Computer Physics Communications 185 (2014) 708–719

Contents lists available at ScienceDirect

Computer Physics Communications

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

Particle-in-Cell algorithms for emerging computer architectures

Viktor K. Decyk [a,*], Tajendra V. Singh [b]

[a] Department of Physics and Astronomy, University of California, Los Angeles, CA 90095-1547, USA

[b] Institute for Digital Research and Education, University of California, Los Angeles, CA 90095-1547, USA

ARTICLE INFO

Article history:
Received 22 June 2013
Received in revised form
4 October 2013
Accepted 14 October 2013
Available online 22 October 2013

Keywords:
Parallel algorithms
Particle-in-Cell
GPU
CUDA
Plasma simulation

ABSTRACT

We have designed Particle-in-Cell algorithms for emerging architectures. These algorithms share a common approach, using fine-grained tiles, but different implementations depending on the architecture. On the GPU, there were two different implementations, one with atomic operations and one with no data collisions, using CUDA C and Fortran. Speedups up to about 50 compared to a single core of the Intel i7 processor have been achieved. There was also an implementation for traditional multi-core processors using OpenMP which achieved high parallel efficiency. We believe that this approach should work for other emerging designs such as Intel Phi coprocessor from the Intel MIC architecture.

1. Introduction

High Performance Computing (HPC) is going through a revolutionary phase with greatly increased parallelism on shared memory processors, but with different approaches. A number of these approaches make use of Single Instruction Multiple Data (SIMD) vector processors, which introduces another level of parallelism. Graphical Processing Units (GPUs) were an early pioneer in this kind of computing, especially after NVIDIA introduced CUDA, a parallel computing architecture with a new parallel programming model and instruction set architecture which substantially simplified programming their devices [1]. Intel has also introduced a similar architecture with the Phi coprocessor [2], but with a different programming model. The Intel Phi coprocessor is based on the Intel Many Integrated Core (MIC) Architecture.

This paper focuses on how to implement Particle-in-Cell (PIC) codes on such emerging architectures. PIC codes model plasmas by integrating self-consistently the trajectories of many charged particles responding to the electromagnetic fields that they themselves generate [3,4]. They are widely used in plasma physics, and with domain-decomposition techniques [5] they have run in parallel on very large distributed memory supercomputers [6]. PIC codes have three major steps: depositing some quantity such as charge or current from particles onto a grid, solving a field equation to obtain the electromagnetic fields, and interpolating electric and magnetic fields to particles from a grid. They have two data structures, particles and fields, that need to efficiently communicate with one another.

A variety of PIC codes have been implemented on GPU platforms in recent years [7–15]. In an earlier paper [8], we described how we implemented a two dimensional (2D) electrostatic spectral PIC code on the NVIDIA GT200 class of GPUs. The fundamental approach we used was to divide space into small tiles (typically containing 2×3 grid points) and to keep particles in each tile stored together in memory. Each tile could then be processed independently in parallel and only a small amount of fast local memory was needed.

More tiles were used than actual processors. The deposit step is very critical because data collisions are possible when two threads attempt to write to the same memory location simultaneously. Because the GT200 series GPUs (such as the Tesla C1060) are very slow in resolving data collisions for floating point data, the algorithm we designed was collision-free. This was accomplished by assigning a different thread to each tile. Because tiles were very small, the cost of reordering particles moving between tiles was substantial.

* Corresponding author. Tel.: +1 310 206 0371.
E-mail address: decyk@physics.ucla.edu (V.K. Decyk).

0010-4655
http://dx.doi.org/10.1016/j.cpc.2013.10.013

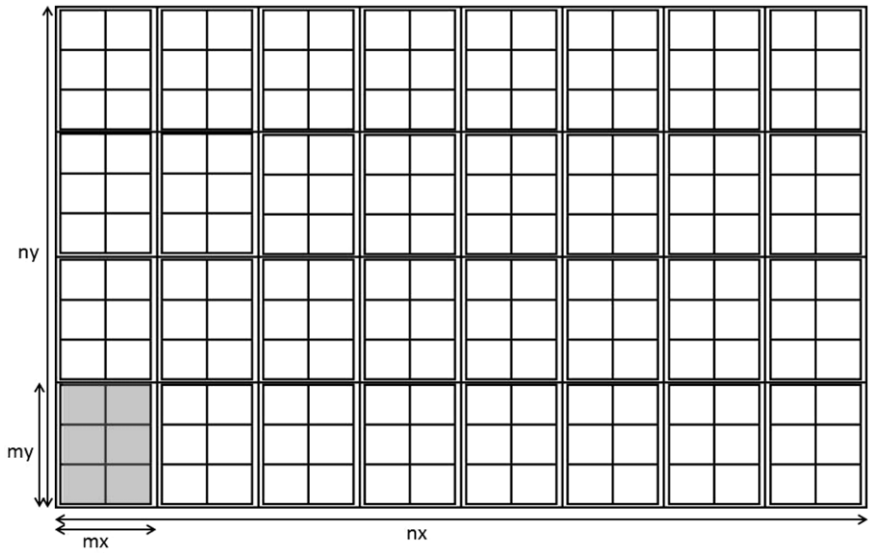

Fig. 1. Partition of grid space into tiles. In this figure, the variables defined in the text have values $\mathtt{nx} = 16$, $\mathtt{ny} = 12$, $\mathtt{mx} = 2$, $\mathtt{my} = 3$, and their sizes are shown. The total number of tiles is 32, with layout $\mathtt{mx1} = 8$, $\mathtt{my1} = 4$. Additional guard cells for the tiles are not shown.

