PIC Codes in New Processors: A Full Relativistic PIC Code in CUDA-Enabled Hardware With Direct Visualization

Paulo Abreu, Ricardo A. Fonseca, João M. Pereira, and Luís O. Silva

***Abstract*—Kinetic plasma simulations using an electromagnetic particle-in-cell (PIC) algorithm have become the tool of choice for numerical modeling of several astrophysical and laboratory scenarios, ranging from astrophysical shocks and plasma shell collisions, to high-intensity laser–plasma interactions, with applications to fast ignition and particle acceleration. However, fully relativistic kinetic codes are computationally intensive, and new computing paradigms are required for one-to-one direct modeling of these scenarios. In this paper, we look at the use of modern graphics processing units for PIC algorithm calculations, discussing the implementation of a fully relativistic PIC code using NVIDIA's Compute Unified Device Architecture, also allowing one for simultaneous visualization of simulation results with negligible impact on performance. Details on the algorithm implementation are given, focusing on grid-particle interpolation and current deposition and also on the direct visualization routines. Finally, we present results from a test simulation of an electron/positron plasma shell collision, focusing on code validation and performance evaluation.**

***Index Terms*—Parallel algorithms, plasma simulation, visualization.**

I. Introduction

THERE are many plasma physics scenarios where fully relativistic particle-in-cell (PIC) codes play a key role in providing essential understanding of the underlying physical processes involved. Due to the the problems they try to solve, PIC implementations tend to be used in large-scale simulations that require large computing resources, usually from tens to thousands of computing cores. Furthermore, the last 15 years have witnessed a trend in the high-performance computing (HPC) scientific community to move from highly customized shared-memory systems to cheap distributed-memory systems, often built with commercial off-the-shell computers. However, since the CPU annual increase in speed has almost stopped, this cheap clusters have started to show their limitations, particularly in computational power (high latency), space, and power needs. As the physical problems these computing clusters try to solve increase in complexity—more memory needs (in capacity and bandwidth) and computational power—the scientific community has started looking for other possibilities from where to get their next HPC system.

Programmable graphics processors (GPUs) have received attention from the scientific computing community since their introduction on market in 2000 [1]. They are a highly interesting alternative to the usual CPUs due to their high computing power (recently over a 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). Since 2007, NVIDIA has been offering an architecture (hardware, drivers, application programming interface, and software development kits) for HPC on GPUs called CUDA (Compute Unified Device Architecture). In particular, "C for CUDA" offers a high-level interface to program and deploy high-performance applications on GPus using a a C-like syntax and compiler.

In this paper, we will present an implementation of a full relativistic 2-D PIC code in CUDA. We will discuss the problems that had to be overcome and the solutions we found which, in most cases, are valid for deploying most particle mesh algorithms in CUDA systems. We will also discuss performance issues while implementing our validation test, a 2-D Weibel instability. In addition, we will show how to take advantage of the visualization/rendering capabilities of CUDA GPUs by implementing a direct visualization system in our code that allows one for live display of the results of the simulation directly on the screen while this is running. These results include rendering of the particles trajectories and the display of different diagnostics [electromagnetic (EM) field, current, and charge density]. All these can be easily achieved at interactive frame rates for millions of particles and hundred of thousands of grid cells.

The rest of this paper is organized as follows. In Section II, we present the basic PIC algorithms that were ported to CUDA.

Manuscript received February 5, 2010; revised August 20, 2010; accepted October 12, 2010. Date of publication December 3, 2010; date of current version February 9, 2011. This work was supported in part by Faculdade de Ciências e Tecnologia, Portugal, under Grants SFRH/BD/17870/2004 and GRID/GRI/81800/2006 and in part by the NVIDIA Professor Partnership Program.

P. Abreu is with the Institute for Plasmas and Nuclear Fusion, Technical University of Lisbon, 1169-047 Lisbon, Portugal, and also with the UNIDCOM/IADE, Research Unit in Design and Communication at IADE Creative University, 1200-649 Lisbon, Portugal (e-mail: paulotex@ist.utl.pt).

R. A. Fonseca is with the DCTI/ISCTE-Lisbon University Institute, 1649-026 Lisbon, Portugal, and also with the Institute for Plasmas and Nuclear Fusion, Technical University of Lisbon, 1169-047 Lisbon, Portugal (e-mail: ricardo.fonseca@ist.utl.pt).

J. M. Pereira is with the Instituto de Engenharia de Sistemas e Computadores, 1000-029 Lisbon, Portugal (e-mail: jap@inesc.pt).

L. O. Silva is with the Institute for Plasmas and Nuclear Fusion, Technical University of Lisbon, 1169-047 Lisbon, Portugal (e-mail: luis.silva@ist.utl.pt).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TPS.2010.2090905

0093-3813/$26.00 © 2010 IEEE

In Section III, we discuss the implementation of those algorithms in "C for CUDA," and in Section IV, the direct visualization features and its integration with the simulation code. In Section V, we present the validation of the code by implementing a Weibel instability simulation. Finally, in Section VI, we discuss the results and performance of the code, and in Section VII, we will offer an overview and conclusions of this paper.

II. Basic Algorithms

Directly modeling the interaction between all particles in a plasma is only feasible for a relatively small number of particles, given that this algorithm requires a number of operations going with $O(N^2)$. Even on a petascale system, a state-of-the-art kinetic plasma simulation with a total number of particles of $\sim 10^{10}$ [2] would require $\sim$1 day for a single time step. To overcome this limitation, numerical simulations often resort to the particle-mesh method for calculations. In this method, particles do not interact directly but rather through fields defined on a grid. Field values are interpolated at particle positions to determine the forces acting on each particle, and particles are deposited back on the grid to advance field values. Although the number of operations depends on the number of grid cells, the number of particles, and the interpolation schemes used, it is several orders of magnitude lower than a particle–particle method.

For plasma physics simulations, this algorithm is generally referred to as the PIC algorithm [3], [5]. Simulation space is discretized as a regular grid, and the EM field values are defined on this grid. To advance simulation particles, we use the Lorentz force acting on the particle, calculated by interpolating the field values at the particle positions. The electric current resulting from particle motion is deposited back on the grid and used to advance the values of the EM field.

In our implementation, particles are advanced using a leap frog scheme. Particle positions $\mathbf{x}$ and fields are defined at time step t_i and are used to calculate the Lorentz force acing on the particle

$$\dot{\mathbf{u}} = \frac{q}{m}\left(\mathbf{E} + \frac{1}{c}\frac{\mathbf{u}}{\gamma} \times \mathbf{B}\right) \tag{1}$$

where $\mathbf{u}$ is the generalized velocity $(\mathbf{u} \equiv \gamma\mathbf{v})$; γ is the Lorentz factor; q/m is the charge/mass ratio of the particle; and $\mathbf{E}$ and $\mathbf{B}$ are the EM field interpolated at the particle's position.

The interpolation of the values of $\mathbf{E}$ and $\mathbf{B}$ at the particle can be seen as the interaction between the simulation grid and the particles. For linear interpolation, this requires the values of two nearest grid points for every direction, leading to four points in 2-D and eight points in 3-D. It should also be noted that, as we will discuss later, the $\mathbf{E}$ and $\mathbf{B}$ are defined on a staggered grid (see Fig. 1), which means that the grid points required for each field component may be different.

