Journal of Computational Physics 230 (2011) 1676–1685

Contents lists available at ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp

Particle-in-cell simulations with charge-conserving current deposition on graphic processing units

Xianglong Kong [a,b,*], Michael C. Huang [c], Chuang Ren [a,b,d], Viktor K. Decyk [e]

[a] Department of Mechanical Engineering, University of Rochester, Rochester, NY 14627, USA
[b] Laboratory for Laser Energetics, University of Rochester, Rochester, NY 14627, USA
[c] Department of Electrical and Computer Engineering, University of Rochester, Rochester, NY 14627, USA
[d] Department of Physics and Astronomy, University of Rochester, Rochester, NY 14627, USA
[e] Department of Physics and Astronomy, University of California Los Angeles, Los Angeles, CA 90095, USA

ARTICLE INFO

Article history:
Received 24 May 2010
Received in revised form 15 November 2010
Accepted 19 November 2010
Available online 26 November 2010

Keywords:
Graphics processing unit (GPU)
Computer unified device architecture (CUDA)
Particle-in-cell (PIC) plasma simulation

ABSTRACT

We present an implementation of a 2D fully relativistic, electromagnetic particle-in-cell code, with charge-conserving current deposition, on parallel graphics processors (GPU) with CUDA. The GPU implementation achieved a one particle-step process time of 2.52 ns for cold plasma runs and 9.15 ns for extremely relativistic plasma runs, which are respectively 81 and 27 times faster than a single threaded state-of-art CPU code. A particle-based computation thread assignment was used in the current deposition scheme and write conflicts among the threads were resolved by a thread racing technique. A parallel particle sorting scheme was also developed and used. The implementation took advantage of fast on-chip shared memory, and can in principle be extended to 3D.

1. Introduction

Particle-in-cell (PIC) methods are a well-established first-principles model that can provide a kinetic description of a plasma by following trajectories of an ensemble of charged particles in self-consistent electromagnetic fields [1]. An atomic computation cycle in a fully-explicit, electromagnetic PIC code consists basically of three parts: *field solver* for evolving electromagnetic fields by solving Maxwell's equations on a grid using finite-difference; *particle pusher* for calculating the Lorentz force on the particles by interpolating the fields to particle positions and advancing particles by solving the Newton's equation; *current deposition* for determining the current density on the grid from the particle motion. Due to their first-principle nature, PIC simulations generally require intensive computation and PIC codes have long been at the frontier of high performance computing. The local nature of the PIC algorithms makes it highly efficient for the codes to run in a massively parallel fashion using domain decomposition. For example, the state-of-art multi-dimensional PIC code OSIRIS [2,3] achieved an 86% efficiency on the IBM BlueGene/P cluster Argonne Intrepid with 32768 CPUs. PIC codes have been widely used to study complicated behaviors of plasmas. However, grand challenge applications such as inertial confinement fusion require computation at extreme scales. PIC codes constantly need to adapt to new computing platforms.

Over the past few years, modern graphics processing units (GPUs) for commodity PC hardware have evolved into programmable massively parallel general computation devices. For example, NVIDIA GeForce GTX 280 claims a peak throughput of operations of nearly 1TFLOPS. The enormous computational potential of GPUs has recently motivated many research

* Corresponding author at: Department of Mechanical Engineering, University of Rochester, Rochester, NY 14627, USA.
E-mail address: xkon@lle.rochester.edu (X. Kong).

0021-9991/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcp.2010.11.032

activities on computation on many-core processors. For example, Bowers et al. [4] developed the 3D PIC code VPIC, which took advantage of the computing capability of IBM Cell microprocessors. An intrinsic figure of merit for computation speed in PIC codes is the average time spent in moving one particle for one time step, T_{ps}. VPIC achieved a $T_{ps} = 5.9$ ns for a cold plasma using the Cell, compared with a $T_{ps} = 155$ ns for OSIRIS on an Intel i7 processor [5]. The utility of GPUs for general-purpose high-performance computations was initially limited by the fact that programs needed to be translated into graphic languages such as OpenGL or Cg. In 2007, NVIDIA released CUDA [6], a programming model using a language that is an extension to the standard C, which makes it much easier to program general-purpose GPU's (GPGPU). Recently, Decyk et al. implemented an electrostatic PIC code using CUDA, achieving a $T_{ps} = 2.3$ ns on an NVIDIA Tesla card [7]. These researches showed that the bottleneck for PIC codes is data movement rather than actual computation [4,7]. The key to reducing T_{ps} is data locality.

In this paper we present an implementation of a 2D electromagnetic PIC simulation code on GPU using CUDA. A key obstacle in writing an electromagnetic code for GPU is the many conditional branches exist in the current deposition algorithm, which are absent in the charge deposition part of an electrostatic code. We have succeeded in implementing a charge-conserving current deposition scheme that greatly reduces computational thread divergency. In addition, we have developed a sorting algorithm suitable for explicit PIC codes that help achieve high data localization. The code has also taken advantage of the high bandwidth allowed by the shared memory on GPU's. This implementation can be extended to 3D, subject to shared memory size constraints.

The rest of this paper is organized as follows. In Section 2, the details of the CUDA implementation of the PIC code are presented. In Section 3, the performance results for our implementation are given and compared with the CPU version. Our conclusions are summarized in Section 4.

