Adaptable Particle-in-Cell Algorithms for Graphical Processing Units

Viktor K. Decyk[1,2] and Tajendra V. Singh[2]

[1]Department of Physics and Astronomy
[2]Institute for Digital Research and Education
University of California, Los Angeles
Los Angeles, CA 90095

Abstract

We developed new parameterized Particle-in-Cell algorithms and data structures for emerging multi-core and many-core architectures. Four parameters allow tuning of this PIC code to different hardware configurations. Particles are kept ordered at each time step. The first application of these algorithms is to NVIDIA Graphical Processing Units, where speedups of about 15-25 compared to an Intel Nehalem processor were obtained for a simple 2D electrostatic code. Electromagnetic codes are expected to get higher speedups due to their greater computational intensity.

Introduction

Computer architectures are rapidly evolving to include more and more processor cores. The designs of these new architectures also vary considerably. These include incremental evolutions of existing architectures, heterogeneous mixtures of processors such as the IBM Cell or AMD APU, and accelerators, such as NVIDIA's Graphical Processing Units (GPUs). As the most advanced computers move from petaflops to exaflops in the next decade, it is likely that they will consist of complex nodes with such multi-core and many-core processors. This variety results in great challenges for application programmers. Not only is parallel computing challenging in its own right, there is also a growing variety of programming models evolving, as older programming models prove inadequate and new ones are proposed. These are disruptive times.

Application developers resist developing different codes for different architectures. To cope with this bewildering variety, they need to find some common features that they can use to parameterize their application algorithms to adapt to different architectures. In our opinion, the common feature that we can exploit in future advanced architectures is that processors will be organized in a hierarchy. At the lowest level, they will consist of a number of tightly coupled SIMD cores, which are all executing the same instruction in lock step and sharing fast memory. At the next level, will be loosely coupled groups of SIMD cores that share a slower memory. Finally, there will be accumulations of groups that share no memory and communicate with message-passing. Furthermore, accessing global memory may be even more of a bottleneck than it is now, since memory speeds will not keep up with processor counts, so that lightweight threads to hide memory latency will be necessary. We think that a cluster of GPUs is the closest currently existing architecture to this future and therefore a good choice for experimentation and development.

One of the important applications in plasma physics are Particle-in-Cell (PIC) codes. These codes model plasmas at the microscopic level by self-consistently integrating the trajectories of charged particles with fields that the particles themselves produce. PIC applications are very compute intensive, as the number of particles in these codes can vary from many thousands to many billions. Such codes are used in many different areas of plasma physics, including space plasma physics, advanced accelerators, and fusion energy research. The US Department of Energy allocated over 200 million nodes hours on their most advanced computers to PIC codes in their INCITE program in 2010, about 12% of the total. This paper will report on new algorithms that we have developed for PIC codes that work well on GPUs, and that we feel are adaptable to other future architectures as they evolve. This field is very new and a variety of approaches to developing PIC codes on GPUs are under development [1-5].

Graphical Processing Units

GPUs consist of a number of SIMD multi-processors (30 on the Tesla C1070). Although each SIMD multi-processor has 8 cores, the hardware executes 32 threads together in what NVIDA calls a warp. Adjacent threads are organized into blocks, which are normally between 32 and 512 threads in size. Blocks can share a small (16 KByte), fast memory, which has a latency of a few clock cycles, and have fast barrier synchronization. The GPU has a global memory (4 GBytes on the C1070) accessible by any thread. High memory bandwidth is possible, but is achieved typically when 16 adjacent threads read memory within the same 64 byte block, which NVIDIA calls data coalescing. It is usually achieved by having adjacent threads read memory with stride 1 access. Memory has substantial latency (hundreds of clock cycles), which is hidden by lightweight threads that can be swapped out in one clock cycle. Thousands of threads can be supported simultaneously and millions can be outstanding. Fine grain parallelism is needed to make efficient use of the hardware. Because access to global memory is usually the bottleneck, streaming algorithms are optimal, where global memory is read only once.

NVIDIA has developed a programming model called CUDA for this device, which is based on C, with extensions. OpenCL is also supported for code which will also be run on other devices, such as Intel SSE. In addition, the Portland Group has developed a Fortran compiler for GPUs.