Advancing particle generalized velocity from time step $t_i - 1/2$ to $t_i + 1/2$ is done using the so-called Boris pusher [4], [5], which is a second-order accurate time-centered numerical technique that has been successfully applied in many

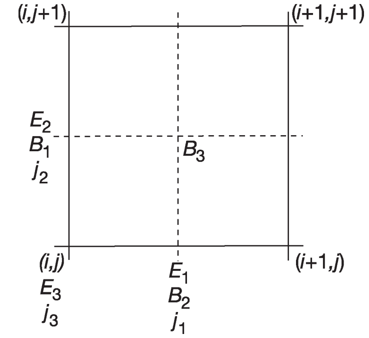

Fig. 1. Staggered 2-D grids in PIC codes.

simulation algorithms [6], and in particular, it has been widely adapted for PIC codes. This technique separates the effects of the electric and magnetic forces in four steps, starting at a time $t_i - 1/2$, as follows: i) add half the electric impulse to $\mathbf{u}$, obtaining $\mathbf{u}'$; ii) rotate $\mathbf{u}'$ with half the magnetic impulse, obtaining $\mathbf{u}''$; iii) rotate $\mathbf{u}'$ with the full magnetic impulse using $\mathbf{u}''$; iv) add the remaining half of the electric impulse.

Using the generalized velocity at time step $t_i + 1/2$, we can then advance the particle positions by doing

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \frac{\mathbf{u}_{i+1/2}}{\gamma_{i+1/2}}\Delta t \tag{2}$$

To advance the EM field values, we rewrite Maxwell's equations, particularly Faraday's and Ampére's law, to give (again in cgs units)

$$\frac{\partial \mathbf{E}}{\partial t} = 4\pi\mathbf{j} - c\nabla \times \mathbf{B} \tag{3}$$

$$\frac{\partial \mathbf{B}}{\partial t} = -\, c\nabla \times \mathbf{E}. \tag{4}$$

Starting from a known set of field values at $t = 0$, we can then advance the EM field components at each time step provided that we find the rotational operator of the $\mathbf{E}$ and $\mathbf{B}$ fields and electric current $\mathbf{j}$ resulting from particle motion. The rotational operator is approximated by a finite-difference operation on the grid using the technique developed by Yee [7]. To improve accuracy, the field values are not defined in the same points inside a grid cell but rather as staggered grids, as shown in Fig. 1, which effectively results in a second-order spatially accurate algorithm. To improve the time accuracy, we advance the fields in three steps, starting with $\mathbf{E}$ and $\mathbf{B}$ defined at a time t_i and $\mathbf{j}$ defined at a time $t_{i+(1/2)}$, as follows: i) advance $\mathbf{B}$ by half a time step using (4); ii) advance $\mathbf{E}$ by a full time step using (3) and the intermediate $\mathbf{B}$; iii) advance the intermediate $\mathbf{B}$ by the remaining half time step. This method allows one for a second-order accuracy with no memory penalty.

The electric current resulting from particle motion also needs to be defined in the staggered grid. However, because of the finite difference approximation of the rotational operator, a simple interpolation of $q\mathbf{u}/\gamma$ is not enough since it will lead to charge conservation errors. To overcome this, current deposition in PIC codes has currently two widely used approaches, namely, the Villasenor–Bunemann method [8] and the Esirkepov method [9]. Both methods ensure exact (analytical)

TABLE I
NUMBER OF SMP AND TOTAL NUMBER OF SP FOR SOME CUDA-ENABLED HARDWARE

GPU model	SMP/SP
Quadro FX1800	8/64
GeForce 9800 GX	16/128
GeForce 8800 GT	14/112
Tesla C1060	30/240
Tesla C870	16/128

charge conservation, and for linear interpolation, both lead to the same result. However, the Villasenor–Bunemann method requires that particle motion is split into motion segments lying inside the same cell, which is generally realized with a set of "IF" statements, so the Esirkepov methods generally yields better performance [10], so we chose the latter for our implementation.

III. CUDA IMPLEMENTATION

A. CUDA Overview

CUDA [11] is both a hardware and a software architecture for creating general purpose programs on a GPU. At the hardware level, it is available for NVIDIA's GeForce series (8000 and better), the Tesla systems, and the 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*: each part 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.

CUDA hardware has a set of multiprocessors [called *Streaming Multiprocessors* (SMP)], whose number varies depending on the GPU model. In the current architecture, each multiprocessor has eight processors [called *Scalar Processors* (SP)], which set the number of concurrent threads. Table I shows the number of SMP and the total number of processors for some of the CUDA-enabled products we have access to.

In what memory is concerned, all the SP cores in a chip have access to whole RAM memory of the device, which is called the *Global Memory*. In addition, all SP cores in an SMP have access to a certain amount of memory in the SMP, called the *Shared Memory*. This shared memory is usually 32 kB in size and has access times two orders of magnitude faster than global memory—typically, 400 cycles for global memory and four cycles for shared memory. Finally, each SP also has access to its own local memory (8 kB) and 32-bit registers (32).

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 the fast shared memory and synchronizing their execution to coordinate memory accesses. From the programmer's point of view, all threads in a block can cooperate (sharing memory and synchronizing) as if they are running concurrently. On the other hand, 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.

B. Particle Push

Due to the lightweight thread architecture in CUDA, it is possible to launch hundreds of thousands of threads at once and let the CUDA implementation serialize the blocks of threads. This means that it is possible to push just one particle per thread and have as many pusher threads as particles or to push several particles per thread. We have implemented a parameter in our code that defines the number of particles pushed per thread. This allows one for a customized balanced between calculation, memory usage, and memory bandwidth.

To make sure that we have the maximum occupancy of the GPU, we have to estimate the number of threads per block, which depends mainly on the amount of the device's memory usage. In our implementation, we found 64 to be the best choice. Since the number of particles might not be divisible by the number of threads per block, we can either add a test at the beginning of each push kernel that checks if the current thread number accesses a valid particle index or do two launches of the kernel: in the first launch, we launch as many threads blocks as possible without going beyond the total number of particles, and the next launch processes the rest of the particles in one incomplete block. We have found the latter approach to be marginally faster than the former (about 1% better with 131 thousand particles).

CUDA hardware only has one double precision arithmetic unit for each SMP. This means that calculations in double precision have a performance that is $\sim 9\times$ lower than in single precision. To allow one for the use of single precision calculations, and thus fully utilize the computational power of the CUDA hardware, particle positions cannot be defined using the corner of the simulation box as a reference. Instead, we must keep track of the cell were the particle is located and define the particle position referenced to the cell corner. In our case, absolute positions are defined with two arrays, namely, one contains the cell number the particle is in (integer) and the other the normalized position (a single precision float in the interval [0, 1]) inside the cell, referenced to the lower left vertex. As particles leave a given cell, their positions are corrected so that they are referenced to the new cell. These extra calculations incur in a small performance penalty, but the overall gain from using single precision is far greater.

