Plasma Particle-in-Cell Codes on GPUs: A Developer's Perspective

Viktor K. Decyk and Tajendra V. Singh

UCLA

Abstract

Particle-in-Cell (PIC) codes are one of the most important codes used in plasma physics and they use substantial computer time at some of the largest supercomputer centers in the world. This presentation will discuss how we modified the fundamental PIC algorithms in order to run effectively on the NVIDIA GPUs, and the lessons learned from that exercise. Preliminary results are very promising and we expect to achieve a factor of about 50-100 speedup over the Intel Nehalem processors on a production code.

Particle-in-Cell Codes

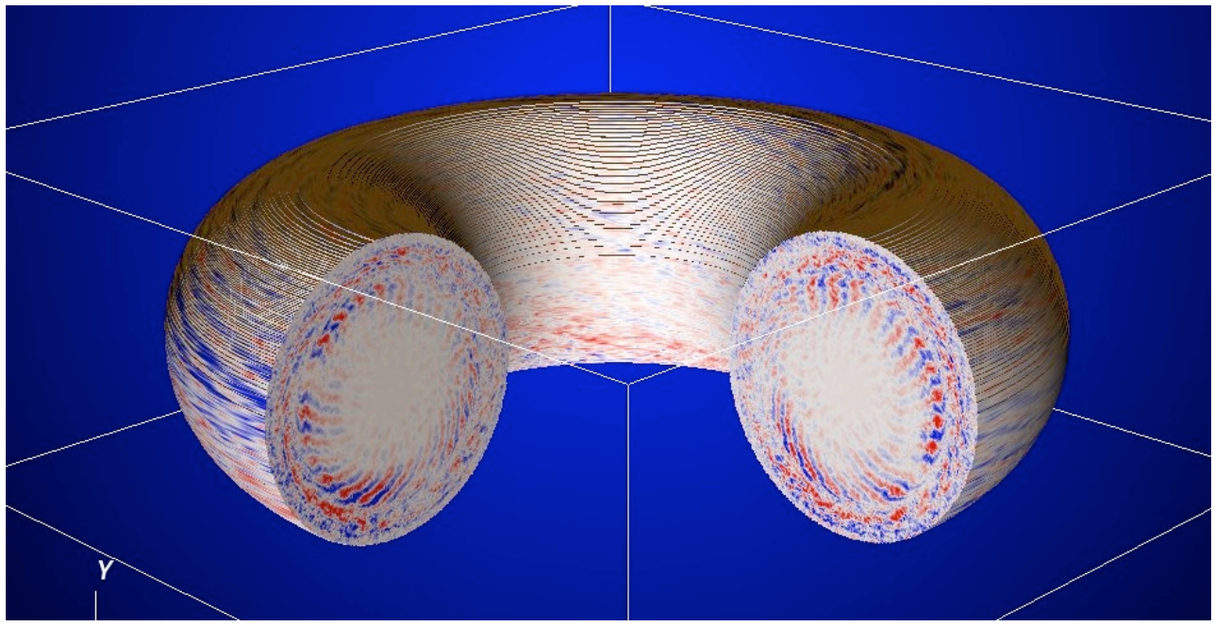
PIC codes integrate the trajectories of many particles interacting self-consistently via electromagnetic fields. They model plasmas at the most fundamental, microscopic level of classical physics.

