A Distributed Memory GPU Implementation of the Boris Particle Pusher Algorithm

P. Abreu$^1$, J.M. Pereira$^2$ and L.O. Silva$^1$

$^1$GoLP/Centro de Física dos Plasmas, Instituto Superior Técnico, Lisboa, Portugal
$^2$IST/INESC-ID, Lisboa, Portugal

Abstract
The Boris pusher is a numerical algorithm to advance charged particles in an electromagnetic field. It is widely used in numerical simulations in Plasma Physics. This short paper explains the implementation of the Boris pusher algorithm on stream processors, in particular on a modern Graphics Processor Unit (GPU) with programmable shading capabilities, and explores the parallelization of the code on several GPUs.


1. Introduction

Numerical simulations in plasma physics [Daw83, BL91] often use an algorithm known as the Boris pusher [Bor70] to advance charge particles in the presence of an electromagnetic (EM) field. It is a second-order time centered numerical technique to solve the equations of motion

\[
\frac{m}{dt} \frac{dv}{dt} = q(E + v \times B)
\]

\[
\frac{dr}{dt} = v
\]

and it has been successful applied in many simulation algorithms [HE88]. In particular, it has been widely adapted for particle-in-cell (PIC) codes. The PIC codes belong to the general class of particle-mesh algorithms, widely used not only in plasma physics but also in fluid dynamics [HSW65]. A detailed description of a state-of-the-art PIC code can be found here [FST*02].

In a PIC code, physical quantities that are related to space (like the EM fields or the current) are stored as vertices of space cells, while quantities that are directly related with particles (like the charge or the momentum) are stored with the particles. When a cell quantity is needed at a particle position (e.g., when the EM fields need to be determined in order to update a particle velocity or position), that value is interpolated at the particle position. To update the cell quantities, the particles’ effect on that quantity has to be calculated. In PIC codes, the sources of the EM fields are the currents and the charges of the particles. The fields are determined from Maxwell’s equations after depositing the current associated with each particle on the grid cells.

The particle push is one of the most time consuming steps in PIC codes. Since it uses vectorial quantities and it often needs to do vectorial operations, the Boris pusher is a very good candidate for high code acceleration, by adapting the algorithm to use vector and stream instructions (SIMD). Although most processors have vector instructions available, like AltiVec on the PowerPC and MMX/SSE/SSE2/SSE3 on the Intel/AMD, they are usually working at the top of their capabilities during a numerical simulation. On the other hand, recent Graphics Processor Units (GPUs) have SIMD capabilities and have shown to provide good performance not only on streaming applications by performing the same operation over large collections of data as well as on applications that have sufficient parallelism and computational intensity to hide memory latency [BFH+04]. GPUs like the ATI X1900 series [ATI06] and the NVIDIA GeForce 7800 series [NVI05], feature both programmable vertex and fragment processors and provide support for floating point operations, making them available targets for streaming computation.

Taking advantage of these characteristics and the fact that GPUs are usually idle during numeric computations, in this short paper we describe the programming of a simplified PIC
code in the GPU’s fragment processor as well as we discuss some issues of a distributed implementation over a cluster of GPUs.

2. The Boris pusher in a simplified PIC code

The computational steps of the Boris pusher are as follows:

1. Starting at a given time \( t \), with a time step \( \Delta t \).
2. the values of the velocity are given for \( t - \frac{\Delta t}{2} \), \( v_{t-\frac{\Delta t}{2}} \).
3. However, the position of the particles \( r_t \) and the electromagnetic fields \( E_t \) and \( B_t \) are time centered at \( t \).
4. Calculation of \( v_{t+\frac{\Delta t}{2}} \) with:

\[
\begin{align*}
v^- & = v_{t-\frac{\Delta t}{2}} + \frac{q}{m} \frac{\Delta t}{2} E_t \\
v' & = v^- + \frac{q}{m} \frac{\Delta t}{2} (v^- \times B_t) \\
v^+ & = v^- + \frac{2 \Delta t}{m} \left( \frac{1}{1 + (\frac{\Delta t}{m} E_t B_t \Delta t \times B_t)} \right) (v^- \times B_t) \\
v_{t+\frac{\Delta t}{2}} & = v^+ + \frac{q}{m} \frac{\Delta t}{2} E_t
\end{align*}
\]

5. Calculation of \( r_{t+\Delta t} \) with:

\[
r_{t+\Delta t} = r_t + \Delta t v_{t+\frac{\Delta t}{2}}.
\]

When used in a PIC code, \( E \) and \( B \) are defined at the each cell’s vertices, and they have to be interpolated at \( r_t \).

