GRAPHICAL PROCESSING UNIT-BASED PARTICLE-IN-CELL SIMULATIONS*

Viktor K. Decyk, Department of Physics and Astronomy, Tajendra V. Singh and Scott A. Friedman, Institute for Digital Research and Education, UCLA, Los Angeles, CA 90095, U. S. A.

Abstract

New emerging multi-core technologies can achieve high performance, but algorithms often need to be redesigned to make effective use of these processors. We will describe a new approach to Particle-in-Cell (PIC) codes and discuss its application to Graphical Processing Units.

INTRODUCTION

High Performance Computing (HPC) has been dominated for the last 15 years by distributed memory parallel computers and the Message-Passing Interface (MPI) programming paradigm. The computational nodes have been relatively simple, with only a few processing cores. This computational model appears to be reaching a limit, with several hundred thousand simple cores in the IBM Blue Gene. The future computational paradigm will likely consist of much more complex nodes, such as Graphical Processing Units (GPUs) or Cell Processors, which can have hundreds of processing cores, with different and still evolving programming paradigms, such as NVIDIA's CUDA. One anticipates that the next generation HPC computers, unlike Blue Gene, will consist of a relatively small number (<1,000) nodes, each of which will contain hundreds of cores. High performance on the node will in most cases require new algorithms. Between nodes, however, it is likely that MPI will continue to be effective.

Particle-in-Cell (PIC) codes [1-2] are one of the most important codes in plasma physics and other sciences, and use substantial computer time at some of the largest supercomputer centers in the world. Such codes integrate the trajectories of many charged particles, each interacting via electromagnetic fields they themselves produce. In anticipation of future requirements, we have been developing algorithms for PIC codes on this new class of multi-core nodes. As much as possible, we would like these new algorithms to be general enough that they would run well on most of the new emerging architectures. We decided to start with NVIDIA GPUs, because they are powerful, inexpensive, and widely available.

These GPUs consist of 12-30 multiprocessors, each of which has 8 processor cores. The control logic performs the same operation on 32 cores at a time. There is a large (up to 4 GBytes) global memory, which has very high aggregate bandwidth (up to 140 GBytes/sec), far higher than the memory bandwidth of a traditional processor. The memory latency (400-600 clocks) is quite high, however. To hide this latency, the NVIDIA GPUs support thousands of threads simultaneously, and can switch threads in one clock period. To use this architecture, there are two challenges to any algorithm. The first is that the high global memory bandwidth is achieved only when adjacent threads read adjacent locations in memory (stride 1 access, or in the vocabulary of NVIDIA, data coalescing). This is due to the fact that memory is read 64 bytes at a time, and if all 64 bytes are used, memory bandwidth is maximized. The second is that there is no cache. However, each multiprocessor has a small (16 KB), fast (4 clocks) memory which can be shared by threads running on that multiprocessor. It is best to read and write global memory only once (with stride 1 access), storing the data that has to be read more than once or does not have stride 1 access, in small pieces. From this we concluded that ordered, streaming algorithms are optimal for this and similar architectures.

PIC codes codes have 3 major components. The first is a deposit step, where particles contribute charge or current field elements to grid points located near the particle's position. The deposit generally involves a scatter operation. The second is a field solver, where some subset of Maxwell's equation is solved to obtain values of electric and/or magnetic field points on a grid from the charge or current grid points. The third is a particle push step, where particles interpolate electric or magnetic fields at a particle's position by interpolating from nearby field elements. The push generally involves a gather operation. Normally, most of the time is spent in the deposit and push steps, since there are usually many more particles than grids. PIC codes typically have low computational intensity. That is, the number of floating point operations (FLOPs) compared to the number of memory accesses is small, around 2 or 3, so that optimizing memory operations is very important. Parallel algorithms for distributed memory parallel computers have been available for many years [3], and such codes have effectively used 1,000-100,000 processors.

PIC codes can implement a streaming algorithm by keeping particles constantly sorted by grid. This minimizes global memory access, since all the particles at the same grid point read the same field elements: the field elements need to be read only once for the entire group (and can be stored in registers). Cache is not needed, since gather/scatter operations are no longer required. Most importantly, it is possible to store particles so that the deposit and push procedures all have optimal stride 1 memory access. The challenge is whether one can sort the particles in an optimal way.

In this paper, we will discuss an implementation of a streaming algorithm for a simple 2D electrostatic

*Work supported by Northrop Grumman, UCLA IDRE, and USDOE (SciDAC)

particle code on the NVIDIA GPUs. This code involves depositing charge, solving a Poisson equation with a spectral method, and implementing a particle push with electric forces only. It is based on one of the codes from the UPIC Framework [4]. An electromagnetic code would differ 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 or its parallelization. The entire code runs on the GPU, in contrast to an earlier work which implemented only a deposit algorithm [5].