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

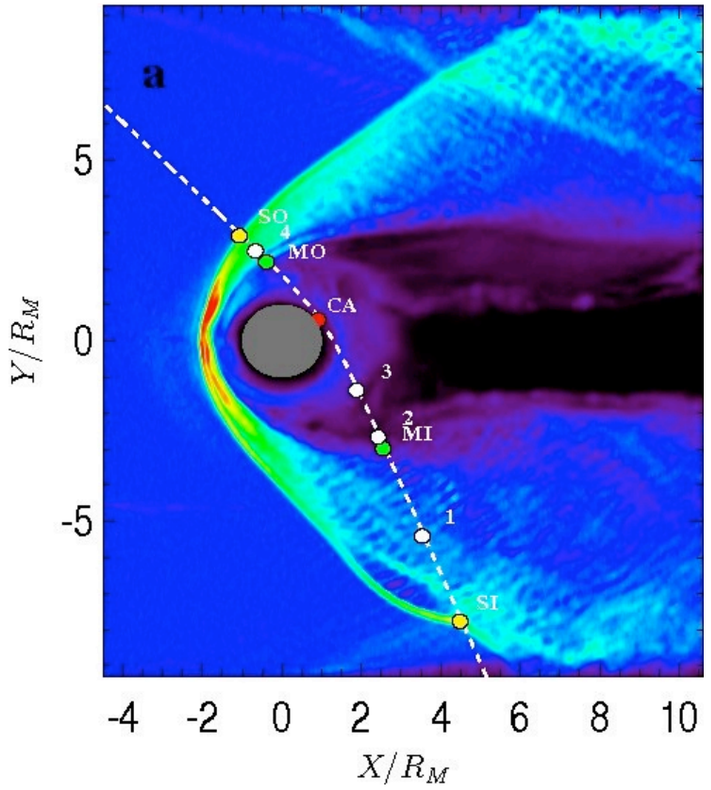
Most complete, but most expensive models. Used when more simple models fail, or to verify the realm of validity of more simple models.

What distinguishes PIC codes from molecular dynamics that a grid is used as a scaffolding to calculate fields rather than direct binary interactions => reduces calculation to order N rather than N^2.