Streaming Algorithm for PIC

Particle-in-Cell codes are a type of N-body code (where all particles are interacting with all the others), but they differ from molecular dynamic (MD) codes. In MD codes where particles interact directly with each other, the calculation is of order N^2. In PIC codes, particles interact via the electric and magnetic fields that they produce. This makes the calculation of order N, and many more particles can be used than in typical MD, up to a trillion currently [6].

A PIC code has three major steps in the inner loop. In the first step, a charge or current density is deposited on a grid. This involves an inverse interpolation (scatter operation) from the particle position to the nearest grid points. The second step involves solving a differential equation (Maxwell's equations or a subset) to obtain the electric and magnetic fields on the grid

from the current or charge density. Finally, the particle acceleration is obtained using Newton's law and the particle positions are updated. This involves an interpolation (gather operation) to obtain the fields at the particle position from the fields on the nearest grid points. Thus PIC codes have two data structures, particles and fields, that need to communicate with one another. Usually, most of the time is spent in steps one and three, and most of the CPU time is spent in only a few subroutines. Textbooks are available which describe such codes [7-8].

PIC codes have been parallelized for distributed memory computers for many years [9], and they have obtained good scaling with up to 300,000 processors for large problem sizes. The parallelization is usually achieved by coarse grain domain decomposition, keeping particles and the fields they need on the same node.

To achieve a streaming algorithm for PIC requires that particles and fields each be read and written only once. For particles this is the usual case. However, for fields this is not so, since there are many particles per cell and different particles are at different locations in space and read different fields. The only way to achieve a streaming algorithm is to keep particles ordered, so that all the particles which would interpolate to the same grid points are stored together. Then the fields the particles need can be read only once and saved in registers or a small local array. Fine grain parallelism can thus be implemented. We will illustrate this with a 2D electrostatic code, which uses only Poisson's equation to obtain an electric field, and is derived from one of the codes in the UPIC framework [10].

Parallel Charge Deposit

We will begin with the first step of the PIC code, the charge deposit. The original Fortran listing of this procedure with bi-linear interpolation in 2 dimensions (2D) is shown below.

```
dimension part(4,nop), q(nx+1,ny+1) ! nop = number of particles
                                    ! nx, ny = number of grid points
do j = 1, nop
   n = part(1,j)                         ! extract x grid point
   m = part(2,j)                         ! extract y grid point
   dxp = qm*(part(1,j) - real(n))        ! find weights
   dyp = part(2,j) - real(m)
   n = n + 1;  m = m + 1                 ! add 1 for Fortran
   amx = qm - dxp
   amy = 1.0 - dyp
   q(n+1,m+1) = q(n+1,m+1) + dxp*dyp     ! deposit
   q(n,m+1) = q(n,m+1) + amx*dyp
   q(n+1,m) = q(n+1,m) + dxp*amy
   q(n,m) = q(n,m) + amx*amy
enddo
```

In this code, the charge on the particle is split into 4 parts, which are then deposited to the 4 nearest grid points. A particle spatial co-ordinate consists of an integer part which contains the lower grid point, and the deviation from the point. The algorithm first separates the integer part and the deviation from the particle spatial co-ordinate. The integer part is used to address the grid points, and the amount deposited is proportional to how close the particle is to that grid.

Since particles are normally not ordered, each particle will deposit to a different location in memory.

In the new adaptable streaming algorithm, we need a new data structure. Since particles can be processed in any order, we can partition them into independent thread groups, and store them in an array declared as follows:

```
dimension partc(lth,4,nppmax,mth)
```

where `lth` refers to tightly coupled threads, either SIMD cores or threads blocks, while `mth` refers to loosely coupled groups of SIMD cores or a grid of thread blocks in CUDA. Because `lth` is the most rapidly varying dimension, particles with adjacent values of the first index are stored in adjacent locations in memory. This is important in achieving stride 1 memory access (or data coalescing in CUDA). In C, the dimensions would be reversed. The total number of independents threads is the product `lth*mth`. The parameter `nppmax` refers to the maximum number of particles in each thread. Note that `lth` is a tunable parameter that we can set to match the computer architecture.

