memory controllers, the GDDR3 memory controller coalesces a much greater variety of memory access patterns, improving the efficiency as well as peak performance. The memory controller is configured with 1GB of memory and the external interface from the GTX-280, to the host system, is a PCI-Express2 x16 slot with a theoretical 8GB/s of bandwidth in each direction conform published by [6] [16]. Each SM can handle up to 1024 threads and is structured in a way to reduce the decision of how many resources are available in order to reduce the overhead in memory access and enable the synchronization between the threads in a few clock machine. This is a structure of blocks of threads called warps, warp each made off 32 threads. These warps are then executed in a handler called Multiple Single Instruction Thread and each thread in warp has its own records, but they are all running the same instruction. As seen in the taxonomy of Flynn, it classifies this topology as a SIMD machine [12]. Figure 2 show a detailed SM structure in TPC.

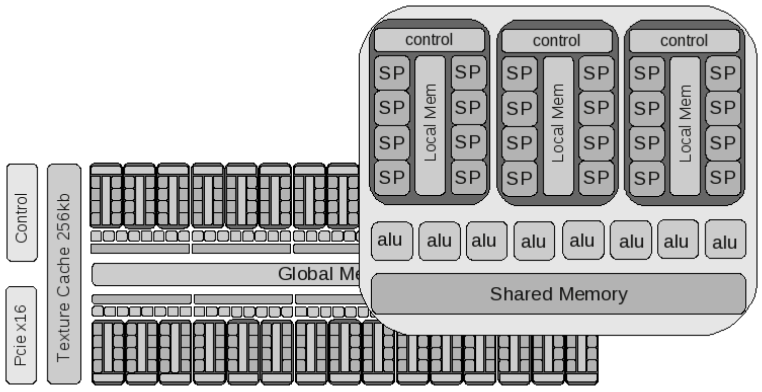

Figure 2: SM zoomed in the GTX-280.

3 CUDA - GPU Programming

CUDA (Compute Unified Device Architecture) is a C-language compiler that is based on the PathScale C compiler, whose origin refers to OPEN64 project [2] [9]. The GPU global block scheduler manages coarse grained parallelism at the thread block level across the whole chip. When a CUDA kernel is started, information for a grid is sent from the host CPU to the GPU. The work distribution unit reads this information and issues the constituent thread blocks to SMs with available capacity. The work distribution unit issues thread blocks in a round-robin fashion to SMs which have sufficient resources to execute it. Some of the factors that are accounted for are the kernel's demand for threads per block, shared memory per block, registers per thread, thread and block state requirements, and the current availability of those resources in each SM. The end goal of the work distributor is to uniformly distribute threads across the SMs to maximize the parallel execution opportunities. Figure 3 show simplified flowchart in CUDA program [19].

3.1 Programming Model

In order to simplify the handling of large numbers of threads in Cuda, this language offers the concept of grids and blocks of threads in which our computational domain, the thousands of threads, which are divided into sets of blocks called grids, in dimensional or two-dimensional way. Each of these blocks may contain up to 512 threads arranged in a three-dimensional grid as shown in figure 4.

The blocks are mapped in the SMs and the Cuda driver offers way to identify both the position of the block in the grid as the position of the thread in the block, this is done through the system variables *blockIdx* and *threadIdx*, which are three-dimensional vectors that point for the index of your thread; through these variables can identify and manipulate the individual threads and this is very useful when dealing with conditions of

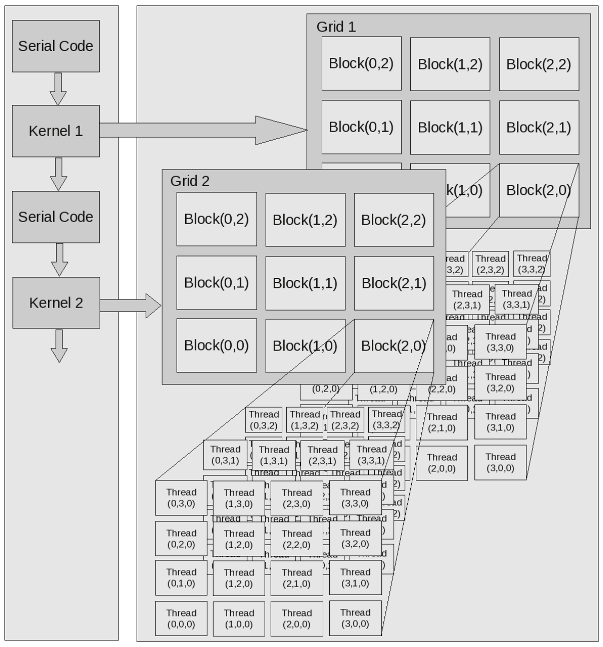

Figure 3: Flowchart on CUDA Programming.

