Adaptable Particle-in-Cell Codes for Graphical Processing Units

Viktor K. Decyk and Tajendra V. Singh

UCLA

Abstract

We have begun development of an adaptable Particle-in-Cell (PIC) code, whose parameters can match different hardware configurations. The data structures reflect three levels of parallelism, contiguous vectors and non-contiguous blocks of vectors, which can share memory, and groups of blocks which do not. Particles are kept ordered at each time step, and the size of a sorting cell is an adjustable parameter. We have implemented a simple 2D electrostatic and 2-1/2D electromagnetic skeleton code whose inner loops run entirely on the NVIDIA GPUs and achieved speedups of 40-60 compared in a single 2.66 GHz Intel Nehalem i7 processor.

Revolution in Hardware

Many new architectures

- Multi-core processors
- SIMD accelerators, GPUs
- Low power embedded processors, FPGAs

How does one cope with this variety?

Which path leads to exaflops?

Unrest in Parallel Software

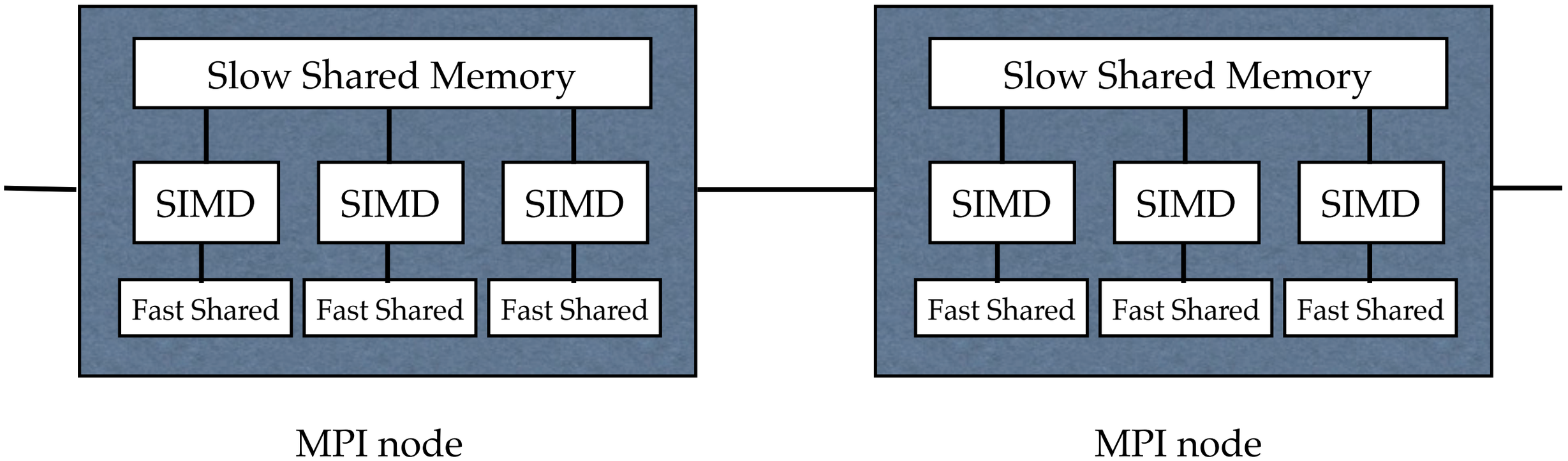
Two programming models
- Distributed memory: MPI
- Shared memory: OpenMP

MPI has dominated high performance computing
- MPI generally worked better even on shared memory hardware
- Shared memory programming models did not scale well

Resurgence in shared memory models
- CUDA on GPUs works well with hundreds of cores
- PGAS Models evolving: Chapel, X10, Co-Array Fortran, UPC

Can we avoid different programming models for different hardware?

Coping Strategy: Simple Hardware Abstraction and Adaptable Algorithms



A distributed memory node consists of

- SIMD (vector) unit works in lockstep with fast shared memory and synchronization
- Multiple SIMD units coupled via slow shared memory and synchronization

Distributed Memory nodes coupled via MPI

Memory is slower than computation, and best accessed with stride 1 addressing

- Streaming algorithms (data read only once) are optimal

Similar to OpenCL model, future exascale computers may be built from these

This hardware model matches GPUs very well

- But can be adapted to other hardware.

On NVIDIA GPU:

- Vector length = block size (typically 32-128)
- Fast shared memory = 16-64 KB.