2. CUDA implementation

2.1. CUDA overview

The Compute Unified Device Architecture (CUDA) [6] is a programming model developed by NVIDIA for general purpose computing on GPUs. The CUDA programming model is based upon the concept of C function-like *kernels* which are executed multiple times in parallel by multiple different *threads*. The *threads* are organized into one-, two- or three-dimensional *blocks*. A group of 32 adjacent threads forms a *warp*. Threads are created, managed, scheduled, and executed in warps. Threads within the same block can cooperate by sharing data and synchronizing their execution. Threads within a warp can communicate and exchange information even more efficiently than threads within a block. However, threads in different blocks cannot cooperate with each other efficiently. The threads of a block execute concurrently on one *streaming multiprocessor* (SM) in the GPU. Each SM consists of single precision *scalar processors* (SP) (current architecture also includes double precision SMs), each capable of executing an independent thread. On-chip *shared memory* and *registers* are also located on the SMs. The amount of on-chip memory is very limited in comparison to the total *global memory* available on the GPU. Global memory is the main memory of the GPU but it is located off-chip and therefore has a considerable latency.

2.2. Data structures

In a PIC simulation, field related quantities are discretized on a spatial grid. We use three 1D arrays, namely E, B, J, to store the serialized 2D grid data of electric fields, magnetic fields and current densities, respectively. We use the CUDA built-in vector type *float4* [6] as the data type of the arrays. The first three components of each *float4* array element are used to represent the corresponding field components in the (x,y,z) directions. The fourth component can be reserved to store diagnostic results, such as field energy.

The description of a particle in 2D PIC simulation usually needs six quantities: positions in the (x,y) direction, momenta in the (x,y,z) direction and the particle charge weight (A variable charge is useful in simulating large density variation). The particle position representation used in our code consists of the indices of the containing cell and the offsets from the lower boundaries of the cell, to better maintain accuracy. Two arrays, namely *xCell* and *xPos*, with CUDA build-in data type *int2* and *float2* respectively, are allocated to store the containing cell indices and the offsets. Empirically, the 64-bit built-in data type, such as *float2* and *int2*, can offer a slightly higher global memory access bandwidth the 32-bit data type, such as *float* and *int* [8]. The data of the momenta and charge are bundled into a *float4* array u. The first three components of each element of u represent the momenta in the (x,y,z) directions, and the last component keeps the charge. We will discuss the order in which the particles are arranged in the data array in Section 2.3.

2.3. Current deposition

Many PIC codes use the charge-conserving current deposition schemes [9] from TRISTAN [10] to avoid solving the Poisson's equation. The movement of a particle from P_0 at (x_0,y_0) to P_1 at (x_1,y_1) is split into multiple parts if the particle goes through cell boundaries. Usually, the Courant condition for the field solver limits the time step so that the particle moves less than the dimension of a single cell in each direction in one step. In this case, this can lead to five kinds of possible current splits

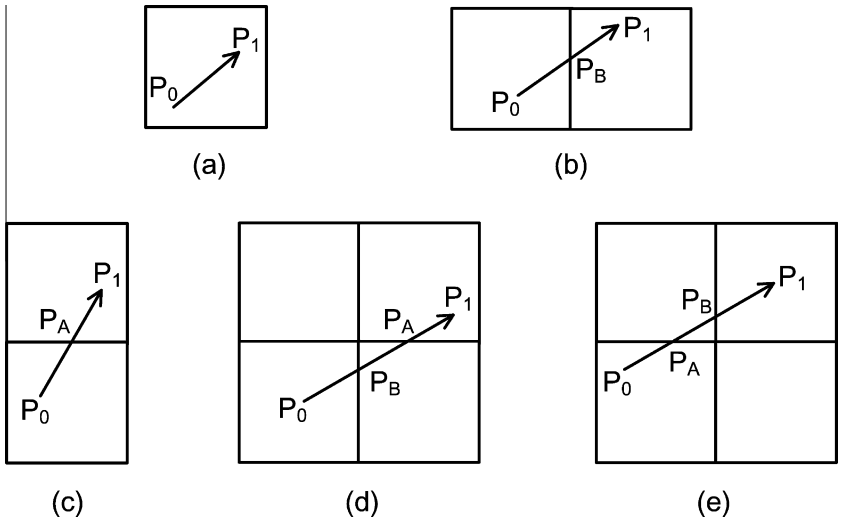

Fig. 1. Illustration of the current splits of the modified-TRISTAN method.

(Fig. 1). In CUDA, a warp serially executes each branch path taken, disabling threads that are not on that path. The data dependent conditional branches corresponding to different current split cases in the TRISTAN algorithm make it unsuitable to be implemented directly on GPU, since the conditional branches can significantly reduce the instruction throughput by causing threads in the same warp to diverge.

There was previous work on IF-free current deposition schemes which assumes a zigzag particle trajectory in one time step [11]. It was faster than the TRISTAN method but less accurate. We have developed a new algorithm based on the TRISTAN method. The new algorithm has a non-diverged form so that all currents will follow the same path to deposit. The basic idea is to split the movements of all particles into three parts by two splitting points P_A and P_B, regardless whether the particle cross the cell boundary or not (Fig. 1). The locations of P_A and P_B depend on how many boundaries the particle movement crosses:

(1) If the movement does not cross any boundary (Fig. 1(a)), P_A and P_B are at the same location where the extension line of $\overline{P_0P_1}$ intersects with the containing cell boundary.
(2) If the movement crosses one boundary (Fig. 1(b) and (c)), P_A and P_B are at the same intersection on the cell boundary.
(3) If the movement crosses two boundaries (Fig. 1(d) and (e)), P_A and P_B are at the two intersections on the two boundaries, with P_A nearer to P_0 than P_B.