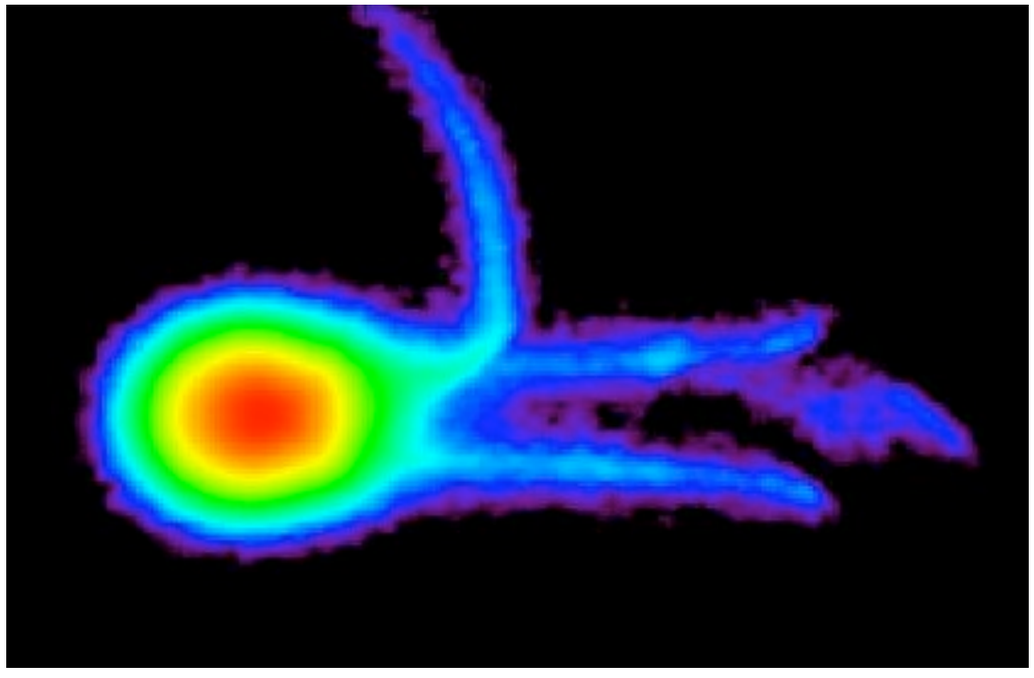
Particle-in-Cell Codes



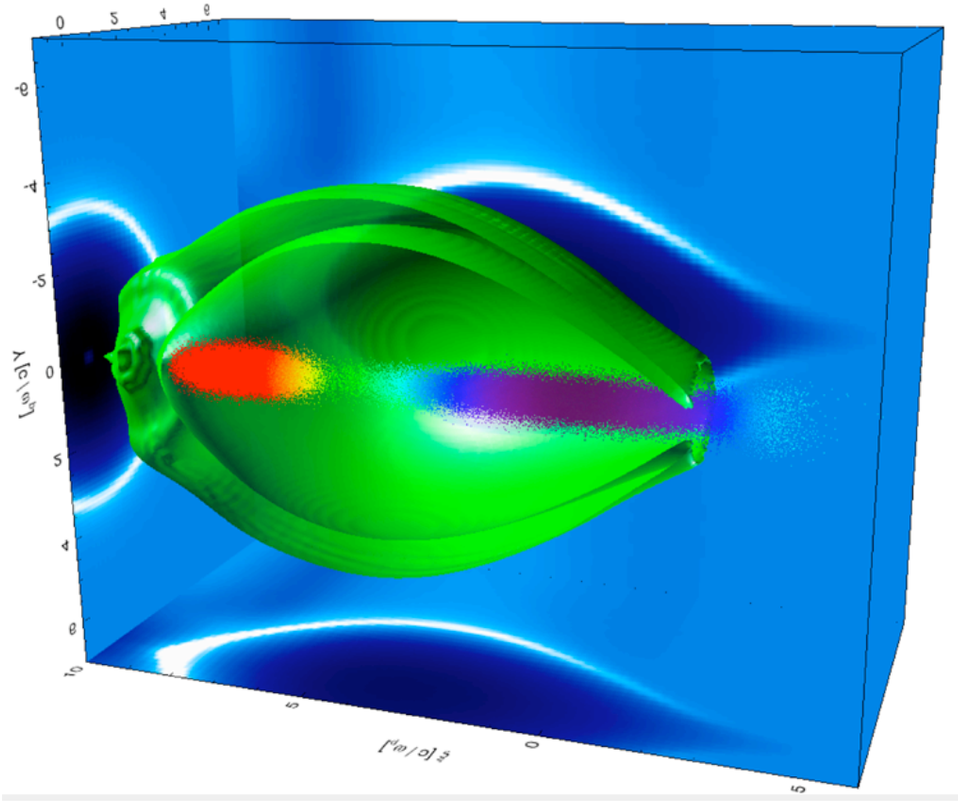
Simulation of ITG turbulence in Tokamaks
R. Sanchez et. al.