On Intel multicore:

- Vector length for CPU = 1
- Vector length for SSE = 4
- Fast shared memory = L1 Cache

Particle-in-Cell Codes

PIC codes integrate the trajectories of many particles interacting self-consistently via electromagnetic fields. They model plasmas at the most fundamental, microscopic level of classical physics.

PIC codes are widely used plasma physics, such as in fusion energy research, plasma accelerators, space physics, ion propulsion, plasma processing, and many other areas.

Most complete, but most expensive models. Used when more simple models fail, or to verify the realm of validity of more simple models.

Largest calculations:

- ~1 trillion interacting particles (on Roadrunner, Kraken)
- ~300,000 processors (on Jugene)

Particle-in-Cell Codes

Simplest plasma model is electrostatic:

1. Calculate charge density on a mesh from particles:

$$\rho(\boldsymbol{x}) = \sum_i q_i S(\boldsymbol{x} - \boldsymbol{x}_i)$$

2. Solve Poisson's equation:

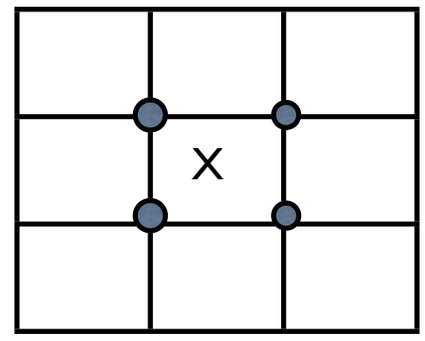
$$\nabla \cdot \boldsymbol{E} = 4\pi\rho$$

3. Advance particle's co-ordinates using Newton's Law:

$$m_i \frac{d\boldsymbol{v}_i}{dt} = q_i \int \boldsymbol{E}(\boldsymbol{x}) S(\boldsymbol{x}_i - \boldsymbol{x})\, d\boldsymbol{x} \qquad \frac{d\boldsymbol{x}_i}{dt} = \boldsymbol{v}_i$$

Inverse interpolation (scatter operation) is used in step 1 to distribute a particle's charge onto nearby locations on a grid.

Interpolation (gather operation) is used in step 3 to approximate the electric field from grids near a particle's location.



Designing New Particle-in-Cell (PIC) Algorithms

Most important bottleneck is memory access
- PIC codes have low computational intensity (few flops/memory access)
- Memory access is irregular (gather/scatter)

PIC codes can implement a streaming algorithm by keeping particles ordered by cell.
- Minimizes global memory access since field elements need to be read only once.
- Cache is not needed, gather/scatter can be avoided.
- Deposit and particles update can have optimal stride 1 access.
- Single precision can be used for particles

Additional benefits for SIMD
- Each cell with associated particles can be processed independently and in lockstep
- Many cells mean fine grain parallelism is possible

Challenge: optimizing particle reordering

Designing New Particle-in-Cell (PIC) Algorithms: Particle Update

The original particle update loop for 2D in Fortran is:

```
   dimension part(idimp,nop), fxy(2,nx+1,ny+1)      ! idimp = number of co-ordinates
                                                    ! nop = number of particles
                                                    ! nx, ny = number of grids in x, y
   do j = 1, nop
      nn = part(1,j)           ! extract x grid point
      mm = part(2,j)           ! extract y grid point
! find interpolation weights
      dxp = part(1,j) - real(nn)
      dyp = part(2,j) - real(mm)
      nn = nn + 1
      mm = mm + 1
      amx = 1.0 - dxp
      amy = 1.0 - dyp
! find acceleration
      dx = dyp*(dxp*fxy(1,nn+1,mm+1) + amx*fxy(1,nn,mm+1))
         + amy*(dxp*fxy(1,nn+1,mm) + amx*fxy(1,nn,mm))
      dy = dyp*(dxp*fxy(2,nn+1,mm+1) + amx*fxy(2,nn,mm+1))
         + amy*(dxp*fxy(2,nn+1,mm) + amx*fxy(2,nn,mm))
! new velocity
      part(3,j) = part(3,j) + qtm*dx
      part(4,j) = part(4,j) + qtm*dy
! new position
      part(1,j) = part(1,j) + part(3,j)*dt
      part(2,j) = part(2,j) + part(4,j)*dt
   enddo
```

Designing New Particle-in-Cell (PIC) Algorithms

A thread is the fundamental unit of parallelization (smallest unit of independent work)

We will associate a thread with one or more cells and particles located at that cell
- Each cell can contain one or more grid points

`lth` = number of threads that work in lockstep as a unit
- We designate this a vector or a block of threads

`mth` = number of independent vectors or blocks
- Total number of threads = `lth*mth`

We create a new data structure for particles, partitioned among threads:

```
dimension part(lth,idimp,npmax,mth)
```

`npmax` is the maximum number of particles in a each thread. In Fortran, the first index is most rapidly varying, so adjacent elements of the vector read adjacent memory locations (stride 1). The C, the dimensions would be reversed. This is **domain decomposition**.