Another relevant issue in high-performance codes is memory bandwidth. In order to increase memory access bandwidth, the copy from the global to the shared memory of particle quantities like $\mathbf{r}$ and $\mathbf{u}$ is done in a coalescent fashion by all threads in a block whenever possible. Memory bandwidth from/to global memory can be further increased by sorting the arrays with the particles' data (relative position, cell number, velocity, and charge) according to the cell each particle belongs to. If these

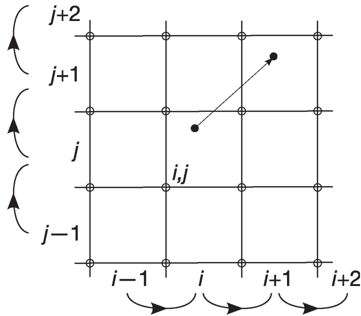

Fig. 2. Illustration of current deposition in the Esirkepov method. The 16 marked cells have current deposited, although the bottom four and the left four have no deposit, as stated in [9].

arrays are sorted in global memory, then all accesses have maximal coherence, in particular, while accessing the EM field array for interpolation. This ensures that accessing particle data are always coherent if we have one thread per particle.

The code allows one for each thread to loop through several particles, pushing one at a time. In this case, coherence in reading and writing particle data is lost. However, there is an important gain in reading $\mathbf{E}$ and $\mathbf{B}$ fields, since it is likely that the same values will be used by several particles in the same push thread. Not only less global memory access is needed per particle, but also the computational operations of each push thread is increased over the memory access operations (less data access and more computation). Hence, for each particle in a thread, a check is done: if the particle belongs to a different cell than the previous particle, new values for the EM field are read from the global to the local (registers) memory; if the particles belong to the same cell, then the the EM-field values have already been read and can be used for interpolation.

Finally, an interesting feature of modern graphic processors is that they contain texture units that can do zeroth- and first-order interpolation. Thus, if we store the EM field as textures, linear interpolation at the particle's position can be done by hardware simply as a texture fetch. We have achieved an increase in performance of $3\times$ over a nontexture fetch interpolation. However, this approach limits the usage of variables stored as textures to read-only. It is possible to update a texture with new values by changing its memory buffer to global memory, but we found the overhead too expensive to be worth it. Nonetheless, and although in this paper we use only linear interpolation, it is interesting to note that higher order interpolation have also been achieved in GPUs in general [12] and in CUDA-enabled products in particular [13].

C. Current Deposition

Most of the current deposit algorithm explained in [9], where for each particle we calculate the amount of current to be deposited for each of the cells crossed during the last time step motion, has a straightforward implementation in CUDA architecture. However, since each particle will deposit current in up to 16 cells in 2-D linear interpolation (see Fig. 2), particles from many cells may be contributing to the current in any given cell.

This step is then likely to produce a significant amount of memory collisions, as several different threads try to deposit the current in the same cell. These threads can be from the same block, in which case, one might devise some kind of synchronization between them, but they can also belong to different blocks, in which case, synchronization is harder to implement in an efficient way. However, several strategies are available.

One solution is to serialize current deposition. This is very simple to implement at the cost of a significant performance penalty, even after parallelizing the deposition of each current component. The cost depends on the number of threads that would have to be queued. In CUDA, we would be effectively running hundreds of thousands of threads (from one to a few particles per thread) almost in serial, just three at a time (one concurrent thread for each space component). The performance hit is so significant (about two orders of magnitude) that it is more effective to transfer the data back to the CPU just for the current deposition and transfer it back to the GPU for the rest of the simulation loop.

Another possibility is to take advantage of the particle sorting per cell referred previously. If the particles are sorted according to the cell they belong to (before the push), then we can accumulate the current on one thread per cell basis: each thread loops through all the particles in a cell and deposits the current at each of the 16/25 vertices. If synchronization is guaranteed, then we can be sure that only one cell is updated at a time. This approach has the drawback of adding a sorting step at every simulation cycle. The sorting penalty could be minimized by adding more cells per thread and thus sorting every other time step. A similar approach has been done in [14] by dividing the simulation space in slabs that get assigned to threads blocks. Since collisions are avoided in a block [15], no sorting is required in a slab but only between slabs.

A third possibility is to use atomic operations. These operations ensure that there are no memory collisions by creating a memory lock on a memory position being written by a given thread. CUDA offers a limited set of atomic operations that act both on global and shared memory. In particular, it provides an atomic operation that allows one accumulation of values for integer values, although such operation is not available for floating point data. To overcome this limitation, we use the `atomicExch()` function, which is also available for floating point numbers. Given a memory address and a given value, this function exchanges the given value with the one currently in memory.

Using this function, it is possible to devise a simple algorithm that simulates any atomic operation, as long as that operation is commutative and has a neutral element, as follows.

1) Given a work value to accumulate at a certain position in memory.
2) Ensure that it is different than the neutral element.
3) Atomic-exchange the previous value at that memory position with the neutral element for the accumulation function.
4) Accumulate the previous value with the work value, generating a new value.

5) Atomic-exchange the new value at the memory position.
6) If the value we got back *is not* the neutral element, this means that some other thread placed this value there as a result of its own calculation. So use this value as a new work value and go to 3.
7) Otherwise, if it is the neutral element, finish.

The following pseudocode illustrates the steps described previously:

function PSEUDOATOMICOPERATION(address, value)
  $work_value \leftarrow value$
  while$(work_value \neq neutral_element)$ **do**
    $prev_value \leftarrow$ **atomicExchange**$(addr., neutral_element)$
    $new_value \leftarrow$ **operation**$(prev_value, work_value)$
    $work_value \leftarrow$ **atomicExchange**$(address, new_value)$
  end while
end function

In our first approach, we started by implementing the pseudoatomic option. There are several advantages to this approach. First, it allows one for a straightforward code which is easier to port from/to other multithreaded architectures which might have a similar atomic exchange operation but may not suffer from these global/shared memory issues. Second, this allows us to avoid load balancing issues, since we can deposit the current per particle and not per cell. On a per-cell current deposit, a nonuniform particle distribution would necessarily cause different loads throughout the threads: those dealing with cells with less particles would finish first than those dealing with more particles. With one particle per thread, or with a more general n particle per thread approach, this nonuniform thread load is avoided, since all threads are dealing with the same number of particles and thus have the same amount of calculations to do.

However, one disadvantage is that we have two atomic exchanges per particle and per cell, which might slow down the code seriously if the particles are ordered per cell inside a block of threads. A simple solution is to initialize the array of the particles' positions in a way that minimizes the chance that two consecutive particles' indexes are unlikely to be inside the same cell. Even better, all threads in a warp (the group of 32 threads that are known to be synchronized by CUDA) should be working with particles in different cells, preferably cells that are far apart so that there are no collisions inside that warp during current deposition. Although it is easy to initialize the particles so that this condition is met, some care has to be taken to ensure that it still holds throughout the simulation. As the simulation evolves and particles are moved around, this initial distribution condition will no longer hold and some current deposition conflicts will occur, causing a degradation in performance. When that degradation is higher than a certain threshold, a redistribution of the particles over the threads has to be done. This corresponds to a sorting operation and has a certain penalty. However, the threshold for the sorting can be adjusted so that this penalty is minimized in the overall performance balance.