The charge density, on the other hand, has a data dependency or data hazard, since particles in different threads can attempt to simultaneously update the same grid point. There are several possible methods to deal with this data dependency. One way is to use atomic updates, which treat an instruction such as s = s + x as uninterruptible, if they are supported. Supporting locks on memory can achieve this goal. Another method is to determine which of several possible writes actually occurred, and then try again for those writes which did not occur [4]. The last method is to partition memory with extra guards cells so that each thread writes to a different location, then add up those locations that refer to the same grid. This is what is done on distributed memory computers [9]. Since atomic updates are considered to be very slow in the current NVIDA hardware, and the second method seemed to be costly with SIMD processors, we decided on the third method initially.

If particles are sorted by grid, then we can partition the charge density the same way as the particles.

```
dimension qs(lth,4,mth), number of threads: lth*mth = nx*ny
```

where each particle at some grid location can deposit in 4 different locations, 3 of them guard cells. Note that if particles are sorted by grid, then the integer part of the address does not have to be stored, just the deviation. This allows one to get greater precision for the spatial co-ordinates, important when using single precision. The particle co-ordinates would always lie in the range $0 < x < 1$, and $0 < y < 1$. The parallel deposit subroutine is shown below:

```
dimension s(4)            1 local accumulation array s
do m = 1, mth             ! outer loops can be done in parallel
   do l = 1, lth
      s(1) = 0.0          ! zero out accumulation array s
      s(2) = 0.0
      s(3) = 0.0
      s(4) = 0.0
      do j = 1, npp(l,m)         ! loop over particles
         dxp = partc(l,1,j,m)        ! find weights
         dyp = partc(l,2,j,m)
         dxp = qm*dxp
         amy = 1.0 - dyp
         amx = qm - dxp
         s(1) = s(1) + dxp*dyp       ! accumulate charge
         s(2) = s(2) + amx*dyp
         s(3) = s(3) + dxp*amy
         s(4) = s(4) + amx*amy
      enddo
      qs(l,1,m) = qs(l,1,m) + s(1)   ! deposit charge
      qs(l,2,m) = qs(l,2,m) + s(2)
      qs(l,3,m) = qs(l,3,m) + s(3)
      qs(l,4,m) = qs(l,4,m) + s(4)
   enddo
enddo
```

In this algorithm, the particles at a particular cell first deposit to a local accumulation array `s`, of size 4 words. When all the particles are processed, the local accumulation array is added to the charge density array `qs`. The loops over thread indices `l` and `m` can be done in parallel, where the variable `npp(l,m)` contains the number of actual particles assigned to that thread and where `npp(l,m) < nppmax`. When the deposit is completed, the 4 locations in `qs` need to be added to the appropriate locations in the array `q`.

The algorithm can be generalized in several ways. For example, if the cost of maintaining the particle order depends on how many particles are leaving a grid, then this can be reduced by defining a sorting cell to contain multiple grid points. For example, if we define the parameters `ngpx` and `ngpy` to describe the number of gridpoints in x and y in a cell, respectively, then particles will have co-ordinates $0 < x < ngpx$, and $0 < y < ngpy$, stored in arbitrary order within the cell. This also reduces the number of duplicate guard cells needed.

In that case, we have to enlarge the charge density array `qs` and local accumulation array `s`:

```
dimension qs(lth,(ngpx+1)*(ngpy+1),mth), s((ngpx+1)*(ngpy+1))
number of threads: lth*mth = ((nx-1)/ngpx+1)*((ny-1)/ngpy+1)
```

The algorithm would also have to be modified to determine which grid within a cell the particle belongs to and deposit the charge to the appropriate grid points. The structure of the code would remain the same, however, first accumulating all the particles in a cell in the local array `s`, then adding to the density array `qs`. There are no data dependencies anywhere in this procedure.

