EAI Endorsed Transactions
on Internet of Things

Research Article **EAI.EU**

Implementation of Beeman's algorithm to calculate execution time on GPU using CUDA

Youness Rtal[1,*], Abdelkader Hadjoudja[1]

[1]Department of Physics, Laboratory of Electronic Systems, Information Processing, Mechanics and Energy, Faculty of Sciences, Ibn Tofail University, Kenitra, Morocco

Abstract

Graphics processing units (GPUs) are microprocessors designed to the operation of display and manipulation of graphics data. . Currently, these graphics processor are found on all graphics hardware and have become very important instruments for parallel computing. GPUs are practical tools for the development of several fields like decoding and encoding, solving differential equations. Their advantages are increase in performance, faster data processing and reduced power consumption. It is simple to program a GPU with CUDA C to run parallel calculations. But it is necessary to have an understanding of the architectural aspects of the GPU and CUDA C. This paper, we will describe and implement Beeman's algorithm on GPU and CPU using CUDA C to solve the differential equation of charged particles in an electromagnetic field. Our goal is to evaluate the performances of the implementation on GPU and CPU processors and to deduce the efficiency of the use of GPUs.

Keywords: Beeman algorithm, GPU, CPU, Thread, Bloc, Grille, CUDA C/C++.

Received on 25 July 2022, accepted on 17 August 2022, published on 15 December 2022

Copyright © 2022 Youness Rtal *et al.*, licensed to EAI. This is an open access article distributed under the terms of the CC BY-NC-SA 4.0, which permits copying, redistributing, remixing, transformation, and building upon the material in any medium so long as the original work is properly cited.

doi: 10.4108/eetiot.v8i4.2937

*Corresponding author. Email: youness.pc4@gmail.com

1. Introduction

Over the past twenty years, the computational capabilities of graphics processing units (GPUs) for personal computers have evolved considerably. From the acceleration of a few fixed graphics rendering functions, GPUs have gradually incorporated the acceleration of more generic functions, to reach nowadays a level of programmability similar to that of a central processing unit (CPU). Thus, technologies have appeared that allow generic programs to be executed on GPUs. Among these, NVIDIA's Compute Unified Device Architecture (CUDA) technology has emerged as the most successful solution.

The Beeman algorithm that we will implement is a numerical integration method for second-order ordinary differential equations, specifically Newton's equations of motion $\ddot{x} = F(x)$. It was designed to allow for a large number of particles in molecular dynamics simulations. There is a direct or explicit variant and an implicit variant of the method. The direct variant was published by Schofield in 1973 [2] as a personal communication from Beeman. This is commonly referred to as Beeman's method. It is a variant of the Verlet integration method [3]. It produces identical positions, but uses a different formula for the velocities. In 1976, Beeman published [1] a class of multi-step implicit (predictor-corrector) methods, where Beeman's method is the direct variant of the third order method in this class.

This paper will implement the Beeman algorithm on GPU and CPU processors using CUDA C, to solve the differential equation of charged particles in an electromagnetic field. The objective of this study is to compare the performance of the implementation of particle on GPU and CPU and deduce the efficiency of GPU processors for parallel computing.

The upcoming section of this paper are organized as the following: In section 2, we present the CUDA architecture and the hardware used. The section 3 introduces Beeman

algorithm that is implemented to solve the problem. In section 4, we discuss the results of the implementation. The last section concludes our paper.

2. The CUDA architecture and the used hardware

2.1. The CUDA architecture

CUDA is a parallel programming model and software environment developed by NVIDIA [4]. It provides programmers with a set of instructions that enable GPU acceleration for data-parallel computing. The computational performance of many applications can be dramatically increased by using CUDA directly or by linking to GPU accelerated libraries.