ORDERED CHARGE DEPOSIT

In a traditional PIC code, the particle coordinates are stored as (or ultimately translated to) grid units, where the integer part of a coordinate refers to the nearest grid point, and the deviation from the grid point is used as a weight in the interpolation. In the charge deposit, one would first extract the integer part of the coordinate, and then add the weights to the nearest grid points. The most commonly used interpolation is linear, which in 2D would involve the four nearest grid points. Two arrays are used as the data structures, a particle array part and a charge array q. In Fortran, they would be declared as follows:

```
dimension part(idimp,nop), q(nx+1,ny+1)
```

where idimp is the number of coordinates describing a particle. In this case there are 4 coordinates, corresponding to two positions, x and y, and two velocities, vx and vy, respectively. The size of the grid is given by nx and ny, and nop is the number of particles. The charge on a particle is given by qm. The traditional deposit loop is:

```
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
```

When particles are sorted, a new data structure is needed. Particles can still be stored in a 2D array as before, but they are now grouped together, and there could be gaps between groups, since the number of particles per grid can vary. The location of where a group of particles at a grid starts and the number of particles at that grid are stored in a separate array. The new data structures are declared in Fortran as follows:

```
dimension part(idimp,npmax)     ! npmax > nop
dimension npic(2,nx*ny)
```

The element npic(1,k) contains the number of particles at grid k, and the element npic(2,k) contains the location in the array part where this group starts. The loop over particles now becomes a double loop as follows:

```
do k = 1, nx*ny          ! new outer loop over grids
  joff = npic(2,k)       ! memory offset
  do j = 1, npic(1,k)    ! inner loop over particles at grid
  x = part(1,j+joff)     ! obtain the x coordinate
  ...
  enddo
enddo
```

A charge deposit loop for ordered particles can be written:

```
k2 = 0
do k = 1, nx*ny              ! outer loop over grids
k2 = k2 + 1                  ! increment cell address
sqll = 0.0; squl = 0.0       ! zero out local accumulators
sqlu = 0.0; squu = 0.0
joff = npic(2,k)
do j = 1, npic(1,k)          ! loop over particles at grid
  dxp = qm*(part(1,j+joff))     ! find weights
  dyp = part(2,j+joff)
  amx = qm - dxp
  amy = 1.0 - dyp
  squu = squu + dxp*dyp       ! first sum charges locally
  sqlu = sqlu + amx*dyp
  squl = squl + dxp*amy
  sqll = sqll + amx*amy
enddo
q(k2) = q(k2) + sqll         ! then deposit sum in array
q(k2+1) = q(k2+1) + squl
q(k2+nx+1) = q(k2+nx+1) + sqlu
q(k2+nx+2) = q(k2+nx+2) + squu
enddo
```

Note that the integer part of a coordinate no longer needs to be stored, since it is known from the grid location. This improves accuracy with 32 bit arithmetic, since all bits are used to store weights. The contribution of all the particles at a grid are first summed locally into register variables, then added to the grid. This reduces the number of memory references needed and improves the computation intensity of this subroutine from less than 2 to around 5.

PARALLEL CHARGE DEPOSITS

This ordered algorithm does not run safely in parallel, however, since a particle at one grid writes to other grids, and two threads cannot safely update the same grid at the same time. There are two possible approaches. A traditional approach is to implement an atomic update, where the sum s = s + x is performed as a single,

uninterruptible operation by locking or protecting the memory in some fashion. CUDA supports atomic updates for integers, but not for floating point numbers. Protecting memory, however, is slow and not very portable in most computer languages. An alternative approach is to have each thread write to its own memory locations, which includes additional guard cells that are added up later. Such techniques are common in distributed memory algorithms, but require additional memory. We shall adopt the latter approach, which is known as domain decomposition.

The parallel algorithm will assign each thread ngrid grid points, which are defined in an array kcell. The charge density array q now needs to include guard cells for an extra row and column. If ngrid < nx, the number of guard cells needed is ngrid+2. The worst case is ngrid = 1, and nthreads = nx*ny, when 3 guard cells are needed for each grid. These new data structures are declared in Fortran as follows:

```
dimension q(2*ngrid+2,nthreads)
dimension kcell(2,nthreads)
```

The element kcell(1,kth) contains the initial grid index and kcell(2,kth) contains the final grid index for thread kth.