So our final implementation included the pseudoatomic option, a sorting operation, and a *stride* distance, that is, the distance in particle index between the particles handled by one thread and the next consecutive thread. This guarantees that the atomic collisions were minimized during current deposition. The sorting is done in three steps: we first copy the cell index of each particle to a separate array; then, we generate a sort index based on this array using a radix sort algorithm available in CUDPP [16]; and finally, we shuffle the particle data using the same sort index. In Section VI, a discussion on the sorting penalty and the optimal stride value is presented. Fig. 3 shows the implementation of the stride during particle push and current deposition.

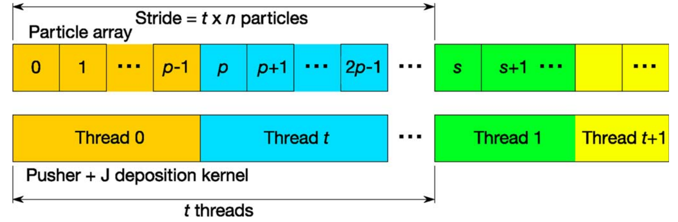

Fig. 3. Illustration of a stride during particle push and current deposition, so that kernels in the same warp handle particles in different cells. p is number of particles per thread and s is the stride.

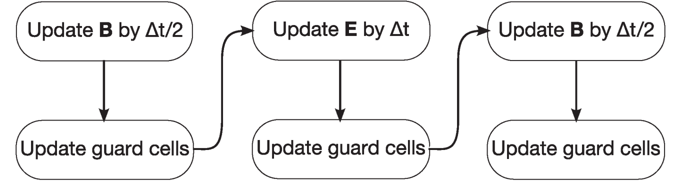

Fig. 4. Illustration of the sequence call of the three different kernels required for a full EM field update.

D. EM Field Update

Updating the EM field was done as described previously [(3) and (4)] using the staggered grid of Fig. 1 and a three-step finite-differences method. Actually, several kernels had to be implemented and launched in sequence, since this step requires at least two global synchronization steps, namely, one after the update of the first half of the $\mathbf{B}$ field and another after the full update of the $\mathbf{E}$ field. Each step is implemented as a kernel, and between them an update of the guard cells is also necessary. An overview of the sequence of different kernel launches is shown in Fig. 4. All these kernels are launched as one thread per cell—or one thread per lower left vertex of a cell.

E. Boundary Conditions and Guard Cells

The algorithm was implemented using Dirichlet periodic boundary conditions, defining the simulation space as closed and periodic. This means that the neighbors of the cells lying at the lower boundary of the simulation will be the cells at the upper boundary of the simulations and vice versa, and that particles leaving one side of the simulation box will reenter on the other side. From the point of view of the simulation grids, this is implemented through the use of guard cells, which are extra cells added at the simulation boundaries, where the values from the grid points at the other boundaries are replicated.

This allows one for a much simpler simulation algorithm, where all cells are treated equally, and guard cell values are updated after the iteration is complete. To this end, we have implemented two different kernels to handle guard cells: one adds the accumulated current density from the guard cells to its correspondent physical cells in the grid, while the other copies the EM fields from the physical border cells to the guard cells. It should also be noted that this technique is similar to what is used in distributed memory parallel PIC algorithms, where guard cells in one computation device correspond to grid points on a neighboring computation device and could in principle be used in a system where multiple CUDA devices operate cooperatively.

IV. Direct Visualization

As with any numerical experiment, visualization plays a critical role in PIC simulations. This can be a time-consuming and computationally demanding task that can benefit greatly from the fact that simulation is being run on the GPU itself. Since most of the CUDA-enabled devices available are graphic processors (NVIDIA's GeForce and Quadro boards), the data are already available in video memory and can readily be displayed, avoiding time consuming memory transfers from CPU to video memory. Even in CUDA hardware that is not a video card, such as NVIDIA's Tesla boards, the bandwidth throughput to video RAM over PCI Express $\times 16$ is very high (up to 1000 GB/s), allowing one for very efficient visualization. There are also several postprocessing visualization diagnostics, such as smoothing or energy calculations, that can make use of the available computational power on the GPU, bringing an added benefit to doing direct visualization. In this sense, we have developed code in OpenGL that tightly integrates with our PIC implementation in CUDA, expanding it to allow one for the display and exploration of the resulting simulation data. This expanded system is able to display millions of particles and also to produce several diagnostics (EM field, current, and charge density) and other custom diagnostics at interactive frame rates.

A. Particle (Point) Visualization

As with any particle code, particle point data are one of the fundamental datatypes to be visualized. As explained in Section III-B, particle positions are stored as normalized coordinates to the grid cell, using values in the range [0, 1], together with the particle cell index, stored as an integer. To display particle (point) data in OpenGL, two approaches are available: i) for each simulation particle, we could move the coordinate system to the origin of each cell in the grid and render a point in the particle position using the corresponding scale factor (the cell dimension length); or ii) we could convert the normalized particle coordinates and cell index to a global position array that would be used to draw the particles.

The latter approach was chosen in this paper, since it offers several benefits over the first, as follows.

- Handling different scaling in each dimension is easier and more straight forward than with OpenGL coordinate scaling.

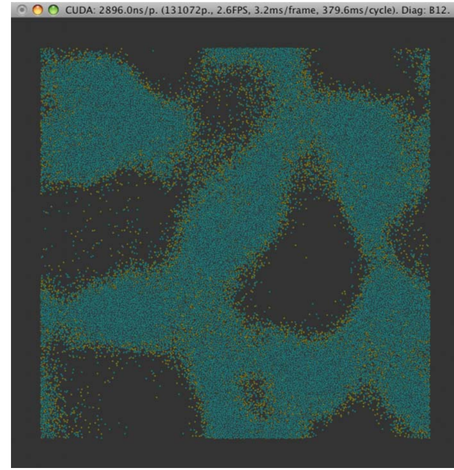

Fig. 5. Example of direct particle visualization during a full 2-D PIC simulation of a Weibel instability (positrons in cyan and electrons in yellow).

- The conversion of coordinates can be done very efficiently in CUDA, since all the data are independent from one another and is already available in the GPU memory.
- After conversion, the data can be displayed very fast in OpenGL using a Vertex Buffer Object (VBO) [17, ch. 2]. VBOs cannot be used efficiently in the first method.

Particle visualization is then done using a CUDA kernel that converts from normalized to absolute coordinates. This conversion is done in a single-pass maximizing memory bandwidth. The resulting data are then used as a vertex buffer object to be displayed in one single OpenGL render command. All operations are realized in video memory achieving a minimal overhead. Fig. 5 shows the direct particle visualization during the simulation of a 2-D Weibel instability.

B. Grid Visualization and Diagnostics

Being a particle-mesh algorithm, visualization tools for PIC codes are also required to handle grid data, and we have implemented several OpenGL routines for this type of visualization. They take advantage of the programmer interoperability of textures between OpenGL and CUDA. The main idea is to use data in CUDA global memory (GPU's RAM) as an OpenGL texture to be rendered and displayed. Again, this allows one for fast rendering, since no transfer of data from the CPU to the GPU is necessary. We have also implemented several grid-related diagnostics, which are particularly adequate for GPU algorithms, such as vector field magnitudes or EM field energies. Fig. 6 shows a direct visualization of the charge density of the electrons during the formation of a 2-D Weibel instability, described in detail in Section V later, and Fig. 8 shows the transverse magnetic field energy evolution, $B_1^2 + B_2^2$, at four different stages of this simulation.