CUDA C/C++ is a useful adaptation and extension of programming languages for parallel algorithms. The core idea of CUDA is to execute thousands of threads in parallel to optimize computational results. A CUDA program is a single unified code system that consists of both host and peripheral programs. The host program is compiled only using the standard C compiler. The peripheral program is written using CUDA instructions for the parallel tasks, to program code execution on a GPU with CUDA, one must define the piece of code to be executed (which will be executed in a multitude of threads), and start its execution from the main thread running on the CPU. Thus, the piece of code to be executed on the GPU is defined by the programmer as a function in C that respects certain constraints, and is called a "kernel" in CUDA terminology. The execution of a kernel on a GPU follows a certain syntax, where the programmer specifies the number of threads executing the kernel, under which organization, with a certain amount of shared memory, etc.

CUDA is a technology for performing scientific calculations on GPUs. It is the product of the NVIDIA laboratories. The term CUDA is used to refer to both the hardware and the programming language. The CUDA architecture divides the GPU device into grids and each grid contains a fixed number of thread blocks in a hierarchical structure, as shown in Figure 1 [16]. The configuration of grids and thread blocks helps the programmer to efficiently use all the computational capabilities of the graphics card. [6, 9]

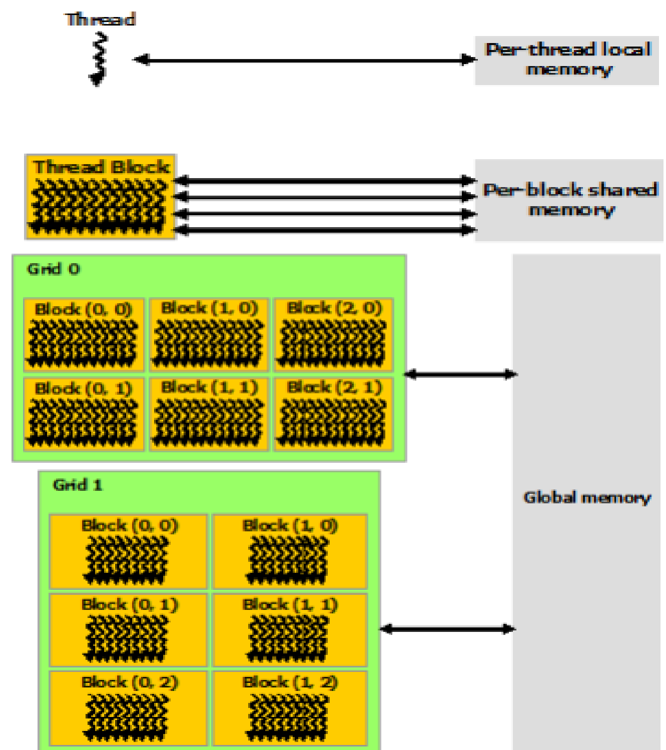

Figure 1. The architecture of the CUDA program

GPU programming with CUDA offers three main types of memory as shown in Figure 1:

Thread-local memory (registers). Memory can also be statically allocated from within a kernel, and according to the CUDA programming model. Such memory will not be global but local memory. Local memory is only visible, and therefore accessible, by the thread allocating it. So all threads executing a kernel will have their own privately allocated local memory.

Block shared memory is a common memory space for a block, organised in 32bit banks. Access to this memory is instantaneous (as for registers), except in the case of an access conflict between two threads in the same block. To allocate a variable in the shared memory space, the variable declaration must be preceded by the keyword __shared__. When calling the kernel, do not forget to specify the size of the shared memory that will be reserved for each block in the kernel launch parameters (<<<...>>>), otherwise an error will occur at runtime. It is possible to synchronise the threads of a block with the __syncthreads() function. This function can be particularly useful when you want to wait until all the threads in the block have finished writing their results to shared memory. The results can then be copied back into global memory, safe in the knowledge that the values in shared memory are correct.