With NVIDIA's introduction of the Fermi architecture, several new features were added that had a substantial impact on optimal PIC algorithms: increased fast local memory, automatic cache, and native support for atomic adds for floating point data. The use of tiles is still optimal, but now it is preferable to assign a block of threads to a tile rather than one thread. We call this algorithm collision-resolving. This paper describes the changes made to the earlier algorithm and the resulting performance of the new algorithm. Furthermore, we show that the general tiling approach can also work well on conventional shared-memory multi-processors and this can be used as the basis for a portable approach to PIC codes on emerging architectures. Both electrostatic and electromagnetic spectral algorithms are described.

2. Collision-free GPU PIC algorithm

An earlier version of this algorithm is described in [8]. Here we summarize the main points and describe some changes. The algorithm is described in 2D space, but can be extended to 3D.

Space is divided into tiles each of which contains mx, my uniformly spaced grid points in x and y respectively, plus additional guard cells. If there are nx, ny grid points in the entire computational region, then there is a total of $\mathtt{mx1} \times \mathtt{my1}$ tiles, where $\mathtt{mx1} = (\mathtt{nx} - 1)/\mathtt{mx} + 1$ and $\mathtt{my1} = (\mathtt{ny} - 1)/\mathtt{my} + 1$. Tiles at the edges can contain grid points which are not active. Fig. 1 describes the partitioning.

To determine which tile a particle belongs to, one first determines the grid point n, m, that the particle belongs to. The tile number is then given by: n/mx and m/my. The same tiling is used in both the collision-free and collision-resolving algorithms.

Both GPUs as well as other high performing processors make use of SIMD processors, which perform nvec vector operations simultaneously (in lockstep). The value of nvec varies from 4 on the Intel SSE to 32 on NVIDIA Fermi devices. These SIMD processors also read data in large blocks, such as 128 bytes on the Fermi device. It is thus advantageous that adjacent memory locations needed by a vector operation are stored in adjacent memory locations. On the GPU, each vector operation is performed by a group of threads (called a thread block in CUDA).

For the collision-free algorithm, each vector element (thread) is responsible for one tile, and the particle data is stored in an array where adjacent threads in a thread block read adjacent memory locations. Such data layout for the particles is described by the following Fortran declaration:

```
real dimension partc(lvect,idimp,nppmx,mbxy1)
```

where lvect is typically the blocksize on the GPU, idimp is the number of particle co-ordinates (2 positions plus 2 or 3 velocities), nppmx is the maximum number of particles expected in a tile, and mbxy1 the number of independent blocks of tiles given by: $(\mathtt{mx1} * \mathtt{my1} - 1)/\mathtt{lvect} + 1$. An auxiliary array kpic is used to store the number of actual particles in each tile in partc. It is declared as follows:

```
integer kpic(mx1*my1)
```

Normally, the code initializes the particle co-ordinates on the host computer in the usual way, then rearranges them by tile into the new data structure and copies them to the GPU.

Advancing particles is now easily and efficiently parallelized. The input to the push procedure is a global field array, which for linear interpolation is declared as follows:

```
real dimension fxye(idimp-2,nx+1,ny+1)
```

where idimp-2 corresponds to the number of velocity co-ordinates (2 for the electrostatic code, 3 for the electromagnetic code). Each thread block copies the fields it requires from global memory into a small local (shared) memory array which can be read much faster than

the global array. For linear interpolation this is declared as follows:

```
real, shared, dimension(idimp-2,mx+1,my+1,lvect) :: sfxy
```

Each thread processes different tiles and different particles independently, and there are no data collisions. Shared memory is not actually shared among threads currently, but it is the fastest memory available in CUDA which allows indirect addressing.

The structure of the deposit step is similar to that of the particle push. For example, the charge density is first accumulated to a local (shared) array, declared as:

```
real, shared, dimension(mx+1,my+1,lvect) :: sq
```

Then it is copied to a tiled global array qs, declared as follows:

```
real, dimension(lvect,mx+1,my+1,mbxy1) :: qs
```

The current deposit is similar. After the deposit, the guard cells in the charge or current array need to be added up. This is done by a separate procedure where each thread is responsible for a different grid point and some threads may read (but not write) data which belong to different tiles. The output goes to a different global array. The output charge density has the declaration:

```
real dimension q(nx,ny)
```

This also avoids any data collisions in the deposit. The push and deposit procedures do not need to know how the tiles are ordered in memory, but the guard cell and the reordering procedures do because they impose periodic boundary conditions. We have stored the tiles as one would a two dimensional array with Fortran ordering, but other orderings are possible. Two features which were described in [8], allowing a thread to process more than one tile and allowing the tile sizes to vary, were not helpful and are not currently supported.

3. Particle reordering with one thread per tile

Implementing the push and deposit procedures on the GPU was relatively straightforward and performed well, as we show below. However, particles need to be reordered at each time step as they move between tiles, and this is a challenge, since reordering is highly irregular and does not fit the GPU architecture well. The procedure we use now is simpler than the procedure described in [8]. It has three steps. In the first step, a list of the indices of the particles leaving a tile and their direction indices (which indicate which of the 8 possible neighboring tiles the particles are going to), are written to an array nhole. The sum of the number of particles going to each of the direction indices is also written to an array ncl. Fig. 2 illustrates which direction corresponds to which index. The arrays are declared as follows:

```
integer dimension nhole(lvect,2,ntmax+1,mbxy1)
integer dimension ncl(8,mx1*my1)
```

where ntmax is the maximum number of particles expected to leave a tile. The total number of particles leaving a tile is also stored in the nhole array. If the number of particles leaving a tile would exceed ntmax, an error is returned. In a production environment, one could reallocate the nhole array upon error and run the procedure again.