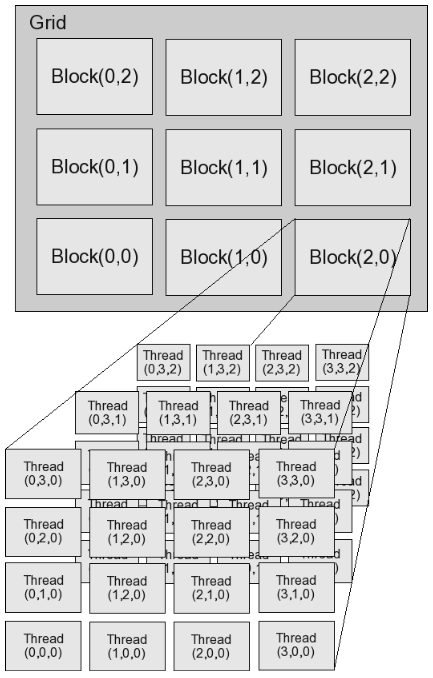

Figure 4: Grids and Threads Blocks on CUDA.

a control problem. The synchronization between the threads of a block is performed by calling the primitive *syncthreads()*, but the synchronization between threads of different blocks is not possible. It is important to note that the programmer has no influence on the result in which an individual thread or threads of a block is processed, the hardware is responsible for this through internal queue managers.

3.2 Memory Model

Each multiprocessor, illustrated as Block (0, 0) and Block (1, 0) if figure 5, contains the following five memory types: one set of local registers per thread, a parallel shared memory that is shared by all the threads in a block and implements the shared memory space. He also a read-only constant memory, it's like a memory cache, that is shared by all the threads . A read-only texture memory cache that is shared by all the processors and speeds up reads from the texture memory space, which is implemented as a read-only region of device memory. The constant memory speeds up reads from the constant memory space, which is implemented as a read-only region of device memory. The global space 1Gb of memory is accessed by more slowly threads, but implements read-write capabilities.

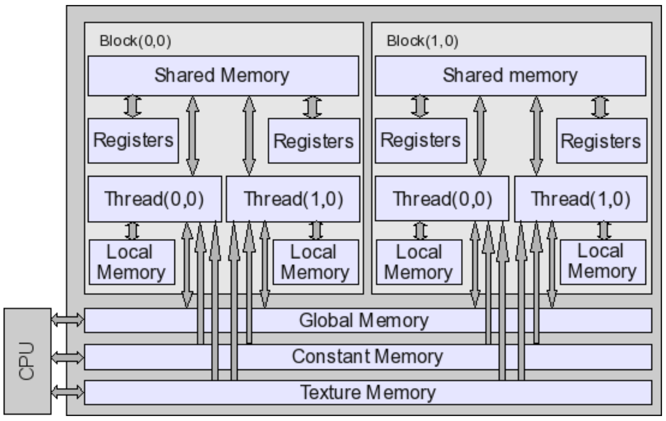

Figure 5: CUDA Memory Model.

4 CPU Models

A several parallel programming models adopted to multi-core processors cold be used as a benchmark problem, like *Bulk Syncronous Parallel* [1], OPENMP [5], Posix Threads [18], MPI [13].

4.1 OPENMP

The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ in several platforms including Unix like in all itś flavors. OpenMP is really a handy and scalable model that gives shared-memory parallel programmers a simple interface for developing parallel all of type of applications. Allied with MPI library [**?**], standart model for distribuited programming, constitute that a powerfull ambience for clusters and other topologies like MIMD [12]. However our approach will be based on posix threads, because this model has most proximity with CUDA programming model.

4.2 CPU Multi-Threading

The POSIX Threads Library is a standard for programming competing processes [3], we call threads, is based on an interface for implementing C/C++ and is executable on Linux or Solaris operating systems. This library provides efficient ways to expand the process running on new competing processes, which can run more efficiently on computer systems with multiple processors and/or processors with multiple cores.

5 The proposed problem

Shielding neutron source represents a essential problem in projects of nuclear reactors and Monte Carlo method [11] is a very useful tool for solving this problem. In contrast to a deterministic method, geometric complexity is a much less significant problem. The accuracy of a Monte Carlo calculation is, of course, limited by the statistical deviation of the quantities to be estimated. Let's considerate the problem represented by neutron transmission over a plate with thickness h; the flux is uniform and orthogonal to a infinite plate surface, like figure 5.

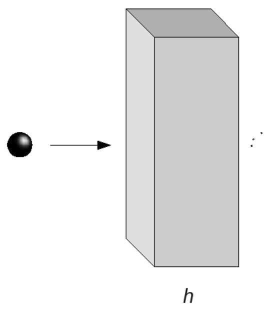

Figure 6: Neutron Shielding in SLAB.

