Streaming the Boris Pusher: a CUDA implementation

Paulo Abreu, Ricardo Fonseca, and Luís O. Silva

Citation: AIP Conference Proceedings **1086**, 328 (2009); doi: 10.1063/1.3080927
View online: http://dx.doi.org/10.1063/1.3080927
View Table of Contents: http://aip.scitation.org/toc/apc/1086/1
Published by the American Institute of Physics

Streaming the Boris Pusher: a CUDA implementation

Paulo Abreu[*,†], Ricardo Fonseca[*,**] and Luís O. Silva[*]

[*]*GoLP, Instituto de Plasmas e Fusão Nuclear, Instituto Superior Técnico, Lisbon, Portugal*
[†]*IADE-U, Lisbon, Portugal*
[**]*ISCTE, Lisbon, Portugal*

Abstract. We demonstrate an implementation of the Boris Particle Pusher Algorithm using CUDA-enabled processors. A simplified PIC code is used as a reference for the pusher and several tests are shown. The deployment time from a developer point of view is also evaluated. Results show that CUDA is a good tool for significant code acceleration with minimal code impact and minimal effort for the developer.

Keywords: Boris pusher, stream processing, CUDA.
PACS: 52.65.Rr

INTRODUCTION

Programmable graphics processors (GPUs) have received attention from the scientific computing community since their introduction on market at the beginning of 2000 [1]. They are a highly interesting alternative to the usual CPUs due to their high computing power (recently near the teraflop per GPU chip), their relative low cost and power consumption, and their capability for massive data parallelism (several thousand simultaneous threads are available on a single chip of a recent GPU).

However, despite their computational power, general programs for the GPU (GPGPU) suffered a serious implementation difficulty: the algorithm had to be mapped to Computer Graphics concepts in order to use graphics API and the program had to be structured in terms of the graphics pipeline. Several approaches were developed (eg., BrookGPU [2] and Sh [3]), and recently the two major GPU vendors have released their own GPGPU programming system: AMD announced CTM (*Close To The Metal*) and Brook support for higher abstraction; NVIDIA released CUDA (Compute Unified Device Architecture), which offers a high level interface with a C-like syntax. These systems allow for developers to build GPGPU applications without using CG concepts but still using the high performance of the GPU. A usual CPU application can thus more easily tap on the GPU performance without requiring CG knowledge from the programmer.

On this article we demonstrate the implementation of a simplified Boris Particle Pusher algorithm on a CUDA environment (GPU and API). Our purpose was not only to measure what performance gains could be achieved (if any), but also to estimate the amount of effort that was involved in porting an existing code base to this new environment.

CP1086, *Advanced Accelerator Concepts: 13th Workshop*, edited by C. B. Schroeder, W. Leemans, and E. Esarey

CUDA OVERVIEW

CUDA [4] is both a hardware and software architecture for creating GPGPU programs. At the hardware level, it is available for some NVIDIA's GeForce series (8000 and better), the Tesla systems and some Quadro equipment. At the software level, it has a software stack composed by the hardware driver, the C-like API and its runtime, and several higher-level mathematical libraries (CUFFT, CUBLAS).

With CUDA, the GPU is viewed as a compute *device* capable of executing a very high number of threads in parallel. Hence, it operates as a coprocessor to the main CPU, or *host*: portion of an application that is executed many times, but independently on different data, can be isolated into a function, called a *kernel*, that is executed on the device as many different threads.

The batch of threads that executes a computational kernel is organized as a *grid of thread blocks*. Each *thread block* is a batch of threads that can cooperate together by efficiently sharing data through some fast shared memory and synchronizing their execution to coordinate memory accesses. Each thread in a block is identified by its *thread ID*, and each block in a grid is identified by its *block ID*. From the programmer's point of view, all threads in a block can cooperate (sharing memory and synchronizing) as if they are running concurrently; blocks in a grid cannot cooperate. This thread granularity allows for kernels to run efficiently on various devices with different parallel capabilities: a device with few parallel capabilities may run all the blocks of a grid sequentially, while a device with a lot of parallel capabilities may run all the blocks in parallel; usually, it is a combination of both. A *warp* is a batch of threads in a block that are *guaranteed* to run concurrently; although the *warp size* is hardware dependent, all the hardware we had available has a warp size of 32.