In the second step, the particles leaving a tile are copied into an ordered global buffer pbuff, where all the particles going in a particular direction are stored together in memory. This ordering is achieved by first performing a running sum (sometimes called a prefix scan or prefix sum) of ncl for each tile, which gives the offset into the pbuff array where to start copying the outgoing particles going in a particular direction. The declaration of the buffer is as follows:

```
real dimension pbuff(lvect,idimp,ntmax,mbxy1)
```

In the third step, the thread controlling each tile reads the ncl array for each of the 8 directions to determine how many particles it needs from each direction, and where they are located in the pbuff array. (Since ncl contains a running sum, the number for each direction is just the difference between two neighboring directions.) The buffer data is then written into the particle array. First any holes left by departing particles are filled. Any additional particles are written at the end of the particle array. If ip holes remain after all incoming particles are processed, then those holes are filled with the last ip particles from the end of the array. In this way, particles are always stored in a contiguous array with no gaps. The auxiliary array kpic is also updated. During processing, the global array ncl is copied to shared memory in the first and second steps and then written back to global memory. No data collisions occur in any of these steps, since each thread always controls its own write buffers. An OpenMP version of the entire reordering scheme in Fortran77 is listed in the Appendix.

The overall scheme, illustrated in Fig. 2, is similar to that used in message-passing on distributed memory computers [5], except that one ordered buffer is used for sending particles instead of separate buffers for each direction. This simplified reordering scheme was relatively straightforward to implement, compared to the previous scheme.

The first step is the most time-consuming, because all the particles need to be read into memory. The remaining two steps only read the particles which are leaving from or arriving into a tile, which is normally a small subset of the total particles. We do not allow particles to cross more than one tile in a time step, but this is possible by executing the reordering scheme more than once.

In the collision-free algorithm, the first step was incorporated into the particle push. This avoids reading all the particles again just to determine which particles need to be moved. In the electromagnetic (EM) code, the push advances the position only half a time step and the current deposit step advances it the remaining half step. This keeps both current and charge deposits properly time-centered. As a result, two reorderings are performed each time step (but fewer particles need to be moved). As in the push, the first step of the second reordering is incorporated into the current deposit.

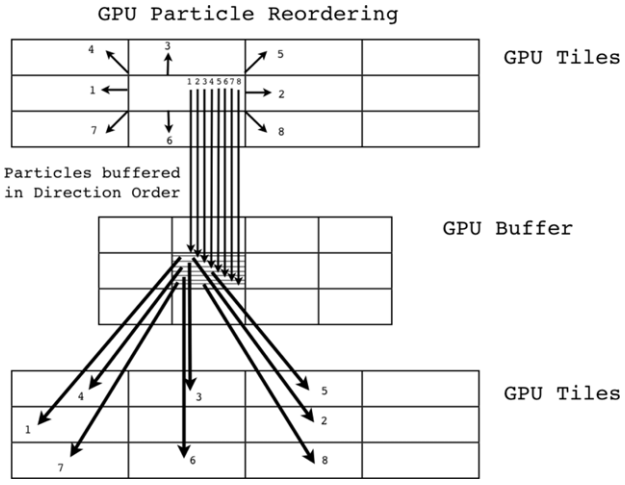

Fig. 2. Particle reordering. Particles leaving a tile are first written into an ordered GPU buffer by one procedure. A separate procedure copies particles coming into a tile from 8 different buffers.

Table 1
Benchmarks of the collision-free algorithm on GT200 GPUs.

```
Electrostatic Code, mx=2,my=3, with dt = 0.1
                Intel i7(1 core)  GPU:S1070
Push               19.9 ns.       0.686 ns.
Deposit             9.0 ns.       0.225 ns.
Reorder             0.3 ns.       0.777 ns.
Total Particle     29.2 ns.       1.688 ns.
Total speedup was about 17.

Electromagnetic Code, mx=1,my=2, with dt = 0.04, c/vth = 10
                 Intel i7(1 core)   GPU:S1070
Push                67.0 ns.        1.562 ns.
Deposit             38.3 ns.        0.980 ns.
Reorder              0.2 ns.        1.191 ns.
Total Particle     105.5 ns.        3.733 ns.
Total speedup was about 28.
```

4. Benchmark results with collision-free algorithm on GT200 architecture GPUs

With this algorithm, we have implemented both a 2D electrostatic (ES) spectral code, as in [8], as well as a 2-1/2D relativistic, EM spectral code, which calculates electric and magnetic fields from the full set of Maxwell's equation, as in [16]. Both codes solve the field equations using Fast Fourier Transforms (FFTs) with periodic boundary conditions, 3 FFTs per time step for the ES code and 10 for the EM code. Spectral EM codes deposit both charge and current density. The GPU routines were implemented in CUDA C, but the main programs and initialization procedures were written in Fortran90 and Fortran77, using gfortran. The benchmark program (Table 1) had a grid of 1024×1024 with 37, 748,730 particles (36 particles per cell) and was run on an NVIDIA Tesla S1070 device, in single precision. A CUDA blocksize = 32 gave the best results. The host machine used Intel i7 CPUs (Dual Intel Xeon—X5650, each with six cores), running at a clock speed of 2.67 GHz. Time reported is per particle per time step. The GPU field solver time is not shown, but it made use of the CUDA FFT library and was about 6% of the total time for the ES case, and 11% in the EM case. A plasma in thermal equilibrium was simulated for 100 steps in the ES case and 250 steps in the EM case.

About 30%–40% of the total time was spent in reordering particles. The average number of particles leaving a tile per time step was 6.5% for the ES case and 1.5% for the EM case. The major limitation of this algorithm is that each thread needs its own copy of the shared memory arrays, and thus the amount of shared memory needed by a thread block is large. Since shared memory is a scarce resource, small tile sizes are required and this leads to a relatively large percentage of particles moving between tiles each time step. The optimal tile size for the EM case was smaller because the shared memory requirements were larger.