If each thread contains only one cell, which contains only one grid, then
the total number of threads equals the number of grids, `lth*mth` = `nx*ny`

Designing New Particle-in-Cell (PIC) Algorithms

If particles are ordered by grid, we can create a new data structure for the fields, partitioned among threads. One possible partition is:

```
dimension fxy(lth,2,nxys,mth)
```

`nxys` is the maximum number of grids in a each thread. In this partition, each thread contains data for its grids in the cells, plus guard cells: an extra grid on the right, and an extra row of grids on the bottom for linear interpolation.

This data structure allows field data to be accessed with stride 1. However, because of the guard cells, the same data appears in more than one thread, so there is redundant data.

If each thread contains only one cell, which contains only one grid, then `nxys` = 4.

Parallelizing particle update is trivial, each particle is independent of others.

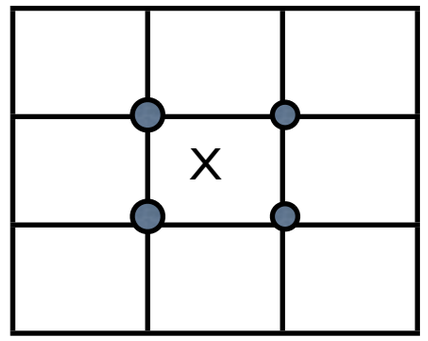
The new particle update loop for special case of 1 grid/thread now has a triple loop:

```
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) = fxy(1,:,1,m)         ! load local force array
      f(:,2) = fxy(1,:,2,m)
      f(:,3) = fxy(1,:,3,m)
      f(:,4) = fxy(1,:,4,m)
      do j = 1, npp(l,m)            ! npp = number of particles in thread
         dxp = part(l,1,j,m)           ! find weights
         dyp = part(l,2,j,m)
         vx = part(l,3,j,m)            ! find velocities
         vy = part(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
         part(l,1,j,m) = dx          ! store weights
         part(l,2,j,m) = dy
         part(l,3,j,m) = vx          ! store velocities
         part(l,4,j,m) = vy
      enddo
   enddo
enddo
```

Particle-in-Cell (PIC) Algorithms: Charge Deposit

Parallel Deposit

To parallelize over grids, there is a problem with the four point interpolation: a particle at one grid writes to other grids, but two threads cannot safely update the same grid point simultaneously. This is called a **data hazard**.



There are three possible approaches

- Use atomic updates or memory locks, if available

[atomic update=the update s = s + x is a single, uninterruptible operation]

- Thread racing: determine which update succeeded, try again for those that did not.
- Use guard cells, then add up the guard cells.

We will use guard cells.

- This avoids any data hazards, widely used in distributed memory computers

Charge density data structure:

```
dimension q(lth,nxys,mth)
```

Designing New Particle-in-Cell (PIC) Algorithms: Charge Deposit
Special case of 1 grid/thread

The original charge deposit loop in 2D

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

  do j = 1, nop
  nn = part(1,j)  ! extract x grid point
  mm = part(2,j)  ! extract y grid point
! find interpolation weights
  dxp = qm*(part(1,j) - real(nn))
  dyp = part(2,j) - real(mm)
  nn = nn + 1
  mm = mm + 1
  amx = qm - dxp
  amy = 1. - dyp
! deposit charge
  q(nn+1,mm+1) = q(nn+1,mm+1) + dxp*dyp
  q(nn,mm+1) = q(nn,mm+1) + amx*dyp
  q(nn+1,mm) = q(nn+1,mm) + dxp*amy
  q(nn,mm) = q(nn,mm) + amx*amy
  enddo
```

The new charge deposit loop in 2D