Each thread that executes on the device has only access to the device's RAM and on-chip memory (no access to host memory). The device's memory is divided in the following memory spaces:

Registers and local memory: This memory space is available to each thread. It is limited in size (32 registers and 8KB local memory on the GeForce 8 series) but very fast (1 cycle access time).

Shared memory: Available to all threads in a block. Fast access (4 cycles access time) but still limited (16KB in size on the GeForce 8 series).

Global memory: Available to all blocks in a grid and also available from the host. It is on the device's RAM, so it is usually large (from several hundreds of MB up to several GB), but access is slow (over 400 cycles of latency).

Constant and texture memory: Like global memory, they are available to all blocks in a grid and from the host. They are optimized for different memory usages. Since they were not used on this article, we will not explain their uses further.

It is important to compute over data in local or shared memory instead of global memory, because of the high latency of the latter.

The general process of launching a computational kernel with CUDA is thus:

1. Transfer data from host memory to device global memory.

2. Determine the block and grid size.
3. Launch the kernel that does the following:
   (a) Copy the relevant set of data from global memory to shared or local memory.
   (b) Compute on that data.
   (c) Transfer the results to global memory.
4. After returning to the host, the results are copied from the device's global memory to the host memory.

THE BORIS PARTICLE PUSHER ALGORITHM

Numerical simulations in plasma physics [5, 6] often use an algorithm known as the Boris pusher [7] to advance charged particles in the presence of an electromagnetic (EM) field. It is a second-order time centered numerical technique to solve the equations of motion

$$m\frac{d\mathbf{v}}{dt} = q(\mathbf{E} + \mathbf{v} \times \mathbf{B}), \qquad \frac{d\mathbf{r}}{dt} = \mathbf{v}$$

and it has been successful applied in many simulation algorithms [8]. In particular, it has been widely adapted for particle-in-cell (PIC) codes.

The computational steps of the Boris pusher are as: 1) Starting at a given time t, with a time step Δt, the values of the velocity are given for $t - \frac{\Delta t}{2}$: $\mathbf{v}_{t-\frac{\Delta t}{2}}$; however, the position of the particles $\mathbf{r}_t$ and the electromagnetic fields $\mathbf{E}_t$ and $\mathbf{B}_t$ are time centered at t; 2) calculate $\mathbf{v}_{t+\frac{\Delta t}{2}}$ with:

$$\mathbf{v}^- = \mathbf{v}_{t-\frac{\Delta t}{2}} + \frac{q}{m}\frac{\Delta t}{2}\mathbf{E}_t, \qquad \mathbf{v}' = \mathbf{v}^- + \frac{q}{m}\frac{\Delta t}{2}(\mathbf{v}^- \times \mathbf{B}_t),$$
$$\mathbf{v}^+ = \mathbf{v}^- + \frac{2\frac{q}{m}\frac{\Delta t}{2}}{1 + \left(\frac{q}{m}\frac{\Delta t}{2}B_t\right)^2}(\mathbf{v}' \times \mathbf{B}_t), \qquad \mathbf{v}_{t+\frac{\Delta t}{2}} = \mathbf{v}^+ + \frac{q}{m}\frac{\Delta t}{2}\mathbf{E}_t;$$

3) calculate $\mathbf{r}_{t+\Delta t}$ with: $\mathbf{r}_{t+\Delta t} = \mathbf{r}_t + \Delta t\, \mathbf{v}_{t+\frac{\Delta t}{2}}$.

When used in a PIC code, $\mathbf{E}$ and $\mathbf{B}$ are defined at the each cell's vertices, and they have to be interpolated at $\mathbf{r}_t$.

A simplified PIC algorithm, where the self-consistent evolution of the fields is neglected, can be resumed as follows: 1) define the initial conditions: $t = t_0$, $\mathbf{r}_{t_0}$, $\mathbf{v}_{t_0-\frac{\Delta t}{2}}$; 2) calculate $\mathbf{B}_t$ and $\mathbf{E}_t$; 3) for each particle: i) interpolate $\mathbf{B}_t$ and $\mathbf{E}_t$ at $\mathbf{r}_t$; ii) push the particles with the Boris pusher, getting the new positions; 4) advance the time by Δt and go to 2.