Suppose that mono-energetic neutron beam and a one-dimensional, cartesian, isotopic and steady state in a homogeneous material. The interaction of neutron with materia is characterized in the case under consideration by tree properties: Σ_a , Σ_s and Σ_f denoting the absorption , scattering and fission cross-section respectively [4] [10]. In this case, shielding project, will considerate only that each particle will be suffering two possible actions : be absorbed or be scattering , and not reflected. Therefore the total cross-section is given by,

$$\Sigma = \Sigma_a + \Sigma_s. \tag{1}$$

The physical interpretation of the cross-sections is a follows : In a collision of a neutron with an atom, the probability of absorption is denoted by,

$$P_a = \frac{\Sigma_a}{\Sigma} \tag{2}$$

and the probability of scattering,

$$P_s = \frac{\Sigma_s}{\Sigma} \tag{3}$$

We call *mean free path*, the distance average distance traveled by an neutron without collision, denoted by γ, and each collision is an independent event can see this process describe a Poisson flux [7]. The Poisson flux is a stochastic process that can be defined in terms of occurrences of events. It expresses the probability of a number of events occurring in a given time period, if they occur with a known average rate and where each event is independent of time elapsed since the last event. The random independent variable γ can assume any positive number and is given by ,

$$\gamma = \frac{1}{\Sigma} \tag{4}$$

with probability density given by,

$$P(x) = \Sigma e^{-x\Sigma} \quad ,0 \leq x \leq \infty \tag{5}$$

Let's calculate the expected value of a random variable with a probability density given by 5,

$$F(x) = \int_0^\infty P(x)dx = \int_0^\infty \Sigma e^{-x\Sigma}dx = \varepsilon \quad ,0 \leq x \leq \infty \tag{6}$$

This density is also called the exponential distribution and the mathematical expectation of a random variable γ.

$$\int_0^\sigma \Sigma e^{-x\Sigma}dx = \varepsilon \tag{7}$$

Computing the integral on the left we get the relation

$$1 - e^{-\Sigma\sigma} = \varepsilon \tag{8}$$

in turn, we get

$$\gamma = -\frac{1}{\Sigma}\ln(1-\varepsilon) \tag{9}$$

But the variable (1 - ε) has exactly the same distribution as ε, and so, we can use the equation

$$\gamma = -\frac{1}{\Sigma}\ln\varepsilon \tag{10}$$

the random variable is estimated through a finite number of random numbers or stories, and the previous equation is computed with the sum of each story [15],

$$\bar{x} = \sum_{i=1}^{n}\gamma_i \tag{11}$$

where, n is the number of histories.

6 Algorithm

The implementation of a sequential algorithm in CPU,

```
NH {Number of histories}
WIDTH {Thickness o Material}
N {Number of Neutron out}
$\gamma$ {Random variable}
for $i$ = 1 to NH do
   $dx = 0$ {Incremental neutron random walks}
   while new is false do
      $U = Random()$
      $dx = -MeanFreePath(\ln(U))$ {Calculate neutron random walks}
      $x \leftarrow x + dx$ {Look to equation 11}
      if $x > WIDTH$ then
         $N \leftarrow N + 1$ {Increase neutron out number}
         $new \leftarrow true$
      end if
      $U = PA(Random())$ {Calculate the absortion probability}
      if $PA \geq 1$ then
         $new \leftarrow true$ {If neutron be absorbed}
      end if
   end while
end for
```

The implementation of a parallel algorithm,

```
NH {Number of histories}
NT {Number of threads}
WIDTH {Thickness of Material}
$T[NT]$ {Allocate a vector of threads}
for $j$ = 1 to NT do
   Tj ← start {Start thread $T_j$}
   $N_j$ {Vector of Number of Neutron Out}
   $\gamma_j$ {Vector of Random Variable}
   NHj=NH/NT {Number of histories}
   for $i$ = 1 to NHj do
```

```
        $dx_j = 0$ {Incremental Neutron Random Walks}
        while new is false do
          $U = Random()$
          $dx_j = -MeanFreePath(\ln(U))$ {Calculate Neutron Random Walks}
          $x_j \leftarrow x_j + dx_j$ {Look to Equation 11}
          if $x_j > WIDTH$ then
            $N_j \leftarrow N_j + 1$ {Increase Neutron Out Number}
            $new \leftarrow true$
          end if
          $U = PA(Random())$ {Calculate the Absortion Probability}
          if $PA \geq 1$ then
            $new \leftarrow true$ {If Neutron be Absorbed}
          end if
        end while
      end for
    end for
    for $j = 1 \rightarrow NT$ do
      Tj ← stop {Stop thread $T_j$}
      $N \leftarrow N_j + N$ {Increase Neutron Out Number}
    end for
```

7 Experiments and Results

We use a plate of 10cm aluminum thickness in simulation, with 0.015 cross-section of absorption and 0.84 cross-section of scattering and was measured attenuation of neutron source. The test cover the whole range 2^{20} to 2^{29} neutron, we show that in table 1.