Another way the algorithm can be generalized is to allow a thread to be responsible for more than one cell. This is useful when the number of threads available is limited. We define the parameter `ngpt` to be the number of sorting cells in a thread. To accommodate this, the particle array needs to have `ngpt` groups of particles for a given thread. The particle structure needs an additional array `kpic` to describe where a given group of particles starts and how many particles it contains:

```
dimension kpic(lth,2,ngpt+1,mth)
```

where `kpic(:,1,k,:)` describes the number of particles in cell number `k` for a given thread and `kpic(:,2,k,:)` describes the memory location where that group starts. The algorithm then contains an additional loop over cells, as follows:

```
do k = 1, ngpt                    ! loop over cells
   joff = kpic(l,2,k,m)
   do j = 1, kpic(l,1,k,m)           ! loop over particles in cell
      dxp = partc(l,1,j+joff,m)           ! find weights
      dyp = partc(l,2,j+joff,m)
      ...
   enddo
enddo
```

We also need to enlarge the charge density array `qs` to the amount:

```
dimension qs(lth,(ngpx*ngpt+1)*(ngpy+1),mth), number of threads:
lth*mth = (((nx-1)/ngpx+1)*((ny-1)/ngpy+1)-1)/ngpt+1
```

Since the deposit will be processed cell by cell, the local accumulation array does not change.

A final generalization is to make the number of cells per thread completely arbitrary. This is useful for achieving load balance. If some cells have more particles than others, one can assign that thread fewer cells to process. The partition in such a case is described by 3 variables in an array `kcell`, which is dimensioned:

```
dimension kcell(lth,3,mth)
```

where `kcell(:,1,:)` describes a minimum cell index, `kcell(:,2,:)` a maximum cell index, and `kcell(:,3,:)` describes the number of grid points in the partition.

Parallel Particle Push

The particle push integrates Newton's equations of motion. In the electrostatic case, we use a simple second-order leap frog-scheme:

```
v(t+dt/2) = v(t-dt/2) + f(x(t))*dt
x(t+dt) = x(t) + v(t+dt/2)*dt
```

where f(x(t)) is the electric force at the particle's location. The particle push is easier to parallelize, since each particle can be treated independently and there are no data dependencies. Since the force array is only read and not written to, it is not necessary to partition it. However, if we do partition it, then it is possible to maintain an optimal stride 1 memory access. Thus we use a distributed force array, similar to the charge density array:

```
dimension fs(lth,2,4,mth)
```

The second dimension refers to the number of dimensions (2), and the third dimension refers to the number of points needed for interpolation (4). The parallel push subroutine is shown below:

```
dimension f(2,4)          ! local force array f
do m = 1, mth             ! outer loops can be done in parallel
   do l = 1, lth
      f(:,1) = fs(1,:,1,m)        ! load local force array
      f(:,2) = fs(1,:,2,m)
      f(:,3) = fs(1,:,3,m)
      f(:,4) = fs(1,:,4,m)
      do j = 1, npp(l,m)          ! loop over particles
         dxp = partc(l,1,j,m)         ! find weights
         dyp = partc(l,2,j,m)
         vx = partc(l,3,j,m)          ! find velocities
         vy = partc(l,4,j,m)
         amy = 1.0 - dyp
         amx = 1.0 - dxp
         dx = dyp*(dxp*f(1,1) + amx*f(1,2))    ! find acceleration in x
              + amy*(dxp*f(1,3) + amx*f(1,4))
         dy = dyp*(dxp*f(2,1) + amx*f(2,2))    ! find acceleration in y
              + amy*(dxp*f(2,3) + amx*f(2,4))
         vx = vx + qtm*dx             ! update particle co-ordinates
         vy = vy + qtm*dy
         dx = dxp + vx*dt
         dy = dyp + vy*dt
         partc(l,1,j,m) = dx          ! store weights
         partc(l,2,j,m) = dy
         partc(l,3,j,m) = vx          ! store velocities
         partc(l,4,j,m) = vy
      enddo
   enddo
enddo
```

Other generalized versions can be created following the pattern illustrated in the parallel charge deposit.

In addition to advancing the particles, the push subroutine also creates a list `nhole` of all the particles that are no longer in the correct cell, and where they need to go. This list is declared as follows:

```
dimension nhole(lth,2,ntmax+1,ngpt+1,mth)
```

where `nhole(:,1,j+1,k,:)` contains the location of the `j`-th particle which needs to be moved from cell `k`, and `nhole(:,2,j+1,k,:)` contains the cell destination. Periodic boundary conditions are assumed. The locations `j` = 1 in this array contains the number of particles that need to be moved in cell k.