IMPLEMENTATION

We have implemented the simplified PIC algorithm in CUDA. We decided to use a 3D simulation space, since this is usually more challenging and was likely to show possible shortcomings of this approach.

According to the CUDA summary previously described, each thread is responsible for pushing one particle. We make sure that the number of particles is divisible by the warp size (in our case, 32) and, for simplification, we also attribute an integer number of particles per space cell. Thus we can set the block size (number of threads per block) as the number of cells in the longer dimension of the simulation space (for instance, along the z axis), guaranteeing at the same time that the block size is a multiple of the warp size. This assures that the GPU has always enough concurrent threads in a block to assure maximal occupancy. The number of blocks per grid of threads is the total number of particles divided by the number of cells in the z axis. The following code illustrates the setting of the block and grid size:

```
// How many threads per block:
ThreadsPerBlock = 64; // Twice the warp size but can be hand tuned
kBlockDim = make_uint3(ThreadsPerBlock, 1, 1);
int kGridSize = CellsNum.x * CellsNum.y * CellsNum.z;
// How many blocks in a grid (is multiple of ThreadsPerBlock?):
if (kGridSize % ThreadsPerBlock)
  printf("Number of cells not optimal.\n");
// Ppc = particles per cell:
kGridDim = make_uint3(kGridSize/ThreadsPerBlock, Ppc, 1);
```

Before launching the kernel, the position **r** and velocity **v** of all the particles, and the **E** and **B** fields have to be initialized and transferred to the device's global memory. This last step is illustrated in the following code for the positions:

```
// Positions: Rdev in GPU mem; Rhost in CPU mem (N = # of particles)
cudaMalloc( (void**)&Rdev, N*sizeof(float3) );
cudaMemcpy( (void *)Rdev, Rhost, N*sizeof(float3),
            cudaMemcpyHostToDevice );
// Same for velocities, E & B ...
```

The previous steps were done on the host (CPU). After launching the kernel, each thread pushes one particle. Since it only needs the position **r** and velocity **v** of that particle, those values are stored in the threads' local memory for immediate access. On the other hand, the **E** and **B** fields are needed for interpolation for all the threads, so they are kept in the global memory of the device. The following code illustrates the main kernel implementation:

```
// Find particle index from thread & block index:
int thread_id = threadIdx.x + threadIdx.y * blockDim.x;
int block_id = blockIdx.x + blockIdx.y * gridDim.x;
int idx = block_id * blockDim.x * blockDim.y + thread_id;

float3 r, v; // This particle pos and vel.
// Copy from shared to local memory for speed:
r = R[idx]; v = V[idx];

float3 Bi, Ei; // Interpolated fields
InterpolateFields_d(r, E, B, PosMin, PosMax,
                    CellsDim, CellsNum, &Ei, &Bi);
CalculateVelocity_d(v, Ei, Bi, wt);
CalculatePosition_d(r, v, PosMin, PosMax, step);
// Store the results back in shared memory (still device):
R[idx] = r; V[idx] = v;
```

After the kernel runs, all particles have been pushed one time step. We than copy the result back from the device's global memory to the CPU memory. Since the computational kernel runs on the GPU, we decided to display the particles immediately after each push using OpenGL commands. This way, we only need to copy back the positions. We could even avoid copying the positions by using OpenGL's Vertex Buffer Objects (VBO). A short comparison of using these two approaches will be done in the next section.

RESULTS AND DISCUSSION

We have tested our implementation on several CUDA enabled GPUs: a GeForce 8800GT, a GeForce 9800GX2 and a Tesla C870. We have also compared that with a similar implementation on the CPU. The CPUs we had available were an AMD Athlon 3200+@2.0 GHz, an Intel Core 2 Duo E6750@2.66 GHz and an Intel Core 2 Duo T9300@2.5 GHz. Table 1 summarizes the results for a simulation space of $32 \times 32 \times 128$ cells and one particle per cell (thus 132 Kparticles). The top half refers to the CPUs performance and the bottom to the GPUs.

TABLE 1. Boris Pusher performance for $32 \times 32 \times 128$ cells with one particle per cell.