Simulation of magnetosphere of Mercury
P. Travnicek et.al,

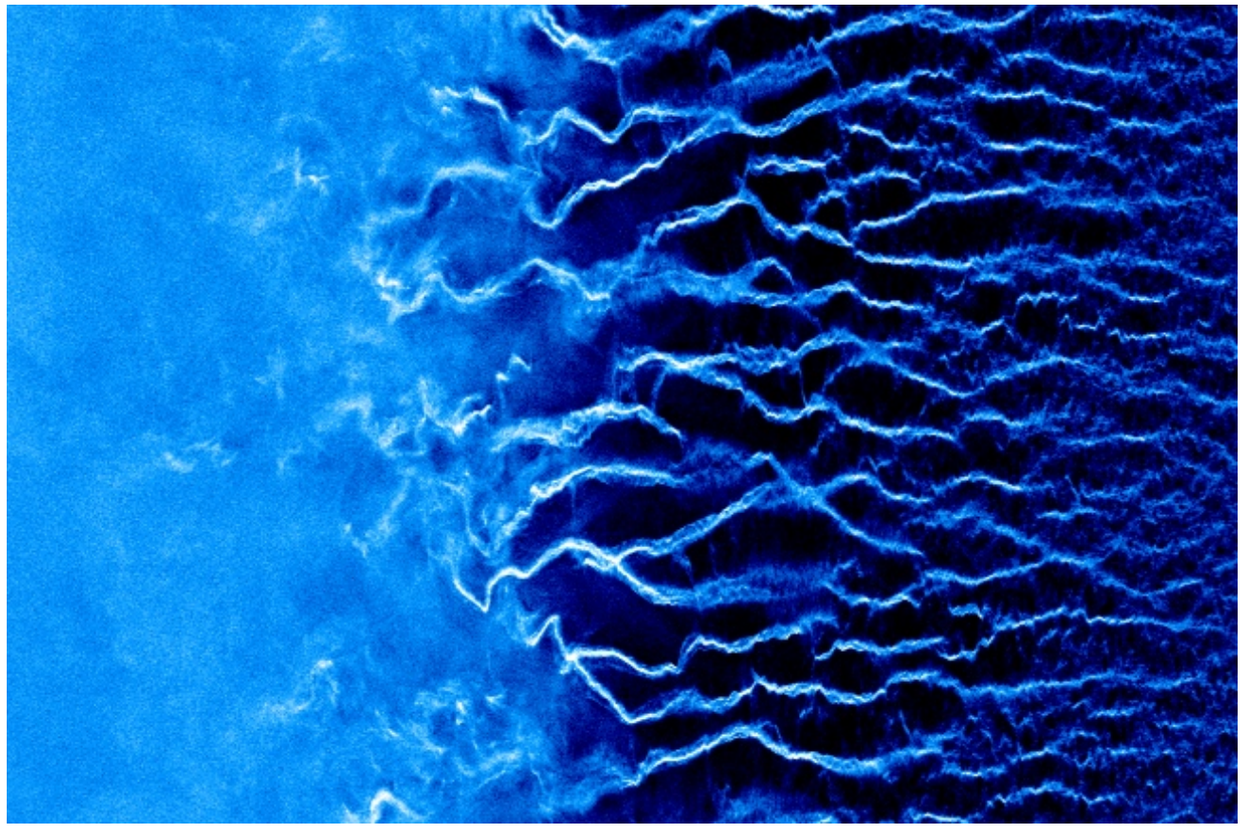


Energetic electrons in laser-driven fusion simulation
J. May and J. Tonge

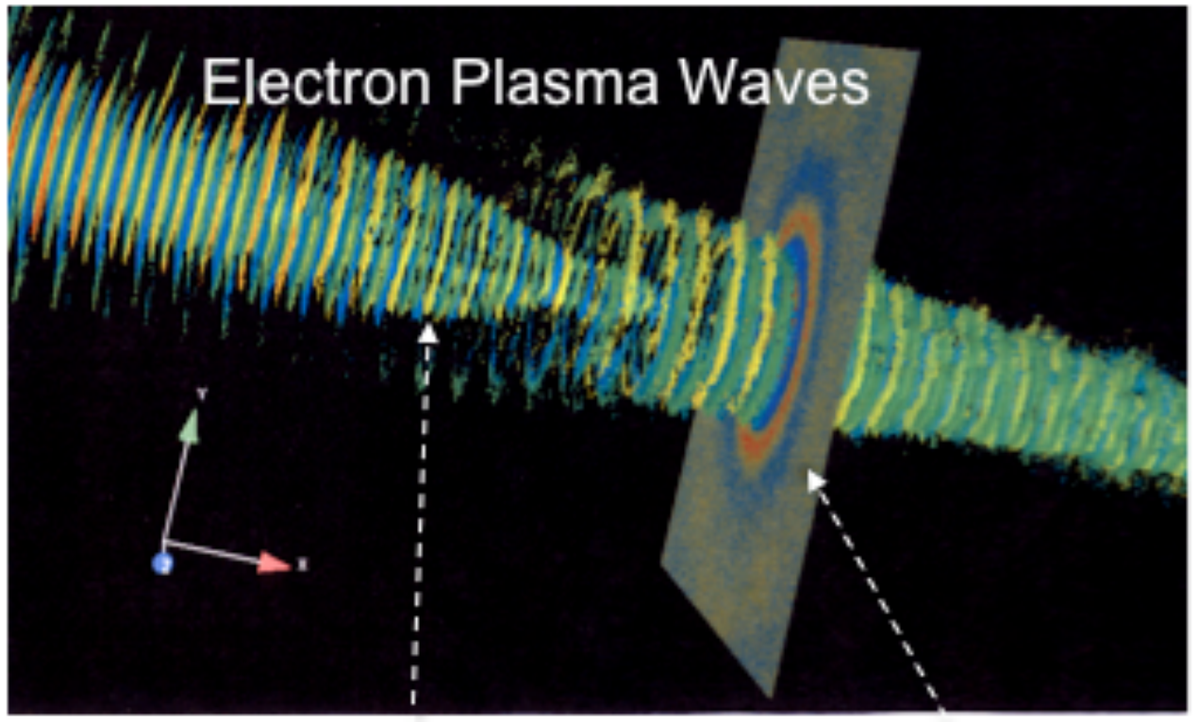


Simulation of compact plasma wakefield accelerator
C. Huang, et. al.