After the splitting points P_A and P_B are located, the current pieces $\overline{P_0P_A}$ and $\overline{P_BP_1}$ are deposited onto the grids, and $\overline{P_AP_B}$ is deposited only when P_A and P_B are at different locations. This algorithm can be implemented using logical functions available in CUDA, to eliminate all conditional branches except the one related to depositing $\overline{P_AP_B}$. In the case of a particle staying in the cell (Fig. 1(a)), the algorithm requires more computations than directly depositing $\overline{P_0P_1}$, which is a small price to pay for reducing thread divergency. In addition, our current deposition scheme may split a current into more current pieces than the original TRISTAN scheme, which may cause additional round-off error. Current smoothing can be implemented to reduce the round-off error.

We have implemented the above modified-TRISTAN method by assigning one CUDA thread to each particle. This particle-based thread assignment is different from the cell-based thread assignment used in Ref. [7]. In theory, the cell-based thread assignment has the advantage of requiring less computation and being able to avoid write conflicts, compared to the particle-based thread assignment [12]. However, the particle-based thread assignment has the advantage of employing a large number of threads. We resolve the write conflicts by a "thread racing" technique that is similar to the "threads tagging" technique, which is (a form of speculative execution) in the NVIDIA's histogram calculation example [13].

Since the particle data is loaded from the global memory sequentially by sequential threads in a warp, the data accessed by threads in a half warp should be kept in the same segment in the global memory (coalesced) [6]. During the current deposition process, the current density on a grid point is updated multiple times, from multiple particles near that grid point. It is important to keep the accumulating current density array in the shared memory to avoid redundant transfers from global memory. However, since the size of the shared memory is limited, we cannot store the entire current density array, but only a small fraction of it in the shared memory for each thread block. Therefore, the particles need to be sorted so that the particles processed by a CUDA thread block deposit their self-generated currents only to the fraction of current density array loaded in that block. The sorting method will be discussed in the Section 2.4.

In our implementation, we group the cells into clusters, and the particles in a cell cluster are processed by a single CUDA thread block. Since the particles near the border of the cluster may move out to the cells outside the cluster, the current density array used by each block needs to have ghost cells outside the cluster (Fig. 2).

However, the ghost cells for any of two adjacent clusters can overlap. Since the threads in different blocks cannot communicate efficiently, adding the accumulated current density in the overlapping area to the current density array in the global memory can cause write conflicts among different blocks. To deal with this issue, we use a four-color scheme to deposit the currents in adjacent clusters separately. The basic idea of the scheme is shown in Fig. 3, where any two adjacent clusters always have different colors. Thus, there is no overlapping area for the clusters with the same color. The current deposition routine is executed four times, each time for clusters with a different color.

Within a cluster, write conflicts between parallel threads could still occur when updating the accumulating current density stored in the shared memory. The write conflicts can be solved easily by using floating point number atomic operations which are supported for the device with compute capability 2.0 [6]. However, even without a device of compute capability 2.0, we can still use an intra-warp technique similar to the "threads tagging" technique in the NVIDIA's histogram calculation example [13] to solve the write conflict issue. The pseudo-code of the implementation of the new technique is shown in List 1.The approach is similar to a "floating point number atomic operation". In the original "threads tagging" technique, the five most significant bits of the quantity that need to be updated are masked and replaced with the tag of the thread, which only works for data with sufficiently narrow bit widths represented with a native integral type. To deal with floating point numbers, we introduce an array of the *volatile* type for threads racing in the shared memory. This racing array has the same dimension as the accumulating current density array of the cluster. For each attempt, at first, instead of adding the data to the accumulating current density array directly, the threads in the same warp write their tags to the corresponding locations in the racing array. If two or more threads try to write their tags to the same location in the racing array, only one thread is guaranteed to succeed [6], and that thread can continue to update the data in the accumulating current density array. After successfully adding their data, the threads withdraw from the subsequent races. All the remaining threads will continue the execution until all the threads in the warp successfully add their data to the accumulating current density array and exit the loop. Finally, the accumulating current density array is added to the current density array in the global memory.

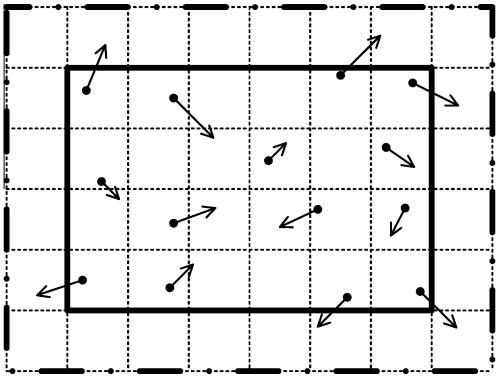

Fig. 2. The area of current density array for each block is surrounded by dash-dot line. The area of the cluster is represented by the solid line. The cells between the dash-dot line and solid line are ghost cells. Particle movements are represent by the arrows with a dot representing the start point of the particle.