For a conventional processor, the parallelization can be expressed by adding an OpenMP style outer loop:

```
!$OMP PARALLEL
!$OMP DO
do kth = 1, nthreads   ! parallel loop over threads kth
kmin = kcell(1,kth)    ! minimum cell number for thread
kmax = kcell(2,kth)    ! maximum cell number for thread
ngrid = kmax - kmin + 1      ! number of cells for thread
k2 = 0
  do k = kmin, kmax

! charge deposit loop for ordered particles as previously shown

    q(k2,kth) = q(k2,kth) + sqll       ! deposit sum in array
    q(k2+1,kth) = q(k2+1,kth) + squl
    q(k2+ngrid+1,kth) = q(k2+ngrid+1,kth) + sqlu
    q(k2+ngrid+2,kth) = q(k2+ngrid+2,kth) + squu
    enddo
enddo
!$OMP END DO
!$OMP END PARALLEL
```

This algorithm will run correctly in parallel. However, it will not run optimally on the GPU. The reason is that adjacent threads do not read adjacent locations in memory (stride 1 access is not maintained). To achieve this, we must declare the arrays so that the thread index is the first dimension in Fortran arrays (in C, the last dimension):

```
dimension q(nthreads,2*ngrid+2), kcell(nthreads,2)
```

More importantly, we also need to partition the particle array and its associated data descriptor by thread index as well. We shall also assign the same number of grids to each thread:

```
dimension part(nthreads,idimp,npmax/nthreads)
dimension npic(nthreads,2,ngrid)
```

Other than reorganizing the data with the new partition, the algorithm remains the same.

FIELD SOLVER

The field solver used in this test code solved Poisson's equation, using spectral methods and making use of CUDA's CUFFT library. The algorithm has three steps. First perform a real to complex 2D FFT on the charge density q. Next, multiply the complex charge density qk by the quantity $-i\mathbf{k}/k^2$ to obtain the complex electric field fk. Finally, perform a complex to real 2D FFT to obtain the electric field f in real space.

We decided to use the cufftExecC2C function which performs multiple 1D complex to complex FFTs, and build our own 2D real to complex FFT using a well known algorithm [6]. The CUDA function requires the data to be packed with no gaps between elements. Since the input charge density array has some of the data in guard cells, we add the guard cells as we copy to a contiguous array. If we choose the parallel loop index to correspond to the index of the output array, this operation can be safely run in parallel. The output of this operation is the form:

```
complex, dimension q(nx/2,ny)
```

Once the data is copied, we perform multiple FFTs in x for each y. We then transpose the data, while modifying it as required by the algorithm[6]. This transpose has stride 1 only on the input. NVIDIA has examples of how to improve this, but so little time was used here, we did not do so. Finally, we perform multiple FFTs in y for each x. The result is an complex array qk of the form:

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

The field solver calculates the two component electric field fk in fourier space from qk, and the operation is reversed to obtain the 2 component electric field in real space:

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

The final step is to create an electric field array with guard cells, described next.

PARALLEL PARTICLE PUSH

The particle push integrates Newton's equation of motion using a 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 dt is the time step, and f(x(t)) is the force at the particle's position, found by interpolation. The push subroutine is structured the same as the deposit. To maintain stride 1 memory access, the electric field is partitioned just like the charge density:

```
f(nthreads,2,2*ngrid+2)
```

The partitioning is done by the field solver as its final step. The 8 components needed to interpolate the electric field for the group of particles at a grid are read once and stored in register variables, then reused by all the particles in the group. The inner loop of the push subroutine is as follows:

```
do k = 1, ngrid
  k2 = k2 + 1
  fxll = f(kth,1,k2)                    ! read forces
  fyll = f(kth,2,k2)
  fxul = f(kth,1,k2+1)
  fyul = f(kth,2,k2+1)
  fxlu = f(kth,1,k2+ngrid+1)
  fylu = f(kth,2,k2+ngrid+1)
  fxuu = f(kth,1,k2+ngrid+2)
  fyuu = f(kth,2,k2+ngrid+2)
  joff = npic(kth,2,k)
  do j = 1, npic(kth,1,k) )     ! loop over particles at grid
    dxp = part(kth,1,j+joff)          ! obtain coordinates
    dyp = part(kth,2,j+joff)
    vx = part(kth,3,j+joff)
    vy = part(kth,4,j+joff)
    amx = 1.0 – dxp
    amy = 1.0 - dyp                   ! find acceleration
    dx = dyp*(dxp*fxuu + amx*fxlu)
              + amy*(dxp*fxul + amx*fxll)
    dy = dyp*(dxp*fyuu + amx*fylu)
              + amy*(dxp*fyul + amx*fyll)
    vx = vx + qtm*dx                  ! update coordinates
    vy = vy + qtm*dy
    dx = dxp + vx*dt
    dy = dyp + vy*dt
    part(kth,1,j+joff) = dx           ! write coordinates
    part(kth,2,j+joff) = dy
    part(kth,3,j+joff) = vx
    part(kth,4,j+joff) = vy
  enddo
enddo
```

PARALLEL PARTICLE SORTING

After the particles have been pushed, they may need to be placed in a new grid group and location in memory. We have tried about a dozen algorithms, and finally selected one which appeared to be best. This algorithm assumes that most particles remain in the group, in which case they are written to the same location they had originally. This maintains stride 1 memory access as much as possible.