The global memory corresponds to the video memory of the graphics card. It offers much more space than shared memory (from 512MB to several GB, compared to a few

tens of KB), but access is much slower (several hundred cycles compared to almost instantaneous access). Global memory allocation is done using functions such as cudaMalloc(), cudaFree(), which are the CUDA equivalents of traditional C memory management functions. It is also possible to make memory copies between main memory (RAM) and global memory using the cudaMemcpyHostToDevice(), cudaMemcpyDeviceToHost() and cudaMemcpyDeviceToDevice() functions. The choice of method depends on the direction of the copy you want to make. In addition, there are asynchronous variants of these functions using cudaMemcpyAsync(). However, it should be remembered that data copies between RAM and global memory pass through the PCI-Express bus, whose bandwidth is limited (8GB/s in theory for a PCIe 2.0 16x bus, between 2 and 4GB/s in practice).

Note also that other memory areas are available (cache for static variables, texture cache), but are physically based on the memory types presented above. There is relatively little information available on these caches, the best thing to do is to test their performance according to the context of use.

2.2. The used hardware

In this paper, the platform used is a regular computer with an Intel Core 2 Duo E6750 processor and an NVIDIA GeForce 8500 GT graphics card. The full characteristics of both platforms are given in documents [8, 10].

3. Presentation of the problem and the numerical solution method

3.1. Presentation of the problem

In this article, we are going to write a program in CUDA C of the Beeman algorithm, which allows us to study and solve the differential equation of charged particles in an electromagnetic field. To simplify the study, we consider that the charged particles are only subjected to the Lorentz force, the electromagnetic force. The knowledge of the trajectories of the charged particles is useful in several fields such as the techniques of PVD (Physical Vapor Deposition) [5]. These techniques allow the deposition of a thin layer of atoms on a substrate. The problem is then to have a prediction of the uniformity of the deposit. In the case of magnetron sputtering, a particular PVD technique, we exploit the confinement of electrons via a magnetron to obtain a stable plasma. This plasma will then be used to bombard a target from which we wish to release the atoms. These atoms will be deposited, by electronic affinity, in a thin layer on the substrate. Without going into details, the knowledge of the trajectories of these electrons can give us access to the quality of the deposit. In this study, we consider a cube subjected to an electric field and a magnetic induction field, as shown in figure 2. We will place N charged particles in the center of this cube.

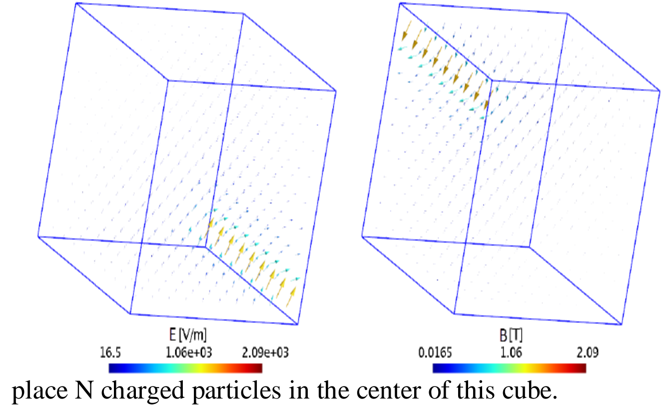

Figure 2. Electric and magnetic induction fields.

We will develop a matrix called Lorentz using the Gmsh software. This one will take an electric field, and a constant magnetic induction field as input, and a set of particles. The matrix will then be loaded to calculate the trajectory of electrons. The different parameters of the matrix are grouped in table 1.

Table 1. The parameters of the Lorentz matrix

Number of iterations	4000
Time step Δt	$10^{-7}\mu s$
Charge of the particle (electron)	$q = -1{,}602176487.10^{-19}kg$
Mass of the particle (electron)	$m = 9{,}10938215.10^{-31}kg$

Each charged particle, when entered into an electromagnetic field is subjected to a force, called the

Lorentz force that has an expression:

$$F = q\left(E(X) + \left(\dot{X} \wedge B(X)\right)\right) \tag{1}$$

By applying the second law of dynamics for a system composed of N charged particles:

$$F = m.\ddot{X} \tag{2}$$

By replacing (1) in (2), we obtain the differential equation of motion of charged particles in an electromagnetic field:

$$\ddot{X} = \frac{q}{m}\left(E(X) + \left(\dot{X} \wedge B(X)\right)\right) \tag{3}$$

With,

X : The position of the particle at a given time.
q : The charge of the particle.
m : The mass of the particle.
$E(X)$: The electric field in X.