Maintaining Particle Order

The most challenging part of implementing a streaming PIC algorithm is maintaining the particle order. In the graphics community, reordering elements into compact substreams of like elements is known as stream compaction. This is less than a full sort, and if particles are already ordered, there should be little cost. The ordering procedure reads the `nhole` array to first determine `ih`, the number of particles to be processed for a given cell. The particles are then handled one at a time, and based on the destination, one of three algorithms are chosen.

The first case applies to particles which are moving from one loosely coupled thread group (thread block in CUDA) to another. If particles have indices `partc(:,:,:,m)`, this means particles with index `m` have a destination with a different index `m`. The particle co-ordinates and the cell destination are copied to a buffer in global memory assigned to each thread called `pbuff`, declared as follows:

```
dimension pbuff(lth,5,npbmx,mth)
```

The number of particles in this buffer is also written out. Once all the particles have been processed and the subroutine ends, this buffer is then read by another program and the particles moved to the proper locations. This algorithm is essentially message-passing, commonly used in distributed memory computers [9], except that the data in the buffer may have multiple destinations. The second program will be described in more detail below.

The second case applies to particles which are moving from one closely coupled thread group to another, that is, between threads which share fast memory (a thread block in CUDA). If particles have indices `partc(l,:,:,m)`, this means particles with index `l` have a destination with a different index `l` but the same m. Since it is possible that two threads may have the same destination thread, we have a possible data hazard. (In fact, the only data hazard in the entire program.) To resolve this, we reserve one word of shared memory for each thread. First we clear the memory (followed by a barrier synchronization). Then those threads which desire to write to another thread's memory, will write their thread index `l` into the shared memory word of the destination thread (and synchronize again). This is a request to send. If two different threads each attempt to write to this request to send location, only one of them will succeed. The threads which wrote requests, then read back what they wrote (in the request to send memory reserved for the remote thread). If what they read agrees with what they wrote, then their request has been accepted. Otherwise, they will try again later. If the request has been accepted, the particle data will be written into a 5 word buffer which has been reserved in shared memory for each thread (followed by another synchronization). Each thread now reads the request to send location reserved for itself. If it is non-zero, then it copies the data in the 5 word particle buffer and stores it in the proper location in its particle array. This location may either be a hole left by a departing particle or at the end of the particle array. Finally, the thread will clear its own request to send in

case more data will arrive, and the process repeats. If the number of threads in a group is less than the number of cells in the x direction, then the process needs to repeat only once.

There are a number of synchronization points in the particle loop, and all processors in the group must participate in the synchronization. Because some threads may finish before others, we need to mask out the remaining calculations on threads which are otherwise done. Thus instead of writing a simple loop over `ih`, we need to write the loop as follows:

```
do j = 1, ihmax
   if (j <= ih) then
   ...
   endif
endo
```

where `ihmax` is the maximum value of `ih` in each thread group. The maximum was found using an algorithm similar to the one supplied in the CUDA Software Development Kit (SDK).

The third case occurs only when there are multiple cells in a thread, and a particle is moving from one cell location to another within the thread. For multiple cells, cells are processed left to right. If the destination is to a cell to the right which has not yet been processed, it is temporarily buffered at the end of the particle array. Otherwise, it is either placed in a hole in the particle array created by a departing particle, or placed at the end of the array. When all the particles in a cell have been processed, holes in the particle array are filled with particles which have been temporarily buffered at the end of the array.

After this procedure has finished, another procedure reads the particle array `pbuff`, which contains particles moving between threads that do not share memory (created in case one above). With linear interpolation and the requirement that particles do not move more than one cell in a time step, for each cell, there are 8 possible cells a particle can move to or from. A list of the threads which are possible sources of particles is contained in an array `icell`. This procedure reads through this list and checks the destination word in `pbuff` for each source to find and extract the particles which belong to this thread. Particles are either placed in any holes which remain, or are placed at the end of the array. If holes still remain, they are filled with particles from the end of the array so that no holes remain.

If the number of cells per thread is completely arbitrary, a small number of additional changes are also needed, such as finding the maximum number of cells in a thread group.