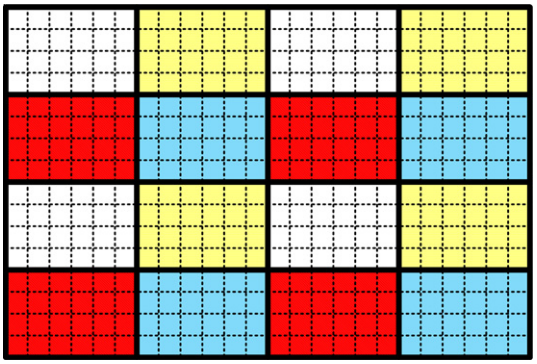

Fig. 3. (Color online) Illustration of the four-color scheme. The dash grid represents the cells. The solid grid represents the clusters.

```
__device__ void addFloat(
                        float4 *s_Jay,        //current density array in the shared memory
                        volatile ushort *s_race,   // array used for threads racing
                        uint ij,              // s_Jay[ij] is the element that need to be updated
                        float4 data,          //data that need to be added
                        ushort tag)           //current thread's tag
{
    bool flag=false;
    do
    {
        s_race[ij]=tag;
        if(s_race[ij]==tag)
        {
            s_Jay[ij] += data;
            flag=true;          //data has been added to s_Jay[ij] successfully
        }
    } while(!flag);
}
```

List 1. The pseudo-code of the implementation of current deposition.

2.4. Particle sorting

In order to ensure that all particles in a cluster are stored contiguously and can deposit to the accumulating current density array in the shared memory, they need to be sorted based on the cluster index every time step. Since they can only move less than a certain distance in a time step, most of the particles stay in the same cluster if the size of cluster is not too small. The order of the particles within a cluster is irrelevant for our deposition scheme. Therefore, performing a full particle sort is unnecessary and too time-consuming. We have developed a four-pass scheme that can efficiently fulfill the sorting requirement and rearrange the particle data. An important feature of this scheme is that the particle data undergo minimum reshuffling and the data rearrange time is greatly reduced compared to a full sort. The time complexity of our sorting scheme is $O(\eta N)$, where N is the total particle number and η is the fraction of the particles that cross the cluster boundaries in each step.

To achieve this, we divide each particle-related array (*xCell*, *xPos*, *u*), into segments, one segment for each cluster. The size of each segment is always larger than the total size of the particle data in the cluster, leaving a buffer region to store the data of the particles that will move into this cluster from the adjacent clusters. The number of particles in each cluster is recorded by an array *bufferIndex* whose dimension is equal to the number of clusters. If we refer to the section that a single particle will occupy in the particle-related arrays as a *slot*, then the elements in *bufferIndex* can be understood as the local index of the first slot of the buffer in a segment. Compared to the defragmentation step in Ref. [12], which requires sorting within a cluster first, our data structure with buffer can effectively reduce the data movement within a cluster.

The steps of the four-pass sort scheme are as follows:

(0) At the end of the current deposition routine, the particles are pushed to new positions, and the *xCell* and *xPos* arrays are updated. By examining the data in the *xCell* array, one can determine which cluster a particle has moved into. Since a particle either stays in the same cluster, or moves to one of the eight adjacent clusters, we can use 4 bits of an integer to code the nine distinct possibilities (There are eight possible adjacent clusters to move into, plus staying in the current one). The migration codes of four particles are combined into one integer to reduce the amount of data movement.

(1) Two adjacent clusters (Fig. 4(a)) in the horizontal direction are combined into a bi-cluster (Fig. 6(a)). This requires that the number of clusters in a row be even. If the number is odd, extra empty cells need to be padded to the edge of the original simulation space to make it even. This padding needs to be done in the data initialization step for the entire code. A CUDA thread block is assigned to a bi-cluster.

(2) In this step, the migration codes of the particles in the left cluster of the bi-cluster are checked, and the particles that move to the right cluster are rearranged. Each slot in the data region of the left cluster is assigned to a CUDA thread. If the particle moves to the right cluster, the particle data in the corresponding slot is copied to the first slot of the buffer region of the right cluster. Since there may be multiple particles moving to the right cluster, the thread racing method described in the current deposition routine is again used here. Upon a successful copy (say, particle 3 in Fig. 4(c.1)), the corresponding migration code of the leaving particle is reset. The data in the last slot of the data region in the left