5. Collision-resolving GPU PIC algorithm

The introduction of the NVIDIA Fermi architecture added several new features that had a substantial impact on optimal PIC algorithms, the most important being hardware support for atomic adds for floating point data. (Atomic adds treat the add as an uninterruptible operation, without interference between threads.) If atomic operations work well, then it is possible to assign a block of threads to a tile, and the total amount of shared memory per block is greatly reduced, allowing larger tile sizes. Atomic operations are needed in the deposit step if multiple threads are used in a tile because two threads might be attempting to update the same grid point at the same time. In order for adjacent threads to access adjacent memory locations, a new data structure is needed for particles, declared as:

```
real dimension ppart(nppmx,idimp,mx1*my1)
```

Implementing the particle advance with one thread block per tile was relatively easy. Instead of a loop over particles as we had in the collision-free algorithm:

```
do j = 1, npp
...
enddo
```

where npp is the number of particles in a tile, one uses the following construct in PGI CUDA Fortran:

```
j = threadIdx%x
do while (j <= npp)
   ...
   j = j + blockDim%x
enddo
```

with similar expressions in CUDA C. When copying the field arrays to shared memory, each thread copies part of the data and the shared memory is actually shared. The declarations for the shared arrays are now:

```
real, shared, dimension(idimp-2,mx+1,my+1) :: sfxy
real, shared, dimension(mx+1,my+1) :: sq
```

Implementing the deposit procedure is similar. The major additional change is the use of the atomicAdd function. Instead of operations such as:

```
sq(n) = sq(n) + a,
```

one writes:

```
old = atomicAdd(sq(n),a)
```

where old will contain the old value of sq(n). In CUDA C the expression is similar. When the particles have been processed, the charge density can be copied to a tiled global array qp, declared as follows:

```
real dimension qp(mx+1,my+1,mxy1)
```

Then guard cells can be added and the result written to the global array q, as in the collision-free case. Alternatively, the charge density and guard cells can be added in the deposit step directly to the global array q using atomicAdds. Since most of the time in the deposit is spent in accumulating the charge density in the shared memory, and not writing to global memory, it does not make much difference which deposit method is chosen. Current deposit is handled the same way.

6. Particle reordering with multiple threads per tile

To implement the particle reordering scheme shown in the Appendix using a vector or group of threads per tile, on the other hand, is much more challenging to implement than with one thread per tile. It requires a different data structure for the array describing the particles leaving a tile, as follows:

```
integer dimension ihole(2,ntmax+1,mxy1)
```

To avoid complex if tests (causing warp divergence in CUDA), data parallel algorithms using masking operations and prefix scans are effective here. As an example of how a masking operation is used, consider the first step of the reordering, where ist represents the direction a particle is going (0 <= ist <= 8), and j is its location in the particle array. The OpenMP version has the following construct:

```
if (ist.gt.0) then
    ncl(ist,k) = ncl(ist,k) + 1
    ih = ih + 1
    ihole(1,ih+1,k) = j
    ihole(2,ih+1,k) = ist
endif
```

When multiple threads, each responsible for a different particle, are executing this in parallel, one can use atomicAdds to sum ncl. However, the remaining steps of the reordering are much simpler if the particle locations j stored in the array ihole at location ih are monotonically increasing. In that case, we cannot use atomicAdds to increment ih because different threads will place the values of j in unpredictable locations ih. Instead we use data parallel programming techniques [17,18], as shown in the following pseudo code, where mask is an integer array, and i refers to the thread index:

```
mask(i) = 0
if (ist > 0) mask(i) = 1
prefixscan(mask)
ih = mask(i)
ip = 0
if (i > 1) ip = mask(i-1)
if (ih > ip) then
   ihole(1,ih+1,k) = j
   ihole(2,ih+1,k) = ist
endif
```

One sets mask to 1 if ist > 0, otherwise 0. After the prefix scan (running sum), the mask(i) array value for thread i will jump by one compared to the previous value for the thread where ist > 0, and the value of mask for that thread is the required ih value. For example, suppose we have a vector length (thread block) of 6, and threads 2 and 5 have ist > 0. In that case, mask = [0 1 0 0 1 0]. The prefix sum gives mask = [0 1 1 1 2 2]. For thread 2, we have ih = mask(2) = 1, and for thread 5, we have ih = mask(5) = 2. For threads where ist = 0 (which means the particle is not moving to another tile), mask(i) = mask(i − 1) and ih = mask(i) is not used.

In the third step of the reordering, the code (shown in the Appendix) has a double loop over directions and the number of particles in each direction.

```
do ii = 1, 8
ip = ...
do j = 1, ip
...
```

where the index ii refers to the number of directions, and ip is the number of incoming particles from that direction. Multiple (segmented) prefix scans are used to find the addresses in the pbuff array for incoming particles in order to read from all directions simultaneously. Prefix scans are also used to load the addresses to determine how to fill any leftover holes if there are more holes than incoming particles.

One could implement the particle reordering scheme more simply using one thread per tile as before. However, it would be less efficient, because the particle data structure ppart used with one thread block per tile for the push and deposit, is not optimal when using one thread per tile for the reordering, making that scheme less efficient here.