C. Interactivity

Interactive control of direct visualization was also implemented, to simplify interaction with the simulation code and for easily switching between the several available diagnostics. The routines implementing interactivity must be unobtrusive,

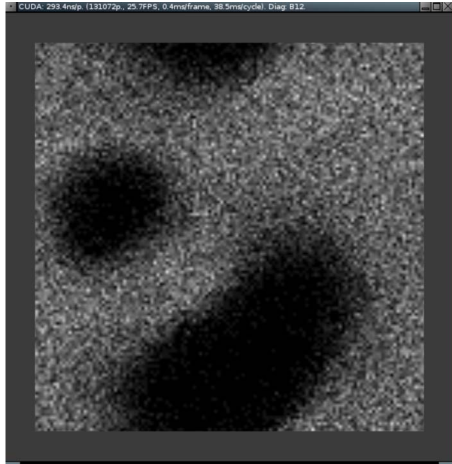

Fig. 6. Direct visualization of charge density of electrons during the formation of a Weibel instability.

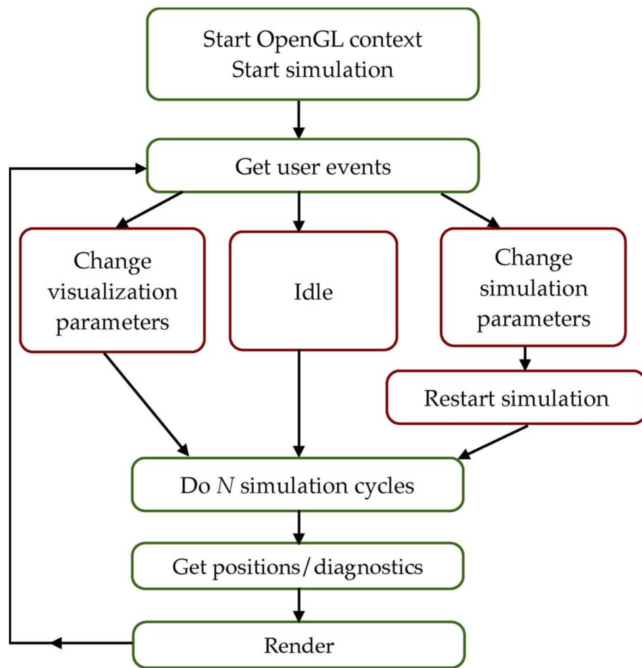

Fig. 7. Overview of the complete code.

so that the simulation performance does not suffer too much, but it also needs to offer enough interactivity to allow the user to have enough control over the visualization and diagnostics. Our choice of implementation went to OpenGL's GLUT, providing a lightweight multiplatform API, with minimal impact on performance, and a simple integration with the CUDA GPU simulation code. Fig. 7 shows a generic overview of a simulation cycle, including parsing user events and rendering. To minimize the impact on performance, it is also possible not to update the visualization at every time step, but only at larger intervals, by doing several simulation cycles per event parsed, or even to skip rendering and user interaction altogether if maximum performance is required.

Finally, we have also added the possibility to control simulation parameters interactively, like the number of particles per cell or the grid spatial resolution, restarting the simulation with the new settings. This is especially useful in testing phases, as it allows one to quickly scan the effect of certain code changes for various simulation scenarios.

V. Application to Weibel Instability

We have tested our CUDA simulation code by running a well-known benchmark problem of the Weibel instability in electron–positron plasmas [18, and references therein]. In this simulation, two plasma clouds, one electron cloud and one positron plasma cloud, initially moving perpendicularly to the simulation plane with some temperature distribution, are set to evolve self-consistently using a fixed neutralizing ion background. The simulation parameters were as follows:

Generalized velocity: $u = \gamma v = (0; 0; \pm 0.6)$, with species 1 (electrons) having the positive velocity and species 2 (positrons) the negative velocity.

Thermal distribution: uniform guassian with 0.1 width.

Grid parameters: number of cells: 128×128; total simulation space: $12.8c/\omega_p \times 12.8c/\omega_p$.

Time step: $\Delta t = 0.07$, which, together with the grid size, satisfies the Courant condition.

In this scenario, the evolution of the plasma clouds is governed by the Weibel instability, leading to the formation of current filaments that then coalesce leaving behind a set of plasma bubbles that remain empty (in 2-D), as shown in Figs. 5 and 6. These bubbles remain stable, with the plasma pressure being balanced by the radiation pressure of the EM fields trapped inside the bubble. The initial kinetic energy of the particles is partially transferred to the EM fields, leading to the formation of magnetic field loops that slowly merge until the simulation reaches a steady state, as shown in Fig. 8.

Most of the simulations were done with 36 (6×6) particles per species per cell, but this parameter could be changed interactively. The simulations completed successfully in several CUDA-enabled systems. The physical results were validated against the EM-PIC code OSIRIS [19] in single precision, using the same simulation parameters, giving consistent macroscopical results both on the growth rate of the instability and dimensions of the formed structures. The small differences found were related to a different choice of random number generator and differences in the particle order for current deposition, which leads to minor differences due to numeric roundoff issues.

VI. Results and Analysis

The performance of the CUDA PIC code was evaluated using the simulation mentioned in the previous section. The code was run with two different configuration of particles: 8 particles per cell (4 for each species), giving a total of 131 072 particles (further referenced as the 131 kpart run), and 72 particles per cell (36 for each species), giving a total of 1 179 648 particles (referenced as the 1.2 Mpart run). Table II presents an overview of the most relevant results using a Tesla C1060 board for simulation and a Quadro FX 1800 for rendering.