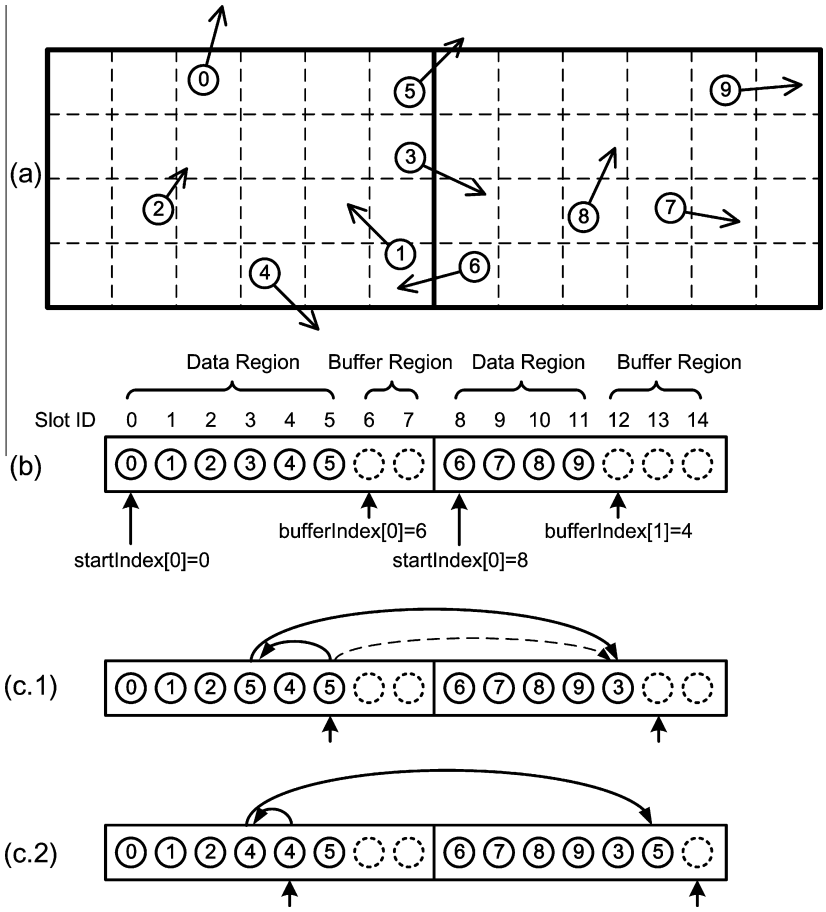

Fig. 4. (a) Illustration of particle movement in the first bi-cluster. The particles are identified by a unique particle ID only for illustration purpose. (b) Illustration of the data structure of the first bi-cluster in the particle-related arrays. The arrows point to the first slot in the buffer region. (c) Illustration of the step (2) of the particle sorting. (c.1) Particle 3 and 5 move to the right cluster, which makes slot 3, and 5 enter the loop. Slot 3 successfully copies its data (particle 3) to slot 12 in the first attempt, thereafter particle 5 fill into slot 3. (c.2) Slot 3 copies its data (particle 5) to slot 13, because particle 5 moves to the right cluster. Thereafter particle 4 fills into slot 3. As a result, the data region of the left cluster stores the data of particle 0, 1,2, 4, and the data region of the right cluster stores the data of particle 3, 5, 6, 7, 8, 9.

cluster (particle 5 in Fig. 4(c.1)) is copied to the vacant slot. The corresponding element in the *bufferIndex* for the left cluster decreases by 1. This "one particle leaves, one particle fills in" copy procedure needs to continue until the particle that ultimately fills in this slot does not move to the right cluster (Fig. 4(c.2)).

(3) Check the migration codes of the particles in the right cluster in the bi-cluster, and rearrange the particles moving to the left cluster, similar to step (2).
(4) Shift the location of bi-cluster to the right by one cluster (see Fig. 6(b)).
(5) Repeat the steps (2) and (3). For the bi-cluster which consists of the cluster in the first column and cluster in the last column, the cluster in the last column is regarded as the right cluster in the bi-cluster. By completing this step, all the particles moving to adjacent clusters in the horizontal direction have been rearranged.
(6) By applying a similar procedure as in steps (2)–(5) to bi-clusters in the vertical direction, the particles moving to adjacent clusters in the vertical direction can be rearranged. One should also notice that the particles moving to the cluster in the diagonal direction (such as the upper-left cluster) are rearranged correctly by going through a horizontal sort followed by a vertical sort.

However, for simulations with highly non-uniform particle distributions, buffer overflows may occur when the number of particle in a cluster exceeds the number of slot of that cluster. To deal with this, we adopt a flexible data structure (See Fig. 4(b)), with an additional array *startIndex* whose dimension is equal to *bufferIndex*. The array *startIndex* can be used to store the first particle index in each segment so that the number of slots in each cluster can vary. When particle-slot ratio in a cluster exceeds a certain threshold A, a global data rearrangement (See Fig. 5) is triggered to increase the buffer size of this cluster by reducing the buffer size of another cluster whose particle-slot ratio is below another threshold B.

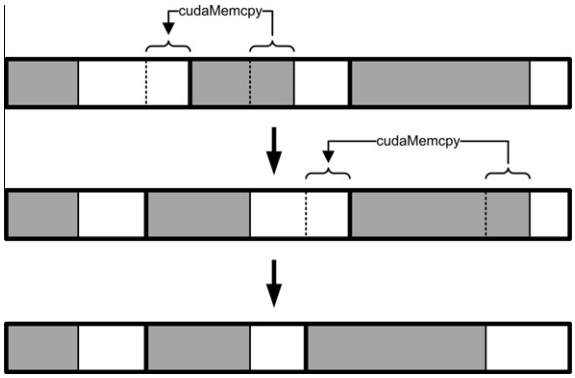

Fig. 5. Illustration of a global data rearrangement involving three clusters. The grey areas represent the data region. The white areas represent the buffer region. As a result, the buffer of the right cluster is increased by decreasing the buffer of the left cluster. In each step, the data movement is done by using the CUDA function *cudaMemcpy*.

2.5. Field solver

In this part, the two time-evolved Maxwell's equations are solved [9] in each time step to update the electromagnetic fields defined on a staggered grid in the simulation space [2]. The CUDA implementation for this part is straightforward.

For example, the x component of the electric field E_x is solved by the following explicit differential equation:

$$E_{x,i1,i2}, = E_{x,i1,i2} - 4\pi\,\mathrm{d}t \times J_{x,i1,i2} + c\,\mathrm{d}t \times \frac{B_{z,i1,i2} - B_{z,i1,i2-1}}{\mathrm{d}y},$$

where $i1$and $i2$ are the indices of the corresponding grid point.

The above equation can be implemented straightforwardly and easily by assigning one CUDA thread to each $E_{x,i1,i2}$ at the grid point.