7. Field solvers

Both the collision-free and collision-resolving algorithms use the same field solvers. The 2D ES code solves Poisson's equation in Fourier space as described in [8]. At the time that paper was written, NVIDIA's real to complex (R2C) FFT was not optimized and performed poorly. The R2C transforms now perform well and as a result, we use an algorithm which uses multiple 1D R2C transforms in x, followed by a transpose, then multiple 1D complex to complex (C2C) transforms in y. This performed better than using NVIDIA's 2D R2C transform and is also more compatible with a future MPI version. The ES code first transforms the charge density, calculates a 2 component electric field in Fourier space, and then performs a 2 component FFT, where both components are transformed at the same time. For the 2-1/2D EM code, the particles have three velocity co-ordinates and 2 spatial co-ordinates. The field solver separates the vector quantities into longitudinal (curl-free) and transverse (divergence-free) quantities. The Poisson equation is solved as in the ES code. In addition, Maxwell's equations for the transverse quantities are updated. The transverse quantities for current, electric and magnetic fields each have 3 components and 3 component FFTs are performed. The ES and EM electric fields are combined in Fourier space before transforming to real space. The outputs of the FFT are global arrays, such as the following for the electric field:

```
real dimension fxy(nx,idimp-2,ny)
```

This array is then copied, with guard cells added, to the array fxye, described in Section 2. The magnetic field is treated similarly.

8. Porting the GPU algorithms to multi-core processors

During the development of the GPU codes, it was helpful to develop serial versions of all the subroutines as an aid in debugging. The procedure shown in the Appendix was an example. After this was done, we realized that the loops over tiles could be easily parallelized with OpenMP, and we could obtain a library of procedures which could run in parallel on current generation multi-core processors. Since these procedures had the same dummy arguments as the GPU procedures, it was possible to use the same main program with either multi-core processors or GPUs and just link with different libraries. With the OpenMP version only the loop over tiles was parallelized, so there were never any collisions within a tile for either the collision-free or collision-resolving algorithm. We implemented the collision-resolving algorithm because it was simpler. The only change was that the particle array with OpenMP was transposed relative to the GPU version, as follows:

```
real, dimension ppart(idimp,nppmx,mx1*my1)
```

This gave better cache performance on traditional architectures.

9. Benchmark results with collision-resolving algorithm on Fermi architecture GPUs

We ran the collision-resolving algorithm on the NVIDIA Fermi M2090 (Table 2). For the ES case and the OpenMP cases it was beneficial to move step one of the reordering scheme (calculating ihole and ncl) into the push procedure (and into the current deposit for the EM code). However, for the EM code on the GPU, it was better to keep all three steps in the reordering scheme. This was due to the increased use of registers and shared memory when extra calculations were included, which reduced the number of independent blocks which can run simultaneously (occupancy). The Fermi GPU has fewer registers than the GT200. Due to its larger memory, we increased the benchmark size to a larger grid of 2048 × 2048 with 150,994,944 particles (36 particles per cell). The implementation reported here used Cuda C with a Fortran90 main program. We used single precision, and a CUDA blocksize = 128 gave the best results. Shared Memory was set to 48 kB. The GPU field solver time is not shown, but was about 6% of the total time for the ES case, and 10% in the EM case.

Note that the OpenMP code result has better than 95% parallel efficiency. On the GPU, about 15%–30% of the time was spent in reordering particles. The average number of particles leaving a tile per time step was 1.0% for the ES case and 0.3% for the EM case. The performance on the M2090 was nearly twice that of the S1070. For the ES case, the reordering time using OpenMP with 12 cores is similar to the time with the GPU. For the EM case, reordering with OpenMP appears much faster than on the GPU, but this is deceptive, because in this case the GPU did not incorporate the first part of the reordering into the push. One important advantage of the OpenMP version of the code

Table 2
Benchmarks of the collision-resolving algorithm on Fermi GPUs.

```
Electrostatic Code, mx=16,my=16, with dt = 0.1
              Intel i7(1 core)  GPU:M2090   OpenMP(12 cores)
Push             22.1 ns.       0.532 ns.      1.678 ns.
Deposit           8.5 ns.       0.227 ns.      0.818 ns.
Reorder           0.4 ns.       0.115 ns.      0.113 ns.
Total Particle   31.0 ns.       0.874 ns.      2.608 ns.
Total speedup was about 35 compared to 1 core, and about 3
compared to 12 cores.

Electromagnetic Code, mx=16,my=16, with dt = 0.04, c/vth = 10
               Intel i7(1 core)   GPU:M2090  OpenMP(12 cores)
Push              66.5 ns.        0.426 ns.     5.645 ns.
Deposit           36.7 ns.        0.918 ns.     3.362 ns.
Reorder            0.4 ns.        0.698 ns.     0.056 ns.
Total Particle   103.6 ns.        2.042 ns.     9.062 ns.
Total speedup was about 51 compared to 1 core, and about 4
compared to 12 cores.
```

is that particles are now ordered, so that additional interactions (such as particle collisions) which need to know the location of nearby particles are much easier and less costly to implement.

The number of floating point operations per particle per time step for the combined push and deposit steps is about 60 for the ES code and 175 for the EM code. The time to complete those two steps is 0.76 ns and 1.34 ns, respectively. This gives a floating point performance of 79 GFlops and 131 GFlops, respectively, where GFlops means one billion floating point operations per second. Since the peak floating point speed of the M2090 is 1.33 TFlops (one trillion floating point operations per second), the ES code is achieving about 6% of peak floating point speed, and the EM code 10% of peak floating point speed.

Such a figure is misleading, however, because these codes are memory bound, with low computational intensity. A better measure of computational efficiency is to include the memory access time. The ES code accesses 40 bytes of global memory per particle per time step, while the EM code accesses 76 bytes. The peak memory speed for global memory access is 142 GB/s for memory with error correction (ECC) turned on, as in our case. The minimum time to just access memory is then 0.28 ns per particle per time step for the ES code and 0.54 ns for the EM code, neglecting latency, accessing field quantities and load imbalance. In comparison, if the device were operating at peak speed (1.33 TFlops), the minimum time to perform the floating point operations would be 0.05 ns per particle per time step for the ES code and 0.13 ns for the EM code. The memory access time will likely partially overlap with floating point calculation time. If we define the ideal time to be the largest of these times (which assumes complete overlap), then we can define the lower bound of the computational efficiency to be the ratio of this ideal time to the actual time. We then obtain a computational efficiency of 37% for the ES code and 40% for the EM code for these two steps. The particle reordering has very flew floating point operations and its memory access is irregular and inefficient. Fortunately, in most cases few particles need to be moved between tiles so this inefficiency does not dominate.