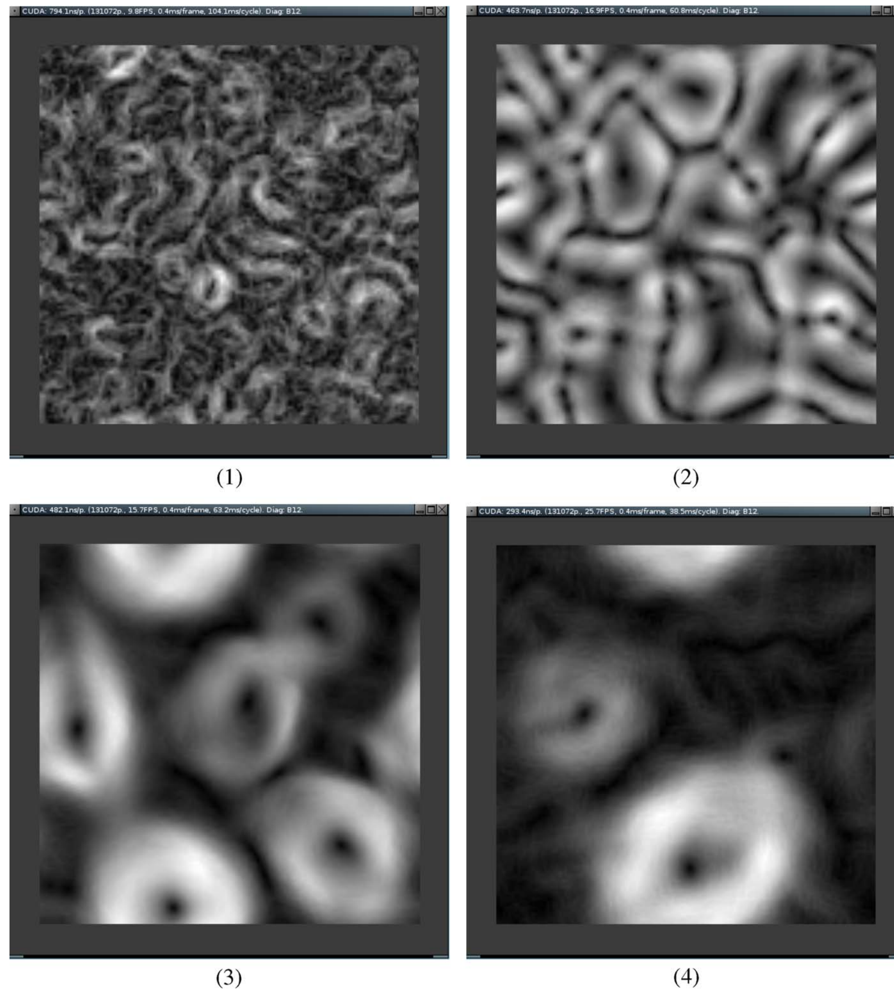

Fig. 8. Example of direct visualization of diagnostics. These four pictures are screen captures of $B_1^2 + B_2^2$ during the formation of a 2-D Weibel instability.

TABLE II
TIME MEASUREMENTS OF A FULL PIC CUDA 2-D IMPLEMENTATION OF A WEIBEL INSTABILITY. GRID DIMENSIONS ARE 128×128 CELLS. THE SYSTEM USED A TESLA C1060 BOARD FOR CUDA AND A QUADRO FX 1800 FOR RENDERING

# particles	render (ns/particle)	cycle (ns/particle)
131 kpart	0.020	72
1.2 Mpart	0.019	39

We also ran our code on the CUDA devices listed in Table I. Our purpose was to evaluate NVIDIA's claims on the portability of CUDA code across a wide range of devices of different computational capabilities. Although we noticed obvious performance differences, our code produced consistent results across all equipment. The Tesla C1060 was the one that presented the best computational performance, so it is the one used throughout this paper.

For comparison, we implemented a CPU version of the GPU algorithm and ran it on the host machine, an Intel Xeon E5420 2.50 GHz with 6 MB cache. The CPU code was compiled using `gcc` version 4.3.2 with full optimizations enabled ($-$`O3`). The results, using one core of the CPU, were 401 ns per particle on the 131 kpart run and 379 ns per particle on the 1.2 Mpart run.

It is interesting to note the difference in performance between the 131 kpart and the 1.2 Mpart runs. The reason is that below one million particles, the Tesla C1060 is not quite full yet and does not have enough threads to hide the memory latency in transfers between global and shared or register memory.

The rendering time can be considered irrelevant (less than 0.05% of a cycle). Even if we manage to lower the cycle time by one order of magnitude, it will still be acceptable if direct visualization is desirable.

We have also estimated the timings for the different parts of the algorithm. It was possible only to estimate and not have a very precise evaluation since in order to have some parts of the algorithm active, others also had to be active. For example, current deposition only works effectively with sorting enabled. Since it is possible to time the sorting step without current deposition, but not the other way around, we estimate the current deposition step by subtracting the sorting time from the total of current deposition and sorting. Table III and Fig. 9 show the time duration of the different push steps: reading and writing out the particle data from/to global memory to/from registers, sorting the particles' array according to the cell's index, interpolating the fields at the particle's position, calculating the new velocity and advancing the particle, and depositing the current for each particle.

TABLE III
TIME ESTIMATIONS FOR DIFFERENT PARTS OF THE PARTICLE PUSHER ALGORITHM. THE SIMULATION USED THE 1.2 Mpart RUN AND 128 × 128 CELLS

Step	Time (ns/particle)
Read & write	4.5
Sort	6.0
Fields interp.	0.66
Vel. & part. advance	0.4
Curr. deposition	23
Total	34.6

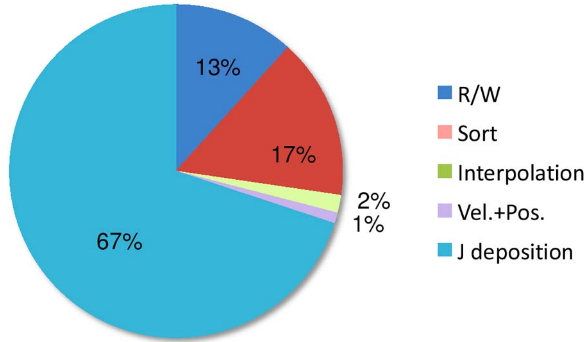

Fig. 9. Percentage of time spent at each step of the pusher algorithm. Based on the values from Table III.

The previous timings do not show the field update step, since it depends on the number of cells and not on the number of particles. The results were 9 ns per cell for the complete cycle shown in Fig. 4. During the complete simulation cycle, this value is not relevant, since usually one has much more particles (millions) than cells (tens of thousands). For example, in our 1.2-Mpart run, the total time per cycle was 46 ms, of which just 0.15 ms corresponded to the EM field update.

We have implemented several of the performance parameters exposed in previous sections and evaluated their impact on the performance of the code. The optimal number of threads per block depends on the kernel in use. We found that a number of 64 or 128 were optimal for the pusher kernel (that also includes field interpolation and current deposition), and that 128 was best for the field update. These values do not depend on the number of particles or of vertices but more on the amount of shared memory and registers required per kernel.

To avoid collisions during current deposition, it should be avoided that two consecutive threads handle particles on the same cell (Section III-C and Fig. 3). So we implemented a *stride* parameter that defines the index distance between the particles handled by two consecutive threads. Moreover, to increase the amount of arithmetic operations per memory access, we have implemented a parameter that defines the number of particles handled by each thread. Interestingly, we have found these parameters to have similar effects. When handling one particle per thread, a stride of 4× the number of particles per cell guaranteed best performance and minimal collisions. Similarly, with a stride of one, we have found that handling 4× the number of particles per cell per thread achieved a similar performance. Fig. 10(a) and (b) represent the variation in performance per particle for different particles per thread (different lines) and different stride sizes (x-axis) for the 1.2-Mpart run. Fig. 10(a) refers to particles per thread that are submultiples of the particles per cell and Fig. 10(b) to particles per thread that are multiples of the particles per cell. As already said, the best performance is achieved with a stride of 1 and 4× the number of particles per cell per thread (39 ns per particle). However, a stride of 4× the number of particles per cell with one particle per cell gets a similar performance.