2.6. Particle pusher

Once the electromagnetic fields are updated, the particles are pushed to new momenta in this part. Since the particles are already sorted based on the index of their containing cluster, we can assign a CUDA thread block to each cluster, and each particle is processed by a CUDA thread. The implementation is rather straightforward. Since every particle needs the electromagnetic field array for interpolation, the electromagnetic field arrays of each cluster are loaded into the shared memory in each block to avoid redundant data transfer from the global memory. Each thread interpolates the electromagnetic field to the current particle position and the particle is pushed to a new momentum according the relativistic Newton's equation. (The positional move is performed in current deposition where the particle migrations codes are calculated.)

3. Performance results

We have benchmarked the performance of our GPU code against the well-tested CPU PIC code OSIRIS, with a series of simulations involving both cold and warm plasmas. The correctness of the GPU algorithms was verified by comparing the fields and currents from the two codes. For the simulations shown in Tables 1 and 2 and Fig. 6, a simulation box of $78.0 \times 70.0\ (c/\omega_p)^2$ with 780×700 cells and 20 million electrons (36 particles per cell) were used. Periodic boundary conditions were used. The electrons in the warm plasma runs had an initial semi-relativistic Maxwellian distribution with an isotropic temperature, $f(p_x,p_y,p_z) = (2\pi p_{th}^2)^{\frac{3}{2}} \exp\left(-\frac{px^2+py^2+pz^2}{2p_{th}^2}\right)$. Here p_i's $(i = x,y,z)$ are the particle momentum and are

Table 1
Performance results of simulations with T_{e0} = 0 eV.

Procedure	T_{ps} (ns)		Speed up	Percentage of bandwidth limit (%)
	GPU	CPU		
Particle pusher	0.58	114.31	197	59
Current deposition	1.39	73.01	53	25
Particle sorting	0.38	63.19	166	7.4
Field solver	0.14	1.81	13	45
Total	2.52	205.06	81	31

Table 2
Performance results of simulations with T_{e0} = 1 keV.

Procedure	T_{ps} (ns)		Speed up	Percentage of bandwidth limit (%)
	GPU	CPU		
Particle pusher	0.69	128.62	186	50
Current deposition	3.27	75.54	23	11
Particle sorting	0.67	62.48	93	4.6[a]
Field solver	0.14	1.82	13	45
Total	4.81	221.87	46	16

[a] This result was calculated by assuming that about 0.53% of the total particles move out of the cluster after one time step.

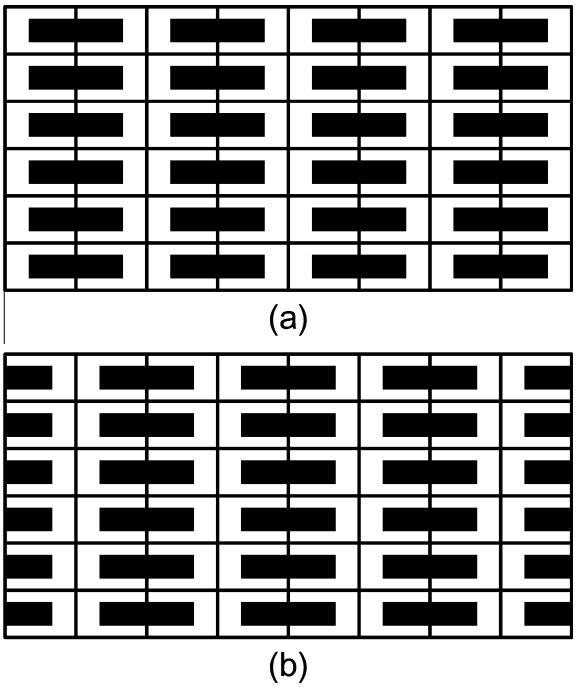

Fig. 6. Each cell in the figure represents a cluster. Two cells with a black bar on them represents a bi-cluster. (a) The bi-cluster combination pattern in step (1) and (b) the bi-cluster combination pattern in step (4).

related to the particle velocity v_i's through $p_i = \gamma m_e v_i$ with $\gamma = (1 - v^2/C^2)^{-\frac{1}{2}}$. The initial electron temperature can be defined as $T_{e0} = 2\left(\sqrt{p_{th}^2 + 1} - 1\right)m_e C^2$. All simulations were run for 1000 time steps with a time step of $0.07\omega_p^{-1}$. In the OSIRIS runs, the particles were sorted every 25 time steps, whereas in the GPU runs they were sorted every time step. In the GPU runs, the size of a cluster was chosen to be 13×7 cells, yielding an accumulating current density array for the cluster of 16×10 elements. The size of the buffer region for the particle arrays was set to be 30% of the size of data region.

All the OSIRIS simulations were with double precision run on a PC with an Intel Core 2 Duo E7200 2.53 GHz CPU, using only a single core, and 4 GB of RAM. The GPU code was run on the same PC with a GeForce GTX 280 graphic card manufactured by EVGA® with 1 GB memory and standard core clock speed of 1.3 GHz. The operation system was Ubuntu 9.04 with the Linux kernel 2.6.28. The OSIRIS code was compiled with GFortan 4.3.2 and GCC 4.3.2. For the GPU code, CUDA (version 2.3) in combination with NVIDA graphics card driver (driver version 190.18) were used. The NVIDIA CUDA Compiler (NVCC) and GCC 4.3.2 were used to compile the GPU code. Both codes were compiled with the optimization options –O3 enabled for the corresponding compilers.