Within a thread, the groups are processed left to right. If a particle is going to a group outside the thread, the particle coordinates and destination group number are written to a message buffer owned by the thread. In addition, the location of the hole created by the departing particle in the original group is recorded in a hole array. If a particle is going to a group within the thread, there are two possibilities. For a particle going to a group to the right (which has not yet been processed), it is temporarily buffered at the end of the particle array in the destination group, and the location of the hole in the original group is recorded. Once a group has been processed, any holes in the group are filled from the temporarily buffered particles, starting from the last written particle and hole. Finally, if a particle is going to a group to the left (which has already been processed), it is placed either in a hole, if there is one, or added to the end of the group of particles.

Once all the particles are processed (there is an implicit synchronization point here), each thread examines the message buffers created by the other threads to see if any particles belong in this thread. To optimize this search, an array icell is created, which contains for each thread, the index of other possible threads to search. For linear interpolation and a uniform partition, this number is normally 8. The array icell changes whenever the partition described by the array kcell changes. The incoming particles are either placed in a hole, if there is one, or added to the end of the appropriate group. Finally, if any holes in a group are left, they are filled with particles from the end of the group.

For particles leaving a thread group, this algorithm is very similar to the message-passing schemes used by distributed memory PIC codes [3,7].

PERFORMANCE RESULTS

Porting this code to the GPU required first translating six (kernel) subroutines into C, and replacing the loop over threads

```
for (kth = 0; kth < nthreads; kth++)
```

with a special CUDA construct:

```
kth = blockIdx.x*blockDim.x+threadIdx.x;
```

In addition, memory had to be allocated on the GPU, and initial data copied from the host. Finally, wrapper functions were written to enable the kernel subroutines to be called from the main Fortran code. At the end of the simulation, the final charge density array was copied to be host to check for correctness.

The following benchmarks were run on a Macintosh Pro, with a 2.66 GHz Intel Xeon W3520(Nehalem) host and a C1060 Tesla card. Because Mac OS does not support the Tesla card, we installed the Fedora 11 Linux operating system on this hardware. The benchmark application had a 256x512 grid and 4,718,592 particles, with a timestep of $\omega_p dt=0.025$. It was run in single precision, and the time reported is the time per particle per time step. For the benchmark, up to 131,072 threads were possible. However, it turned out that the optimal result was obtained for 8,192 threads (so that each thread had 16 grids), with 128 threads/multiprocessor.

	Intel Nehalem	Nvidia Tesla	Speedup
Deposit:	8.2 nsec.	0.16 nsec.	51
Push:	19.9 nsec.	0.56 nsec.	36
Sort:	-	1.30 nsec.	-
Total:	30.0 nsec.	2.27 nsec.	13

The overall speedup for the entire code was about 13. Most of the time was spent in the sorting step, particularly handling particles moving from one thread group to another. The field solver consumed only a small part of the total time in both cases. It should be noted that the new algorithm is more accurate in single precision than the original algorithm, as explained in the Ordered Charge Deposit section .

DISCUSSION

This version of the PIC code made use of global memory only. Access to global memory is the slowest part of the hardware, and is important to optimize that first. It is a very general algorithm and should run on any processor. The charge and push subroutines improved extremely well, with performance within a factor of 2 of the memory bandwidth limit. Clearly, the sorting step needs the most attention. We expect to improve the algorithm in the future by making use of faster local memories. We were somewhat surprised that we could effectively use distributed memory algorithms on such a device, in avoiding data conflicts and maintaining stride 1 memory access. We were impressed that using CUDA was so simple.

The 2D code used for development here is challenging because there are few operations and the overall computational intensity with the new algorithm improves only from 2 to 4 times. Our target application, however, is a 3D electromagnetic code, and our estimate is that the computational intensity with the new algorithm should improve from 2 to 30 times, so we expect much better results there. The sorting, even if not improved, should become relatively less important.

REFERENCES

[1] Charles K. Birdsall and A. Bruce Langdon, Plasma Physics via Computer Simulation [McGraw-Hill, New York, 1885].

[2] Roger W. Hockney and James W. Eastwood, Computer Simulation Using Particles [McGraw-Hill, New York, 1981].

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

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

[5] 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 (2008).

[6] W. H. Press, S. A. Tekolsky, W. T. Vetterling, and B. P. Flannery, Numerical Recipes in Fortran [Cambridge University Press, 1986], p. 504.

[7] P. M. Lyster, P. C. Liewer, R. D. Ferraro, and V. K. Decyk, "Implementation and Characterization of Three-Dimensional Particle-in-Cell Codes on Multiple-Instruction-Multiple-Data Parallel Supercomputers," Computers in Physics 9, 420 (1995).