Processor	**FPS**	**Push** (ms)	**Gain**
Athlon 3200+@2.0 GHz	7.8	126	
Intel E6750@2.66 GHz	16	61	
Intel T9300@2.5 GHz	27	31	
Tesla C870	72	12	2.6×
GeForce 8800GT	54	16	1.9×
GeForce 9800GX2	51	17	1.8×

The CPU code was compiled with `GCC` using the `-O3` optimization option while the CUDA code was compiled using NVIDIA's `nvcc`. The Intel performance was measured only on a single core. The GPU times include memory transfers from the host to the device and back. The FPS column shows the frames per second achieved while displaying the particles. Using OpenGL's VBO showed no significant improvement, since display times were already much shorter than push times (about $\frac{1}{20}$th).

Speed gains were significant, as the left column of the GPUs show, when compared to the fastest processor we had available (the Intel T9300). Specially relevant is the almost $2\times$ gain with the 8800GT: this GPU is already outdated and nonetheless is able to have twice the performance of a recent CPU, including the memory transfers penalty, for a price of $\approx$ 140 Euros.

We also evaluated the time to implement our simplified CUDA PIC code. Installation of the CUDA environment (drivers, API, compiler) was very straightforward and took us less than a day's work (including tweaking some Linux kernel parameters for one of the machines). Learning the CUDA architecture and API was also less than a day. Translating the C++ code to CUDA took us also one day. Debugging and minimal optimizations (e.g. using shared memory instead of global memory) took us two days, so that in less than a week for production.

FUTURE WORK

After this first approach to CUDA, several paths are available for improvement, namely optimizing the CUDA usage and improving the algorithm. To optimize CUDA usage, several options are available. Memory access could be made coalescent, so that accessing device memory from different threads of the same block can be made concurrently. For arrays of 2 floats (eg, in 2D simulations) this happens automatically, but not so for arrays of 3 floats as in a 3D simulation. One solution would be to use 4 floats per 3D variable: memory usage would increase, but device memory access time would shorten considerably. Another optimization would be to overlap host-device memory transfers with kernel execution (*streaming* in CUDA). This decreases kernel launch time, which can be relevant when several kernel calls are made in quickly succession. Finally, for 2D simulations, interpolation and border conditions are readily available on the hardware, thus automatically optimizing that part of the kernel code.

The algorithm can also be improved. First of all, it can be more complete by implementing a particle deposit and field update part thus actually providing a full PIC code on a GPU. Device memory limitations can be overcomed by implementing an MPI version. Another approach is to use CUDA as a module to be called from existing simulation codes, so that only adequate parts of it are translated to CUDA kernels. One important current limitation of CUDA is double precision, with a performance about $9\times$ slower than single precison. However, it is possible to adapt and to optimize the PIC algorithm so that the effects of lack of precision are minimized.

ACKNOWLEDGMENTS

This work was partially supported by FCT, Portugal (grants SFRH/BD/17870/2004 and POCI/66823/2006), and by the NVIDIA Professor Partnership Program. The authors would like to thank Luís Gargaté for his help on some fine details of the Boris Pusher algorithm.

REFERENCES

1. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, *Computer Graphics Forum* **26**, 80–113 (2007).
2. I. Buck, T. Foley, D. Horn, J. Sugerman, P. Hanrahan, M. Houston, and K. Fatahalian, "Brook for GPUs: Stream Computing on Graphics Hardware," in *Proceedings of the ACM SIGGRAPH 2004*, 2004.
3. M. Mccool, S. Du Toit, T. Popa, B. Chan, and K. Moule, "Shader algebra," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, ACM Press, New York, NY, USA, 2004, pp. 787–795.
4. J. Nickolls, I. Buck, M. Garland, and K. Skadron, *Queue* **6**, 40–53 (2008), ISSN 1542-7730.
5. J. M. Dawon, *Rev. Mod. Phys.* **55**, 403–447 (1983).
6. C. K. Birdsall, and A. B. Langdon, *Plasma Physics Via Computer Simulation*, Adam Hilger, UK, 1991.
7. J. P. Boris, "Relativistic Plasma Simulation—Optimization of a Hybrid Code," in *Proc. Fourth Conference on the Numerical Simulation of Plasmas*, 1970, pp. 3–67.
8. R. W. Hockney, and J. W. Eastwood, *Computer Simulation Using Particles*, Institute of Physics Publishing, Bristol and Philadelphia, 1988.