In Table 1, the times spent in each process are listed for both GPU and CPU codes for a cold plasma run where virtually no particle leaves the cell. Also listed are the GPU speed-up over CPU and the GPU performance measured against the bandwidth limit, defined as T(BL)/T(GPU). The bandwidth limited time T(BL) is the time necessary to just move the data needed for each process, without any computation, at the peak memory bandwidth of 141.7 GB/s. We choose T(BL) as the ultimate performance benchmark because for PIC codes the performance is limited more by bandwidth than by math operation capability.

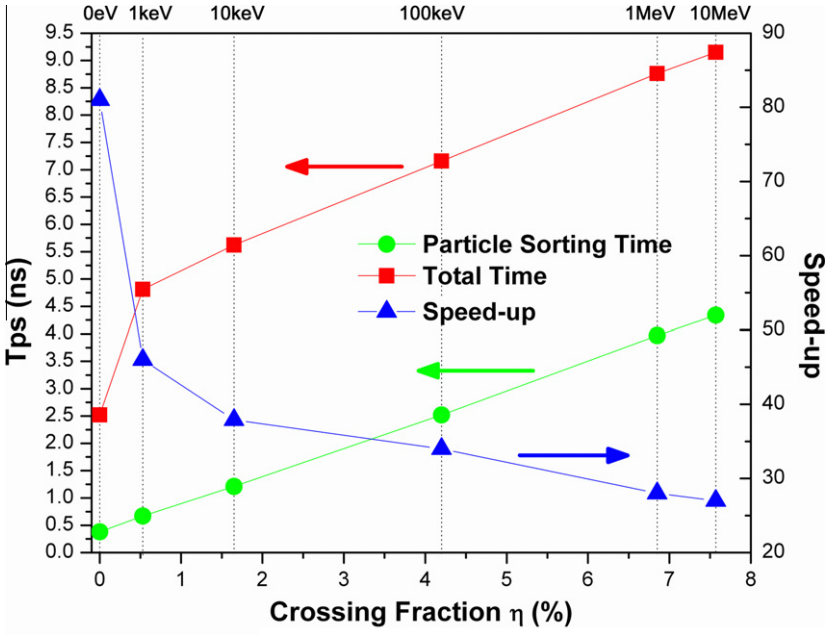

Fig. 7. (Color online) Performance of simulations with different plasma temperatures.

Since there were more particles than cells, the simulations spent most of the time in the three particle-related processes: *particle pusher, current deposition*, and *particle sorting*. The GPU *particle pusher* achieved a high 59% bandwidth limit, showing the advantage of the particle data coalescence and storing the cluster field data in the shared memory. Notice the last column of Table 1 was calculated against the peak bandwidth limit for data *copy* within the GPU memory. The actual bandwidths for *read/write* and between the memory and shared memory/register are smaller. It achieved a speed-up of 197 over the CPU *particle pusher*. The GPU *current deposition* achieved a lower 25% bandwidth limit because there were write conflicts that needed to be resolved via the thread racing technique. Even so, it still achieved a speed-up of 53 over the CPU code. The smaller speed-up compared to *particle pusher* reflects the relative inefficiency of the thread racing technique used. The GPU *particle sorting* achieved a speed-up over CPU of 166 but only a 7.4% of the bandwidth limit. Since virtually no particle moves, there was no particle data rearrangement and the time was spent entirely on reading the migration code array from the global memory into the shared memory and examining it. This showed that the bandwidth within a conditional loop is much smaller than the quoted peak value. For the cold run, the GPU code achieved a T_{ps} of 2.52 ns, a speed-up of 81 over the CPU code.

The cold run represents the asymptotic limit for T_{ps}. In warm plasma runs, T_{ps} would increase, mainly due to the increase of the current deposition and *particle sorting* times. In Table 2, information is listed for a warm plasma run of electron temperature T_{e0} = 1 keV. Compared to the cold run, the *current deposition* time increased, reflecting the cost of thread divergence when some of the particles crossed cell boundaries and some did not. The increase of the *particle sorting* time was from particle data rearrangement, now that a fraction of the particles crossed the cluster boundary. Our sorting scheme is about 14 times faster than a full sort based on the radix sorting algorithm [14] in the CUDA library. The particle crossing fraction linearly increases with the particle thermal velocity and decreases with the cluster size. For the cluster size used here, the crossing fraction η was 0.53% for T_{e0} = 1 keV and would be capped at ~8% for extremely relativistic T_{e0}'s where most particles move with the speed of light. The extra *particle sorting* time, compared with the cold run, increased linearly with η (Fig. 7). In comparison, the subsequent increase of the current deposition time with T_{e0} is very weak and for $\eta > 4\%$ (T_{e0} = 1 - MeV), the *particle sorting* time exceeded the current deposition time to become the dominant part of T_{ps}. For extremely relativistic cases, η can be lowered if a larger cluster size, limited by the shared memory size, can be used. For the 1 keV-run, the GPU code achieved a T_{ps} of 4.81 ns, a speed-up of 46 over the CPU code. For the maximum η-case, it achieved a T_{ps} of 9.15 ns, a speed-up of 27 over the CPU code.

4. Discussion and conclusion

Although the algorithms presented in this paper are mainly 2D, they can in principle be extended to 3D. For the current split scheme in current deposition, given that a particle can at most cross four cells in 3D case, the current can be deposited by splitting into four parts and using an eight-color deposition scheme. For the particle sorting, we can use a six-pass scheme in 3D instead of the four-pass scheme. Performance of a 3D code based on these algorithms is difficult to predict accurately.