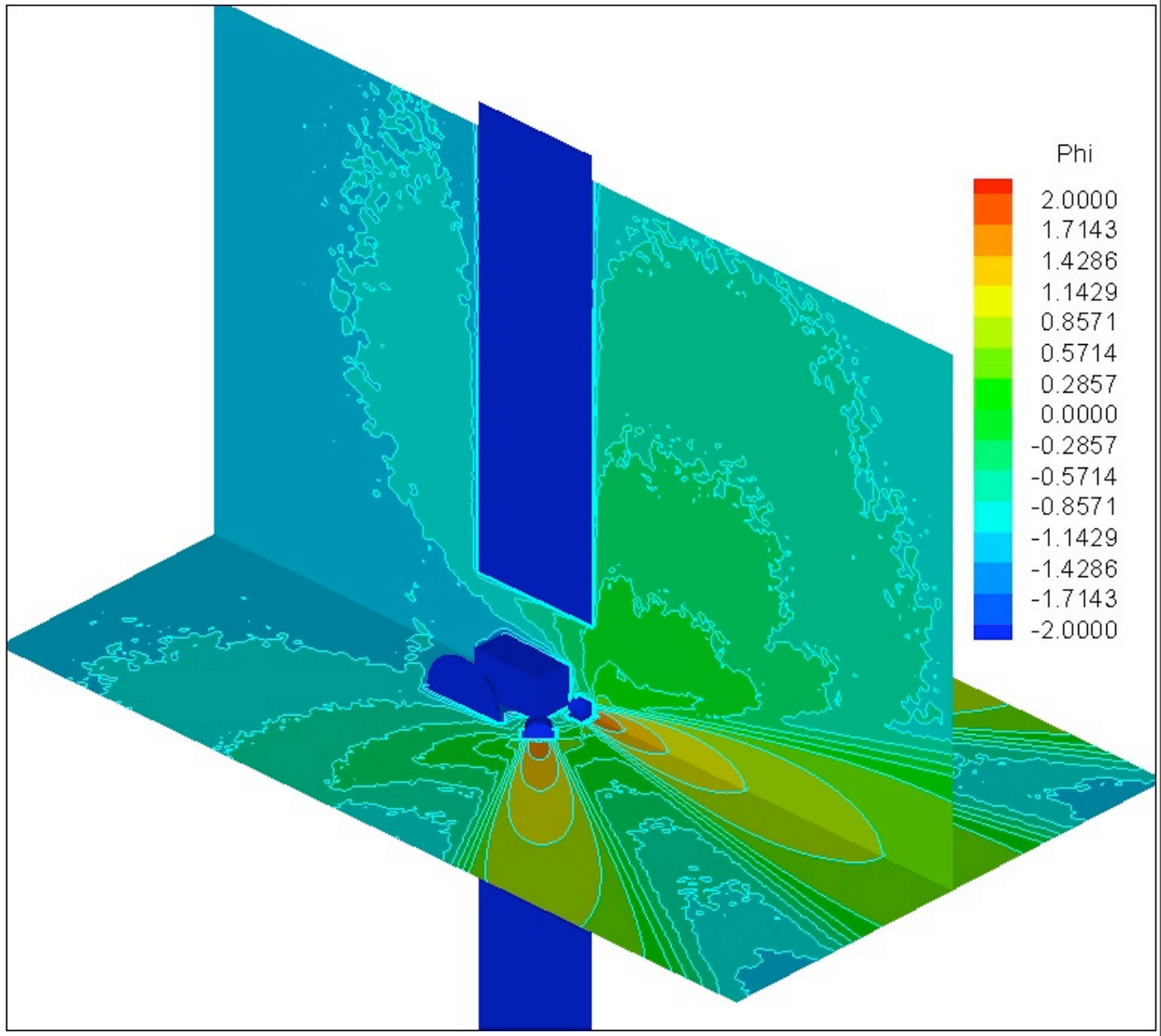
Particle-in-Cell Codes



Relativistic shock simulation, S. Martins, et. al.



Simulation of intense laser-plasma interactions, L. Yin, et. al.



Ion propulsion modeling, J.Wang, et. al.

Particle-in-Cell Codes

Important part of the SciDAC program:

- Magnetic Fusion Energy
- Accelerator Science