A CUDA Fortran version of the collision-resolving algorithm was also implemented and the results were about 10% slower than with native CUDA C.

10. Discussion

It is possible to run the collision-free algorithm on Fermi architectures. Results show that the collision-resolving algorithm is about twice as fast as the collision-free algorithm. On the GT200 architectures, it is not possible to run the collision-resolving algorithm as written because floating point atomic updates are not supported. However, if we use the less efficient atomic CAS (Compare and Swap) operations to implement atomic updates, then the collision-free algorithm is about 10% faster than the collision-resolving algorithm on the GT200.

The optimal tile size was determined empirically. As the tile size increases, fewer particles need to be reordered, and the reordering time decreases. However, the push and deposit run more slowly since more shared memory is required and therefore the number of independent blocks which can run simultaneously (occupancy) goes down.

The simple skeleton codes described here use linear interpolation. The use of higher order interpolation requires more shared memory, which tends to slow the code down. However, the computational intensity (more operations per memory access) is higher, so overall one expects the speedups to be higher.

The GPU hardware consists of a collection of vector processors with small amounts of fast memory, coupled by slow global memory. So it is not surprising that programming GPUs makes use of algorithms familiar to scientific programmers from past architectures. In these PIC codes we have made use of vector algorithms similar to those used on the Cray for particle pushing and field solvers. We made use of data parallel algorithms similar to those used on the Connection Machine [17,19] in the reordering scheme. Blocking techniques were used in the overall tiling design as well as in other procedures such as the data transpose used in the FFT. Message-passing algorithms were also used in the reordering to send and receive particle buffers.

One new element, however, is that programming GPUs favors designs where there are many more threads than physical processors, due to the extremely fast thread switching in the hardware. This helps to hide memory latency. We can also use this feature to assign work to threads for load-balancing, either manually or automatically with the master–slave paradigm. Originally, it was thought that very fine-grained parallelism would be very difficult or impossible to load-balance, since the number of particles in small regions would fluctuate a great deal. But using a large number of threads can avoid this problem.

One would think that with the introduction of cache memory in the Fermi architecture, shared memory might be unnecessary and redundant. However, shared memory is useful because the data there cannot be evicted by the hardware, whereas data in cache could be.

We have determined that fine-grain tiling with reordering every time step is an efficient approach to emerging architectures. We have written two versions, one collision-free for architectures where atomic updates are inefficient, and another collision-resolving where atomic updates work well. We have also implemented a multi-core version which works with high parallel efficiency on current multi-core architectures. One of our goals is to develop a portable approach to PIC codes on other emerging architectures, such as the Intel PHI coprocessor. The Intel PHI has a similar underlying hardware architecture consisting of multiple SIMD processors, but the programming model is different. The authors of [2] claim (p. 16) that applications which work well on GPUs should work well on the PHI, and that all one needs is OpenMP with some efficient vectorization method. We do not expect that one source code will be possible in the near term, because expressing parallelism is still evolving. However, creating different implementations that utilize the same algorithms and similar data structures looks feasible now. Modifying our procedures to support multiple GPUs with MPI also appears to be relatively straightforward. This will be the subject of a future paper.

It is our intention to make compact skeleton codes for various parallel architectures available on the UCLA IDRE web site at: https://idre.ucla.edu/hpc/parallel-plasma-pic-codes. Some are available now.

Acknowledgments

This work was supported by the US Department of Energy under grant DE-SC0008491, the National Science Foundation under grants NSF PHY-0960344 and OCI-1036224, NASA under grant NNX10AK98G, and the UCLA Institute for Digital Research and Education (IDRE).