On the one hand, there are more floating-point calculations pre particle-step in 3D than in 2D. Therefore, a greater speed-up can be expected in 3D than in 2D. On the other hand, each thread in 3D may require more register and shared memory usage than in 2D, which may lower the device occupancy and results in performance degradation. For a fixed amount of GPU shared memory, the cluster size in 3D needs to be reduced. For example, a 16 KB shared memory can accommodate a $5 \times 5 \times 5$ cluster size, compared to the 13×7 cluster size in 2D used here. For a 48 KB memory the cluster size can be increased to $8 \times 8 \times 8$.

There are techniques to resolve write conflicts without using thread racing in current deposition, such as the cell-based charge deposition method used in Ref. [7]. Our present implementation was motivated by using the fast shared memory and maintaining a large enough number of threads. This is especially important in 3D and high-order current deposition schemes where the shared memory size limits the number of cells in a block to only a few or even one. Also, the algorithm here may gain further speed-up through the native floating-point number atomic operations now supported for the device with compute capability 2.x [6].

Due to floating point non-associativity, the thread racing technique used in the code is in principle non-deterministic in the CUDA programming model. The associated randomness, due to round-off errors when particles are processed in a different order, is no different from the randomness introduced when varying the number of processors on a distributed memory machine. In practice, we found that our code is deterministic and reproducible on the hardware described here. In all benchmark runs shown here, the difference between the results from this code and the double-precision OSIRIS results was on the order of single precision roundoff error. In addition, the plasma physics that PIC codes are used to study is concerned mainly with the statistical macroscopic properties of a system, not its underlying microscopic states. Therefore, the theoretical additional randomness due to thread racing should be of little concern in practice.

In summary, we have presented an implementation of a 2D electromagnetic PIC code, with charge-conserving current deposition, on GPU with CUDA. The GPU implementation was found to run 27–81 times faster than a single-threaded state-of-the-art CPU code, depending on the plasma temperature. It achieved a processing speed of 2.52 ns per particle for cold plasma runs and 9.15 ns per particle for extremely relativistic plasma runs. The implementation took advantage of the fast on-chip shared memory. The thread racing technique used also provides a general method of resolving write conflict among computation threads on GPU. The method presented here can in principle be extended to 3D.

We acknowledge useful conversations with Warren Mori and Alice Quillen. This work was supported by U.S. Department of Energy under Grant Nos. DE-FG02–06ER54879 and DE-FC02–04ER54789 and by NSF under Grant Nos. PHY-0903797 and CCF-0747324.

References

[1] C.K. Birdsall, A.B. Langdon, Plasma Physics via Computer Simulation, Institute of Physics, Bristol, 1991.
[2] R.G. Hemker, Particle-in-cell modeling of plasma-based accelerators in two and three dimensions, Ph. D. thesis, University of California, Los Angeles (2000).
[3] R.A. Fonseca, L.O. Silva, F.S. Tsung, V.K. Decyk, W. Lu, C. Ren, W.B. Mori, S. Deng, S. Lee, T. Katsouleas, J.C. Adam, OSIRIS: a three-dimensional, fully relativistic particle in cell code for modeling plasma based accelerators, Lecture Notes in Computer Science 2331 (2002) 342–351.
[4] K.J. Bowers, B.J. Albright, L. Yin, B. Bergen, T.J.T. Kwan, Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation, Physics of Plasmas 15 (2008) 055703.
[5] W.B.Mori, Personal Communication.
[6] NVIDIA CUDA TM Programming Guide Version 3.0., NVIDIA Corporation, 2010.
[7] V.K. Decyk, T.V. Singh, S.A. Friedman, Graphical Processing Unit-Based Particle-in-Cell Simulations, in: Proceedings of the 10th International Computational Accelerator Physics Conference (ICAP2009), San Francisco, CA, 2009.
[8] J.M. Nageswaran, N. Dutt, J.L. Krichmar, A. Nicolau, A.V. Veidenbaum, A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors, Neural Networks 22 (2009) 791–800.
[9] J. Villasenor, O. Buneman, Rigorous charge conservation for local electromagnetic-field solvers, Computer Physics Communications 69 (1992) 306–316.
[10] O. Buneman, TRISTAN The 3-D Electromagnetic Particle Code, in: H. Matsumoto, Y. Omura (Eds.), Computer Space Plasma Physics: Simulation Techniques and Software, Terra Scientific Publishing Company, Tokyo, 1993, pp. 67–84.
[11] T. Umeda, Y. Omura, T. Tominaga, H. Matsumoto, A new charge conservation method in electromagnetic particle-in-cell simulations, Computer Physics Communications 156 (2003) 73–85.
[12] G. Stantchev, W. Dorland, N. Gumerov, Fast parallel Particle-To-Grid interpolation for plasma PIC simulations on the GPU, Journal of Parallel and Distributed Computing 68 (2008) 1339–1349.
[13] V. Podlozhnyuk, Histogram calculation in CUDA, (2007) URL: <http://www.nvidia.com/object/cuda_sample_data-parallel.html#histogram256>.
[14] N. Satish, M. Harris, M. Garland, Designing Efficient Sorting Algorithms for Manycore GPUs, in: Proceedings of IEEE International Parallel and Distributed Processing Symposium 2009, vol. 1, 2009.