$B(X)$: The magnetic induction field in X.
$\dot{X}$: The velocity of the particle.
$\ddot{X}$: The acceleration of the particle.

3.2. The numerical solution method

The resolution of the differential equation (3) describing the motion of each of the N particles of a system requires, because of its complexity, the use of numerical resolution methods. The most commonly used algorithms to solve these differential equations are the Leapfrog algorithm and the Varlet algorithm [6, 3], the Gear method [13, 14, 15] and the Beeman algorithm [11, 12] presented in this article.

3.2.1. Beeman equation and Predictor–corrector

The Beeman algorithm is an algorithm used to solve differential equations. It is based on an explicit predictor-corrector scheme. More precisely, this algorithm first estimates the speed at time t without relying on the position at time $t + \Delta t$. This estimate then allows us to estimate the position of the particle in $t + \Delta t$. Then, via this position estimate, the algorithm can correct the velocity of the particle. Thus, we can compute a better approximation of the position, taking into account this new speed.

The expressions for the position (X), the predicted speed ($\dot{X}_P$) and the corrected speed ($\dot{X}_C$) are grouped below using the Beeman algorithm:

$$\begin{cases} X(t+\Delta t) = X(t) + \dot{X}(t)\Delta t + \frac{2}{3}\ddot{X}(t)\Delta t^2 - \frac{1}{6}\ddot{X}(t-\Delta t)\Delta t^2 + \partial(\Delta t^4) \\ \dot{X}_P(t+\Delta t) = \dot{X}(t) + \frac{3}{2}\ddot{X}(t)\Delta t^2 - \frac{1}{2}\ddot{X}(t-\Delta t)\Delta t + \partial(\Delta t^3) \\ \dot{X}_C(t+\Delta t) = \dot{X}(t) + \frac{1}{3}\ddot{X}(t+\Delta t)\Delta t + \frac{5}{6}\ddot{X}(t)\Delta t - \frac{1}{6}\ddot{X}(t-\Delta t)\Delta t + \partial(\Delta t^3) \end{cases}$$

The Beeman algorithm is used to calculate the execution time of particles charged in an electromagnetic field because the errors produced during the execution of the Beeman algorithm are less than other algorithms.

3.2.2. Steps to follow to write code in CUDA C

A basic CUDA-C program consists mainly of the following parts:

- Install NVIDIA GPU drivers, CUDA SDK and Visual studio as an environment for CUDA v10.2 programming..
- Declaration of CPU and GPU variables.
- Allocation of CPU memory for storing the operand data of the calculation to be executed.
- Filling the allocated memory areas.
- GPU memory allocation for storing the operand data of the calculation to be executed.
- Transfer of the CPU operand data to the GPU.
- Execution of the calculation on the graphics card.
- Transfer of the result from GPU to CPU.
- Release of the memory areas allocated on GPU.
- Release of the memory areas allocated on the CPU.

Creating programs in CUDA C and configuring the environment is an easy task. However, it requires a detailed knowledge of the architecture and writing parallel programs. The most important part of programming in CUDA C is the kernel calls. The distribution of data in the right number of threads is the main element that determines a successful program. [7]

3.2.3. The implemented code in CUDA C

Before the implementation of the Beeman algorithm, it is necessary to know that at the level of parallelization, each CUDA thread will take care of the trajectory of a particle, and we will assume that particles are initially at rest with:

- X represents the position matrix of the particles.
- E represents the matrix containing the electric field interpolated on a regular grid
- B represents the matrix containing the induction field interpolated on a regular grid
- DT represents the size of the time steps
- iE, and iB are vectors containing the values of the electric and induction fields at a given point
- Interp(X[i][j], E, B, iE, iB) interpolates the value of the fields E and B at the point X[i][j] and places the values in iE and iB
- AirPlus, Ai and AiMin are the acceleration vectors in t+Δt, t, and t-Δt.
- VIP, and VIC are the predicted and corrected speed vectors
- k is equal to the ratio q/m of the equation (3).
- cross (E, B) represents the vector product of vectors E and B

3.2.4. The program to implement the Beeman algorithm in CUDA C

```
Void Beeman ()
{
// Copy on Device E, B and first row of X matrices
    CopyOnDevice (E, all)
    CopyOnDevice (B, all)
    CopyOnDevice (X, firstRow)
// Call GPU kernel
     BeemanKernel <<<..., ...>>> ()
// Copy on Host X
     matrix CopyOnHost
     (X, all)
}
__global_void beemanKernel () {
// Thread ID: one thread per particule //
int j = getThreadId ();
// Initialization
// Vi = {0., 0.,
```

```
0.};
AiMin = {0., 0., 0.};
Interp (X [0] [ j], E, B, iE, iB); // Done on GPU
// Beeman //
for (int t = 1; i < M; t ++) {
  // Acceleration at step i
      Ai = k * (iE + cross (Vi, iB)) ;
  // Position update
      X[t][j] = X [t - 1] [j] + Vi * DT + (2/3 * Ai - 1/6 *
AiMin) * DT * DT ;
  // Predicted velocity at step i + 1
      VIP = Vi + (3/2 * Ai - 1/2 * AiMin) * DT ;
// Predicted acceleration
      Interp (X[t][j], E, B, iE, iB);
      AirPlus = K * (iE + cross (VIP, iB));
// Corrected speed
      Vi += (1/3 * AirPlus + 5/6 * Ai - 1/6 * AiMin) * DT ;
// Next step
      AiMin =
      Ai;
}
```

4. The results and discussions

The results of the execution of Beeman's algorithm on GPU and CPU are presented in Figure 3, these best performances are obtained when 20 registers per task, 4352 bytes of shared memory per block and 6 resident blocks per multiprocessor are fixed in the program.

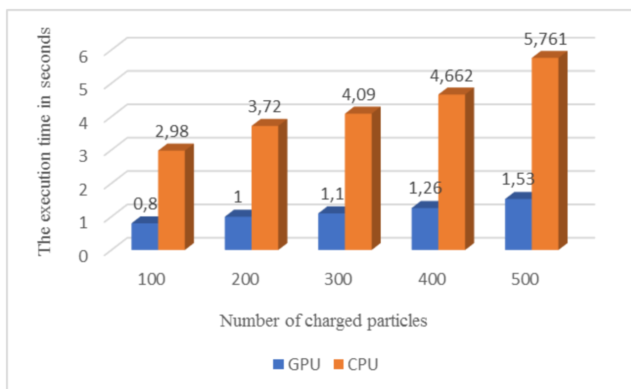

Figure 3. The execution time of the particle implementation charged on the CPU and GPU processors.

Figure 3 shows the evolution of execution time as a function of the number of particles loaded on both CPU, and GPU processors and we notice that:

- When the number of loaded particles increases, the execution time on both GPU and CPU processors increases.
- The execution time of the loaded particles on GPU is faster than CPU.
- For every 100 particles loaded, the execution time on GPU is about 3,7 times quicker than the execution time on CPU.
- This best performance depends on the number of registers per task, the number of bytes of shared memory per block and the number of blocks per multiprocessor.

The implementation results can be explained by the fact that CPUs process data sequentially (task by task), as opposed to the parallel processing of GPUs (several tasks simultaneously). This implies the efficiency of using GPU processors for parallel computing..