Appendix. OpenMP particle reordering

```
      subroutine PPORDER2L(ppart,ppbuff,kpic,ncl,ihole,idimp,nppmx,nx,
     1ny,mx,my,mx1,my1,npbmx,ntmax,irc)
c this subroutine sorts particles by x,y grid in tiles of mx, my
c linear interpolation, with periodic boundary conditions
c tiles are assumed to be arranged in 2D linear memory
c algorithm has 3 steps. first, one finds particles leaving tile and
c stores their number in each direction, location, and destination
c in ncl and ihole. second, a prefix scan of ncl is performed and
c departing particles are buffered in ppbuff in direction order.
c finally, we copy the incoming particles from other tiles into ppart.
c input: all except ppbuff, ncl, ihole, irc
c output: ppart, ppbuff, kpic, ncl, ihole, irc
c ppart(1,n,k) = position x of particle n in tile k
c ppart(2,n,k) = position y of particle n in tile k
c ppbuff(i,n,k) = i co-ordinate of particle n in tile k
c kpic(k) = number of particles in tile k
c ncl(i,k) = number of particles going to destination i, tile k
c ihole(1,:,k) = location of hole in array left by departing particle
c ihole(2,:,k) = direction destination of particle leaving hole
c all for tile k
c ihole(1,1,k) = ih, number of holes left (error, if negative)
c idimp = size of phase space = 4
c nppmx = maximum number of particles in tile
c nx/ny = system length in x/y direction
c mx/my = number of grids in sorting cell in x/y
c mx1 = (system length in x direction - 1)/mx + 1
c my1 = (system length in y direction - 1)/my + 1
c npbmx = size of buffer array ppbuff
c ntmax = size of hole array for particles leaving tiles
c irc = maximum overflow, returned only if error occurs, when irc > 0
c written by viktor k. decyk, ucla
c copyright 2012, regents of the university of california
      implicit none
      integer idimp, nppmx, nx, ny, mx, my, mx1, my1, npbmx, ntmax
      integer irc
      real ppart, ppbuff
      integer kpic, ncl, ihole
      dimension ppart(idimp,nppmx,mx1*my1)
      dimension ppbuff(idimp,npbmx,mx1*my1)
      dimension kpic(mx1*my1), ncl(8,mx1*my1)
      dimension ihole(2,ntmax+1,mx1*my1)
c local data
      integer mxy1, noff, moff, npp, ncoff
      integer i, j, k, ii, kx, ky, ih, nh, ist, nn, mm, isum
      integer ip, j1, j2, kxl, kxr, kk, kl, kr
```

```
      real anx, any, edgelx, edgely, edgerx, edgery, dx, dy
      integer ks
      dimension ks(8)
      mxy1 = mx1*my1
      anx = real(nx)
      any = real(ny)
c find and count particles leaving tiles and determine destination
c update ppart, ihole, ncl
c loop over tiles
!$OMP PARALLEL DO
!$OMP& PRIVATE(j,k,noff,moff,npp,nn,mm,ih,nh,ist,dx,dy,edgelx,edgely,
!$OMP& edgerx,edgery)
      do 30 k = 1, mxy1
      noff = (k - 1)/mx1
      moff = my*noff
      noff = mx*(k - mx1*noff - 1)
      npp = kpic(k)
      nn = min(mx,nx-noff)
      mm = min(my,ny-moff)
      ih = 0
      nh = 0
      edgelx = noff
      edgerx = noff + nn
      edgely = moff
      edgery = moff + mm
c clear counters
      do 10 j = 1, 8
      ncl(j,k) = 0
   10 continue
c loop over particles in tile
      do 20 j = 1, npp
      dx = ppart(1,j,k)
      dy = ppart(2,j,k)
c find particles going out of bounds
      ist = 0
c count how many particles are going in each direction in ncl
c save their address and destination in ihole
c use periodic boundary conditions and check for roundoff error
c ist = direction particle is going
      if (dx.ge.edgerx) then
         if (dx.ge.anx) ppart(1,j,k) = dx - anx
         ist = 2
      else if (dx.lt.edgelx) then
         if (dx.lt.0.0) then
            dx = dx + anx
            if (dx.lt.anx) then
               ist = 1
            else
               dx = 0.0
            endif
            ppart(1,j,k) = dx
         else
            ist = 1
         endif
      endif
      if (dy.ge.edgery) then
         if (dy.ge.any) ppart(2,j,k) = dy - any
         ist = ist + 6
      else if (dy.lt.edgely) then
         if (dy.lt.0.0) then
            dy = dy + any
            if (dy.lt.any) then
               ist = ist + 3
            else
               dy = 0.0
            endif
```

```
            ppart(2,j,k) = dy
         else
            ist = ist + 3
         endif
      endif
      if (ist.gt.0) then
         ncl(ist,k) = ncl(ist,k) + 1
         ih = ih + 1
         if (ih.le.ntmax) then
            ihole(1,ih+1,k) = j
            ihole(2,ih+1,k) = ist
         else
            nh = 1
         endif
      endif
   20 continue
c set error and end of file flag
      if (nh.gt.0) then
         irc = ih
         ih = -ih
      endif
      ihole(1,1,k) = ih
   30 continue
!$OMP END PARALLEL DO
c ihole overflow
      if (irc.gt.0) return
c
c buffer particles that are leaving tile: update ppbuff, ncl
c loop over tiles
!$OMP PARALLEL DO
!$OMP& PRIVATE(i,j,k,isum,ist,nh,ip,j1,ii)
      do 70 k = 1, mxy1
c find address offset for ordered ppbuff array
      isum = 0
      do 40 j = 1, 8
      ist = ncl(j,k)
      ncl(j,k) = isum
      isum = isum + ist
   40 continue
      nh = ihole(1,1,k)
      ip = 0
c loop over particles leaving tile
      do 60 j = 1, nh
c buffer particles that are leaving tile, in direction order
      j1 = ihole(1,j+1,k)
      ist = ihole(2,j+1,k)
      ii = ncl(ist,k) + 1
      if (ii.le.npbmx) then
         do 50 i = 1, idimp
         ppbuff(i,ii,k) = ppart(i,j1,k)
   50    continue
      else
         ip = 1
      endif
      ncl(ist,k) = ii
   60 continue
c set error
      if (ip.gt.0) irc = ncl(8,k)
   70 continue
!$OMP END PARALLEL DO
c ppbuff overflow
      if (irc.gt.0) return
c
c copy incoming particles from buffer into ppart: update ppart, kpic
c loop over tiles
```

```
!$OMP PARALLEL DO
!$OMP& PRIVATE(i,j,k,ii,kk,npp,kx,ky,kl,kr,kxl,kxr,ih,nh,ncoff,ist,j1,
!$OMP& j2,ip,ks)
      do 130 k = 1, mxy1
      npp = kpic(k)
      ky = (k - 1)/mx1 + 1
c loop over tiles in y, assume periodic boundary conditions
      kk = (ky - 1)*mx1
c find tile above
      kl = ky - 1
      if (kl.lt.1) kl = kl + my1
      kl = (kl - 1)*mx1
c find tile below
      kr = ky + 1
      if (kr.gt.my1) kr = kr - my1
      kr = (kr - 1)*mx1
c loop over tiles in x, assume periodic boundary conditions
      kx = k - (ky - 1)*mx1
      kxl = kx - 1
      if (kxl.lt.1) kxl = kxl + mx1
      kxr = kx + 1
      if (kxr.gt.mx1) kxr = kxr - mx1
c find tile number for different directions
      ks(1) = kxr + kk
      ks(2) = kxl + kk
      ks(3) = kx + kr
      ks(4) = kxr + kr
      ks(5) = kxl + kr
      ks(6) = kx + kl
      ks(7) = kxr + kl
      ks(8) = kxl + kl
c loop over directions
      nh = ihole(1,1,k)
      ncoff = 0
      ih = 0
      ist = 0
      j1 = 0
      do 100 ii = 1, 8
      if (ii.gt.1) ncoff = ncl(ii-1,ks(ii))
c ip = number of particles coming from direction ii
      ip = ncl(ii,ks(ii)) - ncoff
      do 90 j = 1, ip
      ih = ih + 1
c insert incoming particles into holes
      if (ih.le.nh) then
         j1 = ihole(1,ih+1,k)
c place overflow at end of array
      else
         j1 = npp + 1
         npp = j1
      endif
      if (j1.le.nppmx) then
         do 80 i = 1, idimp
         ppart(i,j1,k) = ppbuff(i,j+ncoff,ks(ii))
   80    continue
      else
         ist = 1
      endif
   90 continue
  100 continue
c set error
      if (ist.gt.0) irc = j1
c fill up remaining holes in particle array with particles from bottom
      if (ih.lt.nh) then
         ip = nh - ih
```

```
         do 120 j = 1, ip
         j1 = npp - j + 1
         j2 = ihole(1,nh-j+2,k)
         if (j1.gt.j2) then
c move particle only if it is below current hole
            do 110 i = 1, idimp
            ppart(i,j2,k) = ppart(i,j1,k)
  110       continue
         endif
  120    continue
         npp = npp - ip
      endif
      kpic(k) = npp
  130 continue
!$OMP END PARALLEL DO
       return
       end
```