Two other parameters were implemented but had to be disabled for best performance. They were the possibility to serialize the launching of kernels at the CPU level and not to sort at every time step. It was considered that to further minimize memory collisions during current deposition, the kernel threads could be launched in batches, thus guaranteeing that the CUDA device was loaded in a manner that minimized the risk of two particles updating the current in the same cell at the same time. This meant that some part of the thread block control was made at the CPU level and not by the CUDA implementation. This proved to have worse performance than launching all threads at once. Moreover, not sorting at each time step might save the 6-ns step without incurring in too much extra memory conflicts. This was not the case. The memory conflicts by not sorting increased the current deposition by several tens of nanoseconds, so that in the end best performance was achieved by sorting at each time step.

Based on the values of Table III, we also estimated how efficient the device was being used. Table IV shows the estimated number of single precision floating point operations per second [in gigafloating point operations per second (GFLOPS)] of different parts of the pusher algorithm, and also of the whole algorithm. A comparison with the peak performance of the Tesla C1060 [20] is also shown. The same code running on a single core of the Intel Xeon E5420 (theoretical peak performance of 5 GFLOPS) achieved 1.25 GFLOPS. Hence, it can be estimated that for this algorithm one Tesla C1060 can replace 3 Intel Xeon E5420 (or approximately 11 cores).

We have also evaluated the penalty for adding user interaction during the simulation. Table V shows the time it takes for our system to simulate different number of cycles with and without user interaction for the 1.2-Mpart run.

The measured penalty, although significant larger than the rendering, is also still acceptable. For simulations that run for a high number of cycles, it might be desirable to lower the user interaction priority, that is, to run several cycles per user event parsing. The same approach is to be used if the cycle time is made significantly shorter. This possibility has been included in the code, represented as the third line from the bottom of Fig. 7.

The application of this algorithm to large-scale plasma simulations is ultimately limited by the total memory available on the CUDA device (currently up to 6 GB on a Tesla C2070), with state of the art simulations [2] requiring $\sim 10^{10}$ particles to be followed for $\sim 10^6 - 10^7$ time steps, with total memory requirements going up to ∼1 TB. The solution will undoubtedly rely on the use of a (massively) parallel GPU system, where an ecosystem of CUDA devices operate cooperatively communicating through some form of interconnect, much like a distributed memory parallel computer. Our implementation of

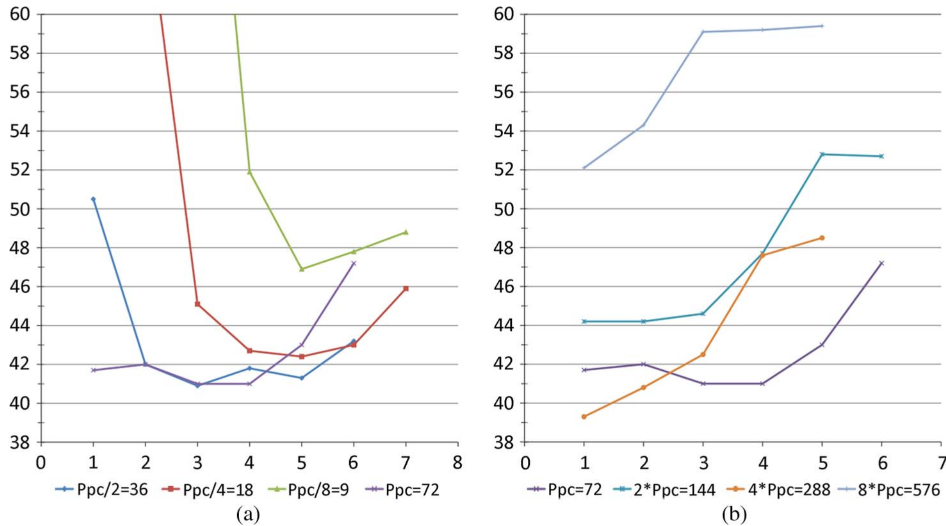

Fig. 10. Evaluation of performance (y-axis, in nanoseconds per particle) with stride size (x-axis) and particles per thread (Ppc, different lines), for the 1.2-Mpart run. Each line uses a number of particles per thread (Ppc) which is either a (a) submultiple or (b) multiple of the number of particles per cell.

TABLE IV
PERFORMANCE ESTIMATION FOR SOME PARTS AND FOR THE COMPLETE PUSHER ALGORITHM. THE SIMULATION USED THE 1.2-Mpart RUN AND 128 × 128 CELLS RUNNING ON THE TESLA C1060. BASED ON THE VALUES FROM TABLE III AND THE THEORETICAL PEAK PERFORMANCE OF 311 GFLOPS

Step	GFLOPS	Efficiency
Fields interp.	103	33%
Vel. & part. advance	200	64%
Curr. deposition	14.1	4.5%
Complete pusher	13.6	4.4%

TABLE V
ESTIMATION OF USER INTERACTION PENALTY ON THE 1.2-Mpart RUN

Cycles	No interaction (seconds)	Interaction (seconds)	Penalty
25	3.764	3.928	4.36%
250	18.533	19.309	4.19%
2500	105.463	112.099	6.29%
5000	195.112	208.265	6.74%

boundary conditions using guard cells, as explained in Section III-E, is well suited to a spatially decomposed parallel version of our algorithm, where each CUDA device is responsible for a smaller region of the total simulation space. Each CUDA device would only need to communicate with neighboring devices sending guard cell values and particles crossing the device boundary. However, this implies a device-to-host transfer when sending data to other nodes and a host-to-device transfer when receiving to be done at every time step, which may have a negative impact on overall performance. The detailed analysis of these issues is beyond the scope of this paper and will be addressed in a future publication.

VII. OVERVIEW AND CONCLUSION

We have implemented a full relativistic 2-D PIC code on a GPU using C for CUDA. We have validated the implementation using a well-known benchmark problem of the Weibel instability in electron–positron plasmas. The code performs significantly faster on a Tesla C1060 than on a single core of an Intel Xeon E5420. The main performance bottleneck is current deposition (approximately 67% of a simulation cycle), since it involves a scattering operation to global memory. We were able to avoid serializing this step by implementing a pseudoatomic add with floats. This implementation can be extended to include other kinds of atomic operations, as long as they have a neutral element and are commutative. To avoid memory access conflicts during the current deposition step, several strategies were implemented and evaluated. The best results were obtained with a particle-sorting mechanism and by ensuring that consecutive threads would deposit current in different cells. This approached achieved minimal memory conflicts during current deposition.

We have also added a direct visualization and user interaction layer to the simulation code. These allow one for more flexibility in using the simulation code and for a better overview and understanding of the results. The rendering does not slow the simulation down significantly. However, the time penalty for user interaction at every time step might be significant. To overcome that case, we have added the possibility to interactively lower the priority of the interaction layer, so that the user is able to control the compromise between better performance or more user interaction.

ACKNOWLEDGMENT

The authors would like to thank the Laboratory of Simulation in Energy and Fluids at Instituto Superior Técnico for graciously allowing us the use of its visualization workstation.

REFERENCES

[1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Comput. Graph. Forum*, vol. 26, no. 1, pp. 80–113, Mar. 2007.

[2] R. A. Fonseca, S. F. Martins, L. O. Silva, J. W. Tonge, F. S. Tsung, and W. B. Mori, "One-to-one direct modeling of experiments and astrophysical scenarios: Pushing the envelope on kinetic plasma simulations," *Plasma Phys. Control. Fusion*, vol. 50, no. 12, p. 124 034, Dec. 2008.