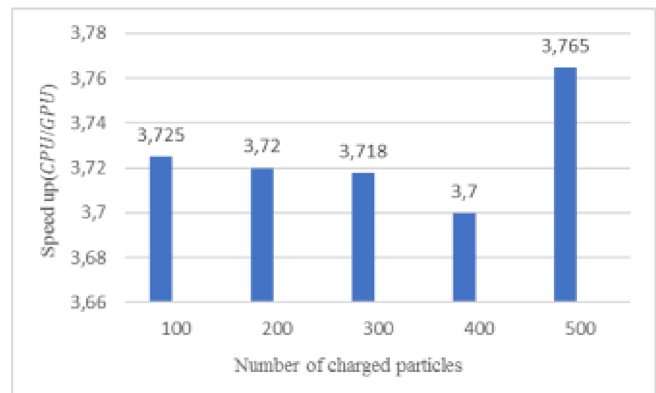

Figure 4. Evolution of Speed up as a function of the number of charged particles

In parallel computing, Speed Up refers to how much faster a parallel algorithm is than a corresponding sequential algorithm. In our case, Speed up = execution time on CPU / execution time on GPU. Figure 4 shows that the Speed Up of this implementation is varied from 3,725 to 3,765. These mainly on the operation of GPU and CPU, and the nature of the charged particles and other parameters. The results of this implementation show that GPU computing is more optimal than CPUs in terms of speed; this optimization is the result of a good selection of the block size used according to the number of processors. Values are very close to each other, and for particles from 100 to 400, the speed up, decreases from 3,725 to 3,7 then increases from 500 particles. The value of speed up depends

5. Conclusion

In this paper, we have successfully implemented Beeman’s algorithm using CUDA C to calculate the execution time of particles charged in an electromagnetic field. This implementation is useful in the PVD (Physical Vapor Deposition) technique and we have seen GPU results outperforming CPUs in terms of execution speed,

this shows the efficiency of using GPUs in parallel computing. Despite this implementation, Nvidia still has many challenges to overcome to keep CUDA C/C++ usable for parallel programming tasks on GPUs, the main task being to convince programmers that this is a credible platform as it features significant processing power. They are becoming the preferred choice for programmers who are proficient in CUDA C/C++.

References

[1] Beeman, David. "Some multistep methods for use in molecular dynamics calculations", Journal of Computational Physics, vol. 20, no. 2, pp. 130–139, 1976.

[2] Schofield, P. "Computer simulation studies of the liquid state", Computer Physics Communications, 5 (1): 17–23, 1973.

[3] Levitt, Michael; Meirovitch, Hagai; Huber, R. "Integrating the equations of motion", Journal of Molecular Biology, 168 (3): 617–620, 1983.

[4] NVIDIA. "NVIDIA CUDA Compute Unified Device Architecture Programming Guide," Version 2.0, 2008.

[5] WD Sproul. "Surface and Coatings Technology," 1996, Elsevier.

[6] Manish Arora. “The Architecture and Evolution of CPU-GPU Systems for General Purpose-Computing,“2012.

[7] Yadav K., Mittal A., Ansari M. A., Vishwarup V. “Parallel Implementation of Similarity Measures on GPU Architecture using CUDA,” 2012.

[8] http://ark.intel.com/Product.aspx?id=30784.

[9] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, Amit Bawaskar. “GPGPU PROCESSING IN CUDA

[10] ARCHITECTURE,” Advanced Computing: An International Journal (ACIJ), Vol.3, No.1, January 2012.

[11] http://www.nvidia.com/object/geforce_8500.html

[12] D. Beeman, J Camp. Phys, pp. 20-130, 1976.

[13] D. Beeman, J Camp. Phys, pp. 52- 24, 1983.

[14] C. W. Gear. "Numerical initial value problems in ordinary differential equations", Newyork, Prentice-Hall,1971.

[15] I.A. Mc Cammon and M. Karplus, Nature, pp. 268-765, 1977.

[16] I.A. Mc Cammon and M. Karplus, Proc. Nat. Acad Sei. USA, pp.58, 1979.

[17] NVIDIA. Whitepaper NVIDIA’s next Generation CUDA Compute Architecture. Nvidia Corp, p. 21, 2009.