Table 1: GPU and CPU Performance (Seconds)

Neutron Number	GPU	% Neutron	1 Thread	2 Threads	4 Threads	8 Threads
1048576	0,007053	0,859375	0,87600	0,2135	0,4210	0,1090
2097152	0,014318	0,871094	1,74400	0,4350	0,8580	0,2100
4194304	0,029786	0,863281	3,51200	0,8630	1,6690	0,4120
8388608	0,058782	0,869141	7,00400	1,7420	3,3780	0,8210
16777216	0,118388	0,869141	14,28800	3,4315	6,6920	1,6383
33554432	0,237209	0,864502	28,71800	6,8885	13,3970	3,2773
67108864	0,471959	0,865234	57,53800	13,6695	26,8520	6,5495
134217728	0,939593	0,864929	114,77400	27,7620	53,8860	13,0983
268435456	1,879119	0,866486	228,46200	54,5930	107,5670	26,1963
536870912	3,750770	0,867325	467,60800	110,1480	216,02	52,3810

The machine under test is a workstation HP vx8600 with dual 5440 xeons processors with 4 cores per processor and 8 gigabytes of ECC RAM. The GPU under test is

GTX-280 by NVIDIA with 1Gbyte of memory. The test consists of a sequence of 10 neutron stories where each story is double the previous and 10 measures were made in each story and got the mean.

8 Conclusions

The GPU-based implementation is damn faster than that of CPU-based one. For a 1-thread system, we get a speedup factor above 100 times. To attain these speeds we just implements the parallel multi-thread algorithm in CUDA. Several improvements can be performed to accelerate this algorithm, such as a *mersenne twister* random generator and MPI/OPENMP implementation in a multi-device-multi-host topology.

References

[1] R.H. Bisseling and W.F. McColl. Scientific computing on bulk synchronous parallel architectures. *Technology and Foundations: Information Processing*, 94:509–514.

[2] I. Buck. Gpu computing with nvidia cuda. In *International Conference on Computer Graphics and Interactive Techniques*. ACM New York, NY, USA, 2007.

[3] D.R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1997.

[4] L. L. Carter and E. D. Cashwell. *Particle Transport Simulation with the Monte Carlo Method; Prepared for the Division of Military Application, U.S. Energy Research and Development Administration*. U. S. Department of Energy, 1975.

[5] R. Chandra. *Parallel programming in OpenMP*. Morgan Kaufmann, 2000.

[6] J.M. Danskin, J.S. Montrym, J.E. Lindholm, S.E. Molnar, and M. French. Parallel Array Architecture for a Graphics Processor, December 15 2006. US Patent App. 11/611,745.

[7] C.A.B. Dantas. *Probabilidade: um curso introdutório*. Edusp, 1997.

[8] P. Després, J. Rinkel, B.H. Hasegawa, and S. Prevrhal. Stream processors: a new platform for Monte Carlo calculations. In *Journal of Physics: Conference Series*, volume 102, page 012007. Institute of Physics Publishing, 2008.

[9] O. Developers. The Open64 web site.

[10] James J. Duderstadt and William R. Martin. *Transport Theory*. 1979.

[11] S.A. Dupree and SK Fraley. *A Monte Carlo primer: A Practical approach to radiation transport*. Kluwer Academic/Plenum Publishers, 2004.

[12] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960.

[13] W. Gropp, E. Lusk, A. Skjellum, and U. Mpi. *Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, Mass., 1999.

[14] Z. He and K. Harada. Solving point-feature labeling placement problem by parallel Hopfield neural network on GPU graphics card. *Machine Graphics & Vision International Journal*, 15(1):99–120, 2006.

[15] M.H. Kalos and P.A. Whitlock. *Monte carlo methods.* Wiley-VCH, 2008.

[16] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[17] Z. Luo, H. Liu, and X. Wu. Artificial neural network computation on graphic process unit. In *2005 IEEE International Joint Conference on Neural Networks, 2005. IJCNN'05. Proceedings*, volume 1.

[18] Frank Mueller. Implementing posix threads under unix: Description of work in progress. In *In Proceedings of the Second Software Engineering Research Forum*, pages 253–261, 1992.

[19] C. NVIDIA. Programming Guide 2.0, 2008.

[20] C.R. Salama. GPU-Based Monte-Carlo Volume Raycasting. In *Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*, pages 411–414. IEEE Computer Society Washington, DC, USA, 2007.

[21] J. Tölke. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. *Computing and Visualization in Science*, pages 1–11.