A simplified PIC code, where the self-consistent evolution of the fields is neglected, can be resumed as follows:

1. Define the initial conditions:

\[
t = t_0, \quad r_0, \quad v_{t_0-\frac{\Delta t}{2}}.
\]

2. Calculate \( B_0 \) and \( E_0 \).
3. For each particle:
   a. Interpolate \( B_t \) and \( E_t \) at \( r_t \).
   b. Push the particles, getting the new positions, with the Boris pusher.
4. Advance the time by \( \Delta t \) and go to 2.

3. Streaming the Boris pusher

Modern GPU have been shown useful for many scientific computations [LLM04, GGHM05]. Recent work has shown the feasibility of using the stream capabilities of modern GPU for particle tracing [KKKW05], mainly due to the following reasons:

- Programmable shaders offer a programmable environment with highly accelerated vectorial operations.
- It is possible to allocate GPU memory that can be used as a texture map or a vertex array alternatively. This gives us the ability to run the whole algorithm with minimal data transfer between CPU and GPU memory.

- GPU support for float precision is improving. Recent processors can use full 32-bit floats in the entire pipeline, and 16-bit floats (OpenEXR format) for textures.

These capabilities of modern GPUs make them good hardware candidates for implementing the Boris pusher and the simplified PIC code explained in the previous section.

The implementation starts by allocating and initializing the textures that will hold the values of the vectorial quantities. The highest precision available for textures is 16-bit floats in the OpenEXR format.

For \( N \) particles, \( r \) and \( v \) are stored in 1D RGB textures of length \( N \) (or of the next power of two, since non power of two textures can still be penalized in performance). Two textures for each one of these quantities are needed, one that stores \( r_t \) and \( v_{t-\frac{\Delta t}{2}} \), and another for \( r_{t+\Delta t} \) and \( v_{t+\frac{\Delta t}{2}} \). We call this textures \( T_{r0}, T_{r1}, T_{v0}, T_{v1} \). Two 3D RGB textures, \( T_E \) and \( T_B \), are also needed, one for \( E \) and the other for \( B \). They have the same dimension has the number of cells of the simulation space (again, some power of two adjustment might be needed). Figure 1 illustrates how these textures are used in the algorithm.

![Figure 1: The use of textures in the GPU implementation of the Boris pusher.](image-url)
Scalar quantities, like \( \Delta t \), \( m \), and \( q \), are passed as uniform shader parameters.

For each particle, \( \mathbf{E} \) and \( \mathbf{B} \) are interpolated at the particle position, after it is read from \( T_{k0} \). The new positions are calculated using the Boris pusher: \( \mathbf{v}_{k+1} = \mathbf{v}_k + \frac{\Delta t}{2} \mathbf{E} \) and \( \mathbf{r}_{k+1} = \mathbf{r}_k + \Delta t \mathbf{v}_k + \frac{\Delta t^2}{2} \mathbf{B} \). At this step, \( T_{k1} \) can also be rendered to the screen.

The time is advanced and a new cycle begins. \( T_{k1} \) is calculated and rendered to \( T_{k+1} \), and \( \mathbf{r}_{k+1} \) is calculated and rendered to the \( T_{k+1} \) texture. At this step, \( T_{k+1} \) can also be rendered to the screen.

Boundary conditions have to be implemented. In our simulation, we use a periodic boundary, so that the particles are re-injected when they leave the simulation box.

### 4. Parallelization Issues

With this implementation, we are able to have a real-time display of a simple PIC algorithm with a Boris pusher. However, the memory size of modern GPUs is still an important limitation. A small simulation, with 128\(^3\) cells and 8 particles per cell requires 408 MB just for texture memory, which might not be available even on video cards with 512 MB RAM.