References

[1] Shane Cook, CUDA Programming: A Developer's Guide to Parallel Computing with GPUs, Morgan Kaufmann, 2012.
[2] James Jeffers, James Reinders, Intel Xeon Phi Coprocessor High Performance Programming, Morgan Kaufmann, 2013.
[3] Charles K. Birdsall, A. Bruce Langdon, Plasma Physics via Computer Simulation, McGraw-Hill, New York, 1985.
[4] Roger W. Hockney, James W. Eastwood, Computer Simulation Using Particles, McGraw-Hill, New York, 1981.
[5] P.C. Liewer, V.K. Decyk, A general concurrent algorithm for plasma particle-in-cell codes, J. Comput. Phys. 85 (1989) 302.
[6] B.J. Winjum, J.E. Fahlen, F.S. Tsung, W.B. Mori, Anomalously hot electrons due to rescatter of stimulated Raman scattering in the kinetic regime, Phys. Rev. Lett. 110 (2013) 165001.
[7] H. Burau, R. Widera, W. Honig, G. Juckeland, A. Debus, T. Kluge, U. Schramm, T.E. Cowan, R. Sauerbrey, M. Bussman, PIConGPU: a fully relativistic particle-in-cell code for a GPU cluster, IEEE Trans. Plasma Sci. 38 (2010) 2831.
[8] Viktor K. Decyk, Tajendra V. Singh, Adaptable particle-in-cell algorithms for graphical processsing units, Comput. Phys. Comm. 182 (2011) 641.
[9] Xianglong Kong, Michael C. Huang, Chuang Ren, Viktor K. Decyk, Particle-in-cell simulations with charge-conserving current deposition on graphical processsing units, J. Comput. Phys. 230 (2011) 1676.
[10] P. Abreu, R. Fonseca, J.M. Pereira, L.O. Silva, PIC codes in new processors: a fully relativistic PIC code in CUDA enabled hardware with direct visualization, IEEE Trans. Plasma Sci. 39 (2011) 675.
[11] R.G. Joseph, G. Ravunnikutty, S. Ranka, E. D'Azevedo, S. Klasky, Efficient GPU implementation for particle in cell algorithm, in: 2011 IEEE International, Intl. Parallel and Distributed Symposium (IPDPS), Anchorage, AK, May 2011.
[12] K. Madduri, Eun-Jin Im, K.Z. Ibrahim, S. Williams, S. Ethier, L. Oliker, Gyrokinetic particle-in-cell optimization on emerging multi-and manycore platforms, Parallel Comput. 37 (2011) 501.
[13] G. Chen, L. Chacon, D.C. Barnes, An efficient mixed-precision, hybrid CPU–GPU implementation of a non-linearly implicit one-dimensional particle-in-cell algorithm, J. Comput. Phys. 231 (2012) 5374.
[14] J. Claustre, B. Chaudhury, G. Fubiani, M. Paulin, J.P. Boeuf, Particle-in-cell Monte Carlo collision model on GPU—application to a low-temperature magnetized plasma, IEEE Trans. Plasma Sci. 41 (2013) 391.
[15] K. Germaschewski, H. Ruhl, W. Fox, A. Bhattacharjee, Dynamic load-balancing and GPU computing with the particle-in-cell code PSC, in: 22nd International Conference on Numerical Simulation of Plasmas ICNSP2001, Long Branch, NJ, September 2011.
[16] J.M. Dawson, Particle simulation of plasmas, Rev. Modern Phys. 55 (1983) 403.
[17] Guy-Rene Perrin, Alain Darte (Eds.), The Data Parallel Programming Model, in: Lecture Notes in Computer Science, vol. 1132, Springer, Berlin, 1996.
[18] Nicholas Wild, The CUDA Handbook: A Comprehensive Guide to GPU Programming, Addison-Wesley, 2013.
[19] W. Daniel Hillis, Lewis W. Tucker, The CM-5 connection machine: a scalable supercomputer, Commun. ACM 36 (1993) 31.