Many INCITE grants (~ 200 million processor hours in 2010, 12% of total):

- Tokamak edge modeling
- Tokamak turbulence and transport
- Plasma based accelerators
- Inertial confinement fusion

Largest calculations:

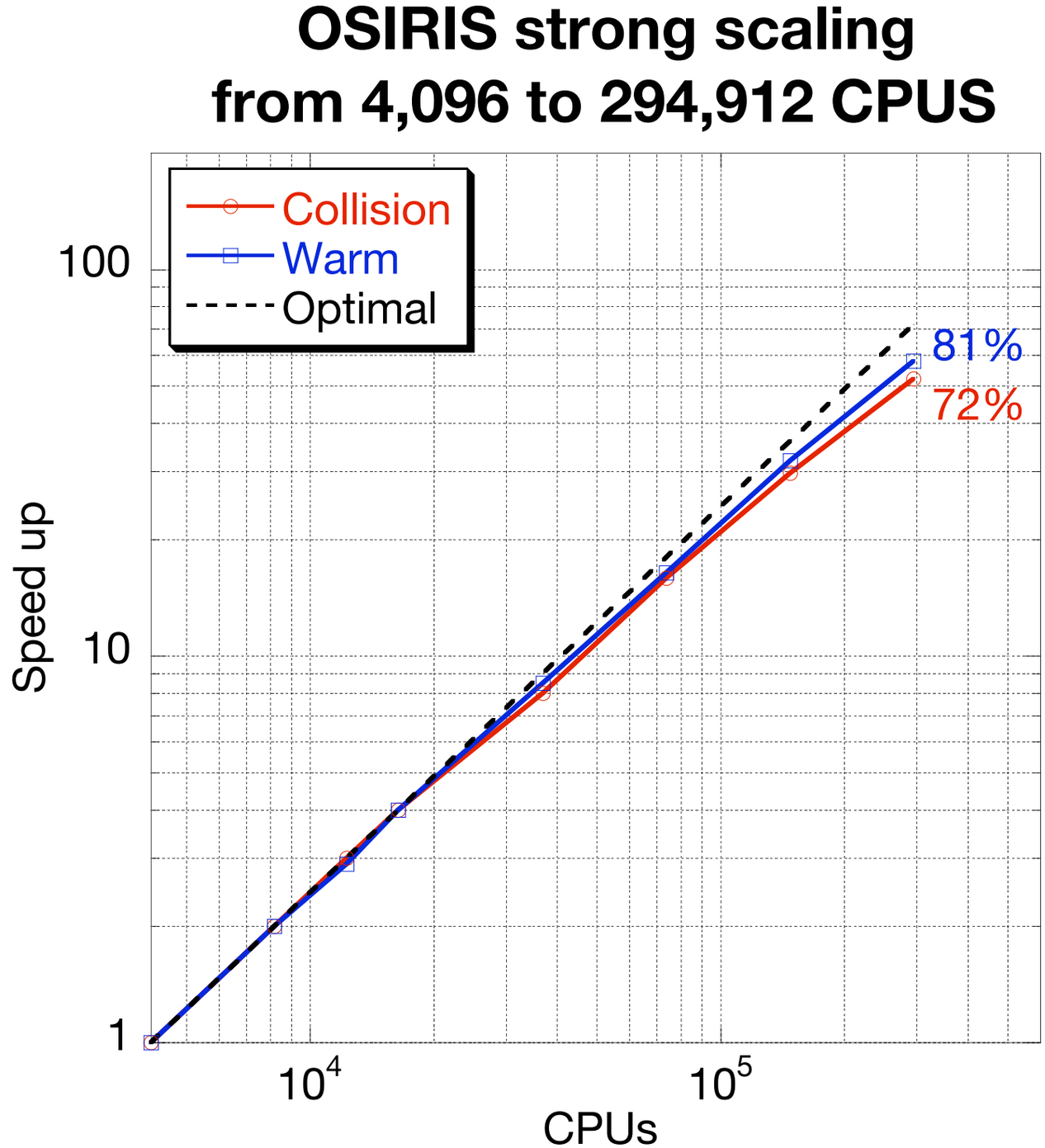
- ~1 trillion interacting particles (on Roadrunner)
- ~300,000 processors (on Jugene)