Field Solver

In this simple electrostatic code, the only forces are the electric fields obtained from Poisson's equation. A spectral method is used, with periodic boundary conditions. First one transforms the charge density in real space q(**r**) to q(**k**) in Fourier space with an FFT. Then one multiplies q(**k**) by -i**k**/k^2 to obtain a two component electric field f(**k**) in Fourier space. Finally one transforms the electric field f(**k**) to real space f(**r**) with a Fast Fourier Transform (FFT).

The FFT desired is a 2D real to complex FFT and its inverse. NVIDIA supplies a number of FFTs in the library, including a 2D complex to complex FFT, a 2D real to complex FFT, and various multiple 1D FFTs. However, NVIDIA warns that the real to complex FFTs are not optimized. It is possible to make an optimized real to complex FFT using a well known algorithm[11]. We benchmarked 4 different ways to perform a 2D real to complex FFT using NIVDIA's library: (1) 2D R2C/C2R, (2) multiple 1D R2C/C2C with a transpose, (3) the algorithm from [11] using multiple 1D C2C with a transpose, and (4) 2D C2C, with extraction of the real part. The transpose used was from the sample code supplied in CUDA's SDK. Using algorithm [11] was always the best, varying from slightly faster to 50% faster as the problem size increased.

The real to complex FFT in algorithm [11] first performs multiple FFTs in x for each y. The data must be contiguous with no gaps. This is followed by a correction step and a transpose, then multiple FFTs in y for each x. The order is reversed in going back to real space.

The input data to the FFT is initially stored in the distributed charge density array `qs`, described earlier. The solver proceeds by first adding the guard cells in `qs`, and writes the result into a contiguous array `q`, which is declared as follows:

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

Adding the guard cells has no data hazard and can be done in parallel, if each thread writes only to its own unique portion of the array `q`. The FFT in x is then performed, and transposed to an array declared as follows:

```
dimension qk(2*ny,nx/2+1)
```

The FFT in y is then performed for 2*(nx/2+1) data points. The calculation of the electric field is easily parallelized by assigning each Fourier mode its own thread. The result is a two component electric field declared as follows:

```
dimension fk(2*ny,2,nx/2+1)
```

In going back to real space, the FFT in y is first performed, then transposed to an array declared as follows:

```
dimension f(nx,2,ny)
```

and the FFT is performed in x. Finally guard cells are added to the distributed force array `fs`, described earlier. Again, there are no data hazards in writing the guard cells, since each thread writes its own data.

Performance Results on GPU

The algorithms described above (7 subroutines, not counting the FFT), were first implemented in Fortran and debugged using Pthreads. Since the Fortran compiler for CUDA did not exist at the time (and CUDA was free), we then translated the procedures into C, and debugged that. Finally, a CUDA version was created. Creating the CUDA kernels from C was trivial in most cases. One merely replaced the loops over the parallel threads:

```
for (m = 0; m < mth; m++) {...}
for (l = 0; l < lth; l++) {...}
```

with the CUDA construct:

```
l = threadIdx.x;
m = blockIdx.x;
```

Some additional changes were necessary when shared memory was used. A Fortran callable wrapper function was written to launch the kernels. Generally, one can think of the kernels as the code which goes inside the parallel loop and the parallel loop parameters are set in the procedure which launches the kernels.

The initialization of the code was performed in Fortran and 7 arrays were copied into the GPU. These arrays included the particle data and the charge density as well as arrays describing the partition. The entire inner loop of the PIC code was run on the GPU, while the original Fortran code ran on the host. At the end of the run, the final charge density on the GPU was sent to the host, and compared with the density calculated on the host, to check for correctness. Each procedure was individually timed on the host with the Unix gettimeofday function. In the wrapper function launching the kernel, cudaThreadSynchronize() was called, so that the wrapper did not return to the host before the kernel was done.