One way to overcome this limitation is to distribute the simulation over several GPUs. Similar approaches have been done with other algorithms [KKKW05, HHH05]. With our simplified PIC code, the particles do not influence the fields. The Boris pusher can be parallelized with very little penalty if we only partition the particles over each node, but still use the full fields. In our previous example, memory requirements are reduced from 408 MB to \( 24 \sqrt{8} P \) MB, where \( P \) is the number of available GPU processors. With just 2 GPUs, this results in 216 MB of texture memory, which fits well in processors with 512 MB video memory.

More interesting is the possibility to run simulations of a reasonable size, like 256\(^3\) cells and 8 particles per cell. This is a 8x increase on texture memory requirements. On a 16 node cluster, 288 MB are needed per node. As long as communication between the nodes is kept to a minimum, this is an acceptable solution.

### 5. Cluster Implementation

An implementation of the techniques described in this short paper have been implemented on commodity hardware that is normally used as part of a Grid node for plasma physics simulation. Each working node (WN) was configured with an AMD Athlon 64 3200+ CPU with 1 GB RAM and a nVidia 6600GT PCIe video card with 128 MB RAM. An user interface node (UI) was also used, which was responsible for initializing the textures with the data and for launching the GPU processes on the working nodes. The LAM MPI implementation was used as the communication interface between the nodes [GLS99].

The application running on the UI detected the WN available and distributed the particles. After receiving the particles, each WN ran the simulation for a certain simulation time or number of steps (set by the UI). During this time, no node communication was needed. The final result (particle position and velocity) was sent from the WN to the UI, where it was stored and displayed.

### 6. Performance

Simulations run on a system with one UI and two WNs showed a very good performance, comparable to the simulation on the CPU. Overhead caused by the transfer of textures between CPU and GPU memory was compensated by the SIMD characteristics of the GPU. Even a full PIC code using the GPU implementation of the Boris pusher showed promising results. An improved system is being built, with more nodes and better video cards (more video memory and improved pipelines), so that more realistic simulations can be run.

### 7. Limitations and Future Work

The biggest limitation in this GPU implementation of the Boris pusher is the lack of support for double precision floats in the GPU. To study some very detailed plasma behavior, half or single precision is not enough. However, even in these cases a GPU Boris pusher with limited float precision is still helpful as a way to see a real-time evolution of the simulation, that helps to quickly grasp the general behavior of the particles.

The limited video memory available to most GPUs (usually less than 1 GB) is also a constrain on the application of this algorithm on big-scale simulations. Parallelization is a good option for a simplified PIC code, since very little node communication is required.

However, using this algorithm with a full PIC code is possible and an interesting challenge. In full PIC codes, the moving charges creates a current \( \mathbf{J} \) which influences the EM fields. Two steps have to be introduced after the Boris pusher in the simplified PIC algorithm detailed in section 2:

- calculate the new current \( \mathbf{J} \) from the position of the charges and from the momenta, called current depositing;
- solve the field advance equations for the \( \mathbf{E} \) and \( \mathbf{B} \) fields, so that the influence of \( \mathbf{J} \) is taken into account.

This not only increases memory requirements (another texture has to be available to read from and to render to), but also forces node communication at each time step, since in order to deposit the current, particles stored in different processors might be needed.

The usual approach is to pass particles from one processor to another, as they leave a cell stored in one processor and enter another cell, stored in a different processor. This would
mean creating and deleting particles from textures, which is very computational expensive for the GPU. However, if video memory allows to store textures for the full $E$, $B$ and $J$, then particles might not have to change textures from one GPU to another. It could be enough to keep track of which particle from which node is influencing $J$ for depositing. A possible way to implement this could be by using a RGBA texture and storing this information in the alpha channel. This would further increase texture memory requirements, but would minimize both the transfer of data between the GPU and the CPU, and the node communication per cycle.

8. Acknowledgments

This work was partially supported by FCT, Portugal, SFRH/BD/17870/2004. The authors would like to thank Ricardo Fonseca, Luís Gargaté and Michael Marti from GoLP, IST, for their help on some fine details of the Boris Pusher algorithm.

References