Particle-in-Cell Codes

Blue Gene Benchmark:

1024x1024x512 cells
8,589,934,592 particles

Not trivial to parallelize:
Two data structures
and inhomogeneous density.



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:

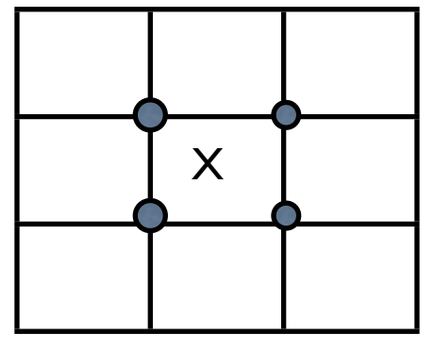
$$\nabla \cdot \boldsymbol{E} = 4\pi\rho$$

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.



GPUs are graphical processing units which consist of:

- 12-30 multiprocessors, each with a small (16KB), fast (4 clocks) shared memory
- Each multi-processor contains 8 processor cores
- Large (0.5-4.0 GB), slow (400-600 clocks) global memory, readable by all units
- No cache
- **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)
- No cache, best to read/write global memory only once

Conclusion: streaming algorithms are optimal

Particle-in-Cell (PIC) Algorithms

PIC codes have low computational intensity (2-3 FLOPs/memory access)

- 2D Electrostatic code has 55 FLOPs/particle update (11 for deposit, 34 for push)
- Memory access is still the bottleneck. FLOPs are cheap

PIC codes can implement a streaming algorithm by keeping particles constantly sorted.

- Minimizes global memory access since field elements need to be read only once.
- Cache is not needed, no gather/scatter.
- Deposit and updating particles can have optimal stride 1 access.
- Single precision can be used for particles

Challenge: optimizing particle sort

Guidelines for Programming GPU

First identify work which can be done independently.
(Typically, loops whose iteration can be done in any order)

To first order, this is similar to:
- vector processors, except favors very long vectors.
- writing parallel loops in OpenMP.