The benchmark problem had a grid of 256x512 = 131,072 grid points, with 4,718,592 particles (36 particles per cell). The benchmark was run on a Macintosh Pro, with a 2.66 GHz Intel Nehalem (W3520) processor and a Tesla C0160. The OS running on this machine was Fedora 11, since Mac OS does not support the Tesla. The CUDA version was 2.3. We also ran the benchmark on a Sun workstation with a GTX 280, running CUDA version 2.3. (We also ran some cases on a Sun workstation with a Tesla and CUDA 3.1, and this gave the same GPU results as the Macintosh.) The Fortran compiler used was gfortran. The code was run in single precision, for 100 time steps. The original Fortran code sorted particles every 50 time steps to improve cache performance.

There are 4 tunable parameters in this code. Two of these are the number of grid points in a cell, `ngpx`, and `ngpy`. The larger the number of grid points in a cell, the more shared memory

was needed, and the slower the push and deposit would run. This was primarily due to the scheduler, which limited how many threads could run simultaneously to avoid running out of resources. One can see this effect by increasing the shared memory request parameter when launching the kernel, even on kernels that do not use any shared memory. At the same time, increasing the number of grid points in a cell improved the particle reordering time, since a smaller percentage of particles would leave the cell. The optimum point on the Tesla turned out to be `ngpx` = 2, `ngpy` = 3. Another tunable parameter was the thread block size, `lth`. The results did not vary too much as this parameter was changed, so long as it was at least as large as the warp size, although larger block sizes could run out of resources. For the optimal case, the best value turned out to be `lth` = 32. The best value of the fourth parameter, the number of cells in a thread, was `ngpt` = 1 in all cases. The total number of parallel threads in the optimal case was 21,888. Particle performance was measured in nanoseconds/particle/time step. The field solver took only 7-10% of the code and was not an important factor in the overall performance.

Three cases were run. The first case was a warm plasma, with a time step typical of an electromagnetic code, summarized in Table I. The average percentage of particles leaving a cell at any time step was 1.7%. The performance on the Tesla was 1.21 nsec. with a speedup of 22 compared to the Intel Nehalem. The second case was a hot plasma, with a time step typical of an electrostatic code, summarized in Table II. The average percentage of particles leaving a cell at any time step was 6.6%. The performance was 1.83 nsec. with a speedup of 15. The last was a cold plasma, which indicates an asymptotic limit, when particles are perfectly ordered and never leave their cells, summarized in Table III. The performance was 0.82 nsec. with a speedup of 30. The GTX 280 was about 6-10% faster than the Tesla.

Discussion and Conclusions

This example used a very simple PIC code. There are only about 60 floating point operations per particle per time step in the original code. The calculation is dominated by memory access. Since the Tesla has a peak bandwidth of 102 GBytes/sec, one can calculate that the time to read just the particle and field data, with no latency or overhead, is about 0.400 nsec. The particle push and deposit, without reordering, are running at about 40-50% of the memory bandwidth limit. Including reordering, the results are 20-50% of the memory bandwidth limit. Flops are nearly free.

A 2D electromagnetic code has about 175 floating point operations per particle per time step, and a 3D electromagnetic code has nearly 300. Relativistic codes have even more. Electromagnetic codes differ from electrostatic ones only in the local operations (depositing current in addition to charge, including magnetic forces in the push), but not in the structure of the algorithm. The memory requirements are not substantially higher if the particles are ordered, and the overhead of reordering should become relatively less important. Thus we expect substantially better performance from electromagnetic codes.

Notice that in these algorithms arrays are constantly reconfigured so that they are optimal for each step. For example, the charge is first deposited in a small local array, then copied to a partitioned global array which has optimal stride 1 access. It is then copied again with guard cells to a contiguous array for the FFT, and finally transposed.

One possibility we explored was whether it would be better to add additional guard cells and not reorder the particles every time step. It was always worse to do so, since the slow down due to the additional usage of shared memory was always larger than the benefit of not reordering.

When these algorithms are run on a single processor, the performance is similar to that of the original Fortran code. However, the particles are now ordered, and this is advantageous in implementing PIC codes that include short range binary interactions, such as collisions or the Particle-Particle, Particle-Mesh algorithm [8]. Normally, adding such interactions doubles the cost of the calculation.

In the future, we plan to extend these algorithms to electromagnetic codes and to 3D. In addition, we plan to modify these codes so that they will work with MPI and a cluster of GPUs. We also intend to port this simple code to other architectures, including other GPUs such as Fermi, to see if the parameterization is adequate, and to other languages, such as OpenCL, as they evolve.