[3] J. M. Dawson, "Particle simulation of plasmas," *Rev. Mod. Phys.*, vol. 55, no. 2, pp. 403–447, Apr. 1983.
[4] J. P. Boris, "Relativistic plasma simulation—Optimization of a hybrid code," in *Proc. 4th Conf. Numer. Simul. Plasmas*, 1970, pp. 3–67.
[5] C. Birdsall and A. Langdon, *Plasma Physics via Computer Simulation*. Bristol, U.K.: Adam Hilger, 1991.
[6] R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles*. Bristol, U.K.: Inst. Phys. Publ., 1988.
[7] K. S. Yee, "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media," *IEEE Trans. Antennas Propag.*, vol. AP-14, no. 3, pp. 302–307, May 1966.
[8] J. Villasenor and O. Buneman, "Rigorous charge conservation for local electromagnetic field solvers," *Comput. Phys. Commun.*, vol. 69, no. 2/3, pp. 306–316, Mar./Apr. 1992.
[9] T. Z. Esirkepov, "Exact charge conservation scheme for particle-in-cell simulation with an arbitrary form-factor," *Comput. Phys. Commun.*, vol. 135, no. 2, pp. 144–153, Apr. 2001.
[10] T. Umeda, "A new charge conservation method in electromagnetic particle-in-cell simulations," *Comput. Phys. Commun.*, vol. 156, no. 1, pp. 73–85, Dec. 2003.
[11] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, Mar./Apr. 2008.
[12] C. Sigg and M. Hadwiger, "Fast third-order texture filtering," in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Reading, MA: Addison-Wesley, 2005, ch. 20, pp. 313–329.
[13] D. Ruijters, B. M. ter Haar Romeny, and P. Suetens, "Efficient GPU-based texture interpolation using uniform B-splines," *J. Graph. Tools*, vol. 13, no. 4, pp. 61–69, 2008.
[14] G. Stantchev, W. Dorland, and N. Gumerov, "Fast parallel particle-to-grid interpolation for plasma PIC simulations on the GPU," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1339–1349, Oct. 2008.
[15] V. Podlozhnyuk, "Histogram calculation in CUDA," NVIDIA whitepaper, 2007. http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/histogram256/doc/histogram.pdf
[16] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Proc. 23rd IEEE Int. Parallel Distrib. Process. Symp.*, May 2009, pp. 1–10.
[17] D. Shreiner, *The OpenGL Programming Guide, The Official Guide to Learning OpenGL*. Reading, MA: Addison-Wesley, 2009.
[18] R. A. Fonseca, L. O. Silva, J. W. Tonge, R. G. Hemker, J. M. Dawson, and W. B. Mori, "Three-dimensional particle-in-cell simulations of the Weibel instability in electron-positron plasmas," *IEEE Trans. Plasma Sci.*, vol. 30, pt. 1, no. 1, pp. 28–29, Feb. 2002.
[19] R. A. Fonseca, L. O. Silva, F. S. Tsung, V. K. Decyk, W. Lu, C. Ren, W. B. Mori, S. Z. Deng, S. Lee, T. C. Katsouleas, and J. C. Adam, "OSIRIS: A three-dimensional, fully relativistic particle in cell code for modeling plasma based accelerators," in *Proc. ICCS*, vol. 2331, *Lecture Notes Computer Science*, 2002, pp. 342–351.
[20] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 451–460, Jun. 2010.

Paulo Abreu was born in 1968. He received the B.S. degree in physics engineering from Universidade Técnica de Lisboa, Lisbon, Portugal, in 1993. He is currently working toward the Ph.D. degree at the Institute for Plasmas and Nuclear Fusion at Instituto Superior Técnico, Lisbon.

His research areas include scientific visualization and optimizing high-performance scientific codes for grid environments and for multicore architectures.

Ricardo A. Fonseca was born in Lisbon, Portugal, on September 11, 1973. He received the degree in physics engineering and the Ph.D. degree in physics, on the subject of laser–plasma electron accelerators, from the Instituto Superior Técnico (IST), Universidade Técnica de Lisboa, Lisbon, in 1996 and 2002, respectively.

Since 1996, he has been with the Laser and Plasma Group, IST. In 2000–2001, he was with the University of California, Los Angeles, where he worked on the numerical modeling of high-intensity laser–plasma interactions. He is currrently a Researcher with the Instituto de Plasmas e Fuso Nuclear, Lisbon. Since 2003, he has also held a permanent position at the Instituto Superior de Ciências do Trabalho e da Empresa, Lisbon University Institute, where he is currently an Associate Professor. He has over 65 published papers in leading scientific journals.

Dr. Fonseca was the recipient of the Oscar Buneman award in 2000. He was a Guest Editor for the IEEE TRANSACTIONS ON PLASMA SCIENCE Special Issues on Laser and Plasma Accelerator Workshop 2007 and on Numerical Simulation of Plasmas 2010.

João M. Pereira received the Ph.D. degree in electrical and computer engineering (computer graphics) from Instituto Superior Técnico/Universidade Técnica de Lisboa (IST/UTL), Lisbon, Portugal, on December 1996 and the M.Sc. and BsEE degrees in electrical and computer engineering from IST/UTL, in 1989 and 1984, respectively.

He is currently an Associate Professor with the Computer Science Department of UTL, where he teaches computer graphics. He coordinates the Distributed Interactive Graphic Systems Research Group at INESC-ID (Computer Systems Engineering Institute). His main research fields are real-time rendering, 3-D game programming, serious games, networked virtual environments, augmented reality, and parallel computer graphics.

Dr. Pereira won several prizes like the international second prize at the "1994 MasParChallenge Contest" and the national first prize "Best Graphic Interactive Demonstrator" in 2004 and 2002, respectively, with the games "Lost Ages—A Massive Role Playing Game" and "Peace and War Games: Large Scale Simulation Over the Internet." He has been involved with several European and national projects. He was also proposal evaluator of the FET during 2009. He is author or coauthor of more than 80 peer-reviewed scientific papers presented at national and international events and journals. He is member of the Eurographics Association.

Luís O. Silva was born in Lisbon, Portugal, in 1969. He received the Licenciatura in physics engineering and the Ph.D. and Habilitation degrees in physics from Instituto Superior Técnico, Lisbon, Portugal, in 1992, 1997, and 2005, respectively.

He was a Postdoctoral Fellow with the University of California Los Angeles, from 1997 to 2001. In 2001, he moved to Instituto Superior Técnico, where he is currently an Associate Professor of physics. He has held a visiting appointment at the Kavli Institute for Theoretical Physics and has been awarded the IBM Scientific Prize 2003, the 2001 Abdus Salam International Center for Theoretical Physics Medal for Excellence in Nonlinear Plasma Physics by a Young Researcher, and the Gulbenkian Prize for Young Researchers 1996. His research interests are on the physics of plasmas under extreme conditions. He is the author and coauthor of more than 120 journal articles.

Prof. Silva was selected as a Fellow of the Division of Plasma Physics of the American Physical Society in 2009.