```
dimension s(4)       1 local accumulation array s

do m = 1, mth
   do l = 1, lth
      s(1) = 0.0     ! zero out accumulation array
      s(2) = 0.0
      s(3) = 0.0
      s(4) = 0.0
      do j = 1, npp(l,m)   ! loop over particles
         dxp = part(l,1,j,m)    ! find weights
         dyp = part(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
      q(l,1,m) = q(l,1,m) + s(1) ! deposit charge
      q(l,2,m) = q(l,2,m) + s(2)
      q(l,3,m) = q(l,3,m) + s(3)
      q(l,4,m) = q(l,4,m) + s(4)
   enddo
enddo
```

Designing New Particle-in-Cell (PIC) Algorithms: Maintaining Particle Order

Stream compaction: reorder elements into substreams of like elements
- Less than a full sort, low overhead if already ordered

Three cases
- 1: Particle moves to another thread in a different vector or block
- 2: Particle moves to another thread in same vector or block
- 3: Particles moves to another cell within same thread

First, create list of particles moving to another cell: `nhole`
- Added to particle update (to avoid reading entire particle data again)

Case 1: Moving to a different vector via slow memory
- Outgoing particles are copied to a particle buffer `pbuff`, along with their destination
- Another procedure reads the buffers, copies particles into proper location
- Essentially message-passing, except buffer contains multiple destinations

New data structures:

```
dimension nhole(lth,2,ntmax+1,mth)
dimension pbuff(lth,idimp+1,npbmx,mth)
```

Designing New Particle-in-Cell (PIC) Algorithms: Maintaining Particle Order

Case 2: Moving to same vector via fast memory

- Write a request to send to a special shared buffer (possible data hazard)
- Determine which request was written (accepted)
- For accepted thread, copy particle data into shared memory particle buffer
- If data awaits, read shared memory particle buffer and copy particle to proper location
- Repeat in case another request is pending

Case 3: only occurs if thread contains multiple cells

- This has not been beneficial on GPUs.

In the end, the particle array belonging to a cell has no gaps

- Particles are moved to any existing holes created by departing particles
- If holes still remain, they are filled with particles from the end of the array

The reordering algorithm does not match the architecture well

- Does not have stride 1 access (poor data coalescing)
- Does not run in lockstep (has warp divergence)

Designing New Particle-in-Cell (PIC) Algorithms

Algorithms can be generalized with additional parameters

Allow multiple grids per cell (`ngpx,ngpy`)

- Field array dimension `nxys` = (`ngpx+1`)*(`npgy+1`)
- Reduces amount of redundant data
- Fewer particles leave cell

Allow multiple cells per thread (`ngpt`)

- Useful if number of thread available is limited
- Has not proved to be beneficial on GPUs.

Allow arbitrary number of cells per thread

- Useful for maintaining load balance

Designing New Particle-in-Cell (PIC) Algorithms: Field Solver

Field Solver is spectral, makes use of FFT:

- Add guard cells in charge density: `dimension q(lth,nxys,mth)`, and copy into contiguous array: `dimension qr(nx,ny)`.
- Perform multiple FFTs of `qr` in x for each y.
- Transpose density: `dimension qk(2*ny,nx/2+1)`.
- Perform multiple FFTs of `qk` in y for each x.
- Solve Poisson equation for electric field: `dimension fk(2*ny,2,nx/2+1)`.
- Perform multiple FFTs of `fk` in y for each x.
- Transpose electric field: `dimension fr(nx,2,ny)`.
- Perform multiple FFTs of `fr` in x for each y.
- copy guard cells in electric field: `dimension fxy(lth,2,nxys,mth)`

Note that field data structure changes in each step.

GPUs are graphical processing units which consist of:

- 12-30 multiprocessors, each with a small (16-48KB), fast (4 clocks) shared memory
- Each multi-processor contains 8-32 processor cores
- Large (0.5-6.0 GB), slow (400-600 clocks) global shared memory, readable by all units
- No cache on some units
- **Very fast (1 clock) hardware thread switching**

GPU Technology has two special features:

- High bandwidth access to global memory (>100 GBytes/sec)
- Ability to handle thousands of threads simultaneously, greatly reducing memory "stalls"

Challenges:

- High global memory bandwidth is achieved only for stride 1 access
  (Stride 1 = adjacent threads read adjacent locations in memory)
- Best to read/write global memory only once

Evaluating New Particle-in-Cell (PIC) Algorithms on GPU: Electrostatic Case

Porting these subroutines to the GPU required:

- First translating subroutines into C
- Replace the loops over threads: "for (m = 0; m < mth; m++)" => "m = blockIdx.x"
  with the CUDA construct: "for (l = 0; l < lth; l++)" => "l = threadIdx.x"
- Write Fortran callable wrapper which invokes subroutine on GPU
- Allocate required arrays in GPU global memory
- Copy initial data to GPU global memory
- Original Fortran program also ran to validate GPU results

2D ES Benchmark with 256x512 grid, 4,718,592 particles, 36 particles/cell

Original Code ran on 2.66 GHz Intel Nehalem (W3520) Host (Macintosh Pro), using gfortran.

Optimal parameters were lth = 32, ngpx = 2, ngpy = 3

Observations:

- About 6% of the particles left the cell each time step
- As ngpx, ngpy increased, reordering time decreased, but push/deposit time increased
- Field solver took about 7-10% of the code

Evaluating New Particle-in-Cell (PIC) Algorithms on GPU: Electrostatic Case

Hot Plasma results with dt = 0.1

	Intel Nehalem	Fermi C2050	Tesla C1060	GTX 280
Push	18.9 ns.	0.56 ns.	0.77 ns.	0.73 ns.
Deposit	8.7 ns.	0.18 ns.	0.26 ns.	0.24 ns.
Reorder	0.4 ns.	0.78 ns.	0.81 ns.	0.70 ns.
Total Particle	28.0 ns.	1.51 ns.	1.83 ns.	1.67 ns.

The time reported is per particle/time step.
The total speedup on the Fermi C2050 was 19x,
on the Telsa C1060 was 15x, and on the GTX 280 was 17x.

Cold Plasma (asymptotic) results with vth = 0, dt = 0.025

	Intel Nehalem	Fermi C2050	Tesla C1060	GTX 280
Push	18.6 ns.	0.43 ns.	0.56 ns.	0.53 ns.
Deposit	8.5 ns.	0.16 ns.	0.23 ns.	0.21 ns.
Reorder	0.4 ns.	0.03 ns.	0.04 ns.	0.04 ns.
Total Particle	27.5 ns.	0.62 ns.	0.82 ns.	0.78 ns.

The time reported is per particle/time step.
The total speedup on the Fermi C2050 was 44x,
on the Telsa C1060 was 30x, and on the GTX 280 was 33x.

Evaluating New Particle-in-Cell (PIC) Algorithms on GPU: Electromagnetic Case

We also implemented a 2-1/2D electromagnetic code, using the same algorithms and same size problem

- Relativistic Boris mover
- Deposit both charge and current density
- Reorder twice per time step
- 10 FFTs per time step

Optimal parameters were lth=64, and
On Fermi C2050, ngpx = 2, ngpy = 2
On Tesla C1060 and GTX280, ngpx = 1, ngpy = 2

Observations:

- About 2.9% of the particles left the cell each time step on Fermi C2050
- About 4.6% of the particles left the cell each time step on Tesla C1060, GTX 2010
- Field solver took about 9% of the code

Evaluating New Particle-in-Cell (PIC) Algorithms on GPU: Electromagnetic Case

Warm Plasma results with c/vth = 10, dt = 0.04

	Intel Nehalem	Fermi C2050	Tesla C1060	GTX 280
Push	81.7 ns.	0.89 ns.	1.13 ns.	1.08 ns.
Deposit	40.7 ns.	0.78 ns.	1.06 ns.	1.04 ns.
Reorder	0.5 ns.	0.57 ns.	1.13 ns.	0.97 ns.
Total Particle	122.9 ns.	2.24 ns.	3.32 ns.	3.09 ns.

The time reported is per particle/time step.
The total speedup on the Fermi C2050 was 55x,
on the Telsa C1060 was 37x, and on the GTX 280 was 40x.

Cold Plasma (asymptotic) results with vth = 0, dt = 0.025

	Intel Nehalem	Fermi C2050	Tesla C1060	GTX 280
Push	78.5 ns.	0.51 ns.	0.79 ns.	0.74 ns.
Deposit	37.3 ns.	0.58 ns.	0.82 ns.	0.81 ns.
Reorder	0.4 ns.	0.10 ns.	0.16 ns.	0.15 ns.
Total Particle	116.2 ns.	1.20 ns.	1.77 ns.	1.70 ns.

The time reported is per particle/time step.
The total speedup on the Fermi C2050 was 97x,
on the Telsa C1060 was 66x, and on the GTX 280 was 69x.

Conclusions

PIC Algorithms are largely a hybrid combination of previous techniques

- Vector techniques from Cray
- Blocking techniques from cache-based architectures
- Message-passing techniques from distributed memory architectures

Scheme should be portable to other architectures with similar hardware abstractions

- OpenCL should allow the code to run on other platforms

Although GPUs and NVIDIA may not survive in the long term, I believe exascale computers will have similar features.

Reference:

V. K. Decyk and T. V. Singh, “Adaptable Particle-in-Cell Algorithms for Graphical Processing Units,” to be published in Computer Physics Communications, 2011.