Table I: Warm Plasma results with vth = 1.0, dt = 0.025

	Intel Nehalem	Tesla C1060	GTX 280
Push	18.6 ns.	0.67 ns.	0.64 ns.
Deposit	8.7 ns.	0.25 ns.	0.24 ns.
Sort	0.4 ns	0.29 ns.	0.26 ns.
Total Particle	27.7 ns	1.21 ns.	1.13 ns.

The time reported is per particle/time step. The total speedup on the Telsa C1060 was 22x and on the GTX 280 was 24x.

Table II: Hot Plasma results with vth = 1.0, dt = 0.1

	Intel Nehalem	Tesla C1060	GTX 280
Push	18.9 ns.	0.77 ns.	0.73 ns.
Deposit	8.7 ns.	0.26 ns.	0.24 ns.
Sort	0.4 ns	0.81 ns.	0.70 ns.
Total Particle	28.0 ns	1.83 ns.	1.67 ns.

The time reported is per particle/time step. The total speedup on the Telsa C1060 was 15x and on the GTX 280 was 17x.

Table III: Cold Plasma (asymptotic) results with vth = 0.0, dt = 0.025

	Intel Nehalem	Tesla C1060	GTX 280
Push	18.6 ns.	0.56 ns.	0.53 ns.
Deposit	8.5 ns.	0.23 ns.	0.21 ns.
Sort	0.4 ns	0.04 ns.	0.04 ns.
Total Particle	27.5 ns	0.82 ns.	0.78 ns.

The time reported is per particle/time step. The total speedup on the Telsa C1060 was 30x and on the GTX 280 was 33x.

Acknowledgements

This work was supported by the USDOE SciDAC program, the Northrop Grumman Corporation, and the UCLA Institute for Digital Research and Education.

References

[1] P. G. Stanchev, W. Dorland, and N. Gumerov, "Fast parallel Particle-to-Grid Interpolation for plasma PIC simulations on the GPU," J. Parallel Distrib. Comput. 68, 1339 (2009).

[2] V. K. Decyk, T. V. Singh, and S. A. Friedman, "Graphical Processing Unit-Based Simulations," Proc. Intl. Computational Accelerator Physics Conf. (ICAP2009), San Francisco, CA , Sept., 2009.

[3] H. Burau, R. Widera, W. Honig, G. Juckeland, A. Debus, T. Kluge, U. Schramm, T. E. Cowan, R. Sauerbrey, and M. Bussman, "PIConGPU: A fully relativistic particle-in-cell code for a GPU cluster," IEEE Transactions on Plasma Science, 38, 2831, 2010.

[4] X. Kong, C. Ren, M. Huang, and V. Decyk, "Particle-in-cell Simulations with Charge-Conserving Current Deposition on Graphic Processing Units," to be published in J. Computational Phys., 2011.

[5] P. Abreu, R. Fonseca, J. M. Pereira, and L. O. Silva, PIC codes in new processors: a full relativistic PIC code in CUDA enabled hardware with direct visualization," to be published in IEEE Transactions on Plasma Science, 2011.

[6] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K.J. Barker, and D. J. Kerbyson, "0.374 Pflops/s trillion-particle kinetic modeling of laser plasma interaction on Roadrunner," Proc. 2008 ACM/IEEE conference on Supercomputing, Austin, TX, Nov., 2008.

[7] Charles K. Birdsall and A. Bruce Langdon, Plasma Physics via Computer Simulation [McGraw-Hill, New York, 1985].

[8] Roger W. Hockney and James W. Eastwood, Computer Simulation Using Particles," [McGraw-Hill, Nye York, 1981].

[9] P. C. Liewer and V. K. Decyk, "A General Concurrent Algorithm for Plasma Particle-in-Cell Codes, "J. Computational Phys. 85, 302 (1989).

[10] V. K. Decyk, "UPIC: A framework for massively parallel particle-in-cell codes," Computer Phys. Comm. 177, 95 (2007).

[11] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, Numerical Recipes in Fortran [Cambridge University Press, Cambridge, 1992], p. 490.