Many optimized algorithms for GPUs have their roots in some vector algorithm on Cray

Original Fortran:

```
do j = 1, nx
  h(j) = f(j)*g(j)
enddo
```

CUDA:

```
j = threadIdx.x + mx*blockIdx.x;
if (j < nx)
   h[j] = f[j]*g[j];
return;
```

OpenMP:

```
!$OMP PARALLEL
!$OMP DO
   do j = 1, nx
     h(j) = f(j)*g(j)
   enddo
!$OMP END DO
!$OMP END PARALLEL
```

Guidelines for Programming GPU

Instead of a single parallel loop, as in OpenMP, CUDA has effectively a double loop:

- An inner loop which shares memory, has fast synchronization, described by the index `threadIdx.x`
- An outer loop which does not, described by the index `blockIdx.x`.

```
OpenMP equivalent:
do blockIdx.x = 1, gridDim.x            ! blocks of threads which do not share memory
do threadIdx.x = 1, blockDim.x          ! threads which share memory
   j = threadIdx.x + mx*(blockIdx.x-1)
   if (j <= nx) h(j) = f(j)*g(j)
enddo
enddo

CUDA:
j = threadIdx.x + mx*blockIdx.x;
if (j < nx)
   h[j] = f[j]*g[j];
return;
```

The if statement is present because `nx` may not be an exact multiple of `blockDim.x`

This dual structure may require a different approach at each level to use effectively.

Guidelines for Programming GPU

Adjacent threads are further divided into SIMD blocks, called warps.

One does not program warps explicitly with SIMD instructions, as in SSE or Altivec, but if instructions within a warp are not the same, the code will run more slowly.

When if statement is needed, it is preferable that all the statements in one warp have the same truth value. (Avoid “warp divergence.”)

Guidelines for Programming GPU

Organize memory reads so adjacent threads read adjacent memory locations.

- Using stride 1 is very important (global memory is read 64 bytes at a time).
- Data locality is largely irrelevant.

This was also true on the Cray, but for different reasons.

- On Cray, non-stride 1 access gave memory bank conflicts; padding arrays avoided that.

On GPU, we read/write memory in "cache lines" (except no cache!).
One solution for non-stride 1 writes:

- accumulate non-stride 1 writes into fast shared memory, then write out with stride 1.
- Similar to blocking algorithms in cache based systems.

The architecture is disruptive, but:
Many algorithms and techniques useful on GPUs are familiar from other architectures.

PIC codes can implement a streaming algorithm by keeping particles sorted. This can be implemented with a new data structure

In the original data structure, particles are stored in a single 2D array

```
dimension part(idimp,nop) ! idimp = dimension of phase space, 4 for 2D
                          ! nop = number of particles

do j = 1, nop             ! original particle loop
x = part(1,j)             ! x co-ordinate
enddo
```

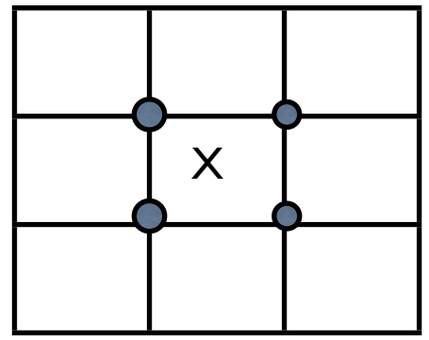
In the new data structure, particles are stored in a single 2D array, but their locations in memory are stored in a separate array. There could be gaps between groups of particle, since the number of particles per cell can vary.

```
dimension parta(idimp,npmax)     ! npmax > nop, includes room for overflow
dimension npic(2,ngrids)         ! ngrids, number of grids, nx*ny in 2D

do k = 1, ngrids           ! new outer loop over grids
npg = npic(1,k)            ! number of particles at this grid
joff = npic(2,k)           ! location where this group starts
do j = 1, npg              ! new inner loop over particles at grid
x = parta(1,j+joff)        ! x co-ordinate
enddo
enddo
```

Particle-in-Cell (PIC) Algorithms

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 two possible approaches
- use guard cells, then add up the guard cells.
- lock memory or use atomic updates if available

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

We will use guard cells, since it does not require special hardware or language features.
- This is essentially **domain decomposition**.

The original charge deposit loop in 2D

```
   dimension part(idimp,nop), q(nxv,nyv)

   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) - float(nn))
   dyp = part(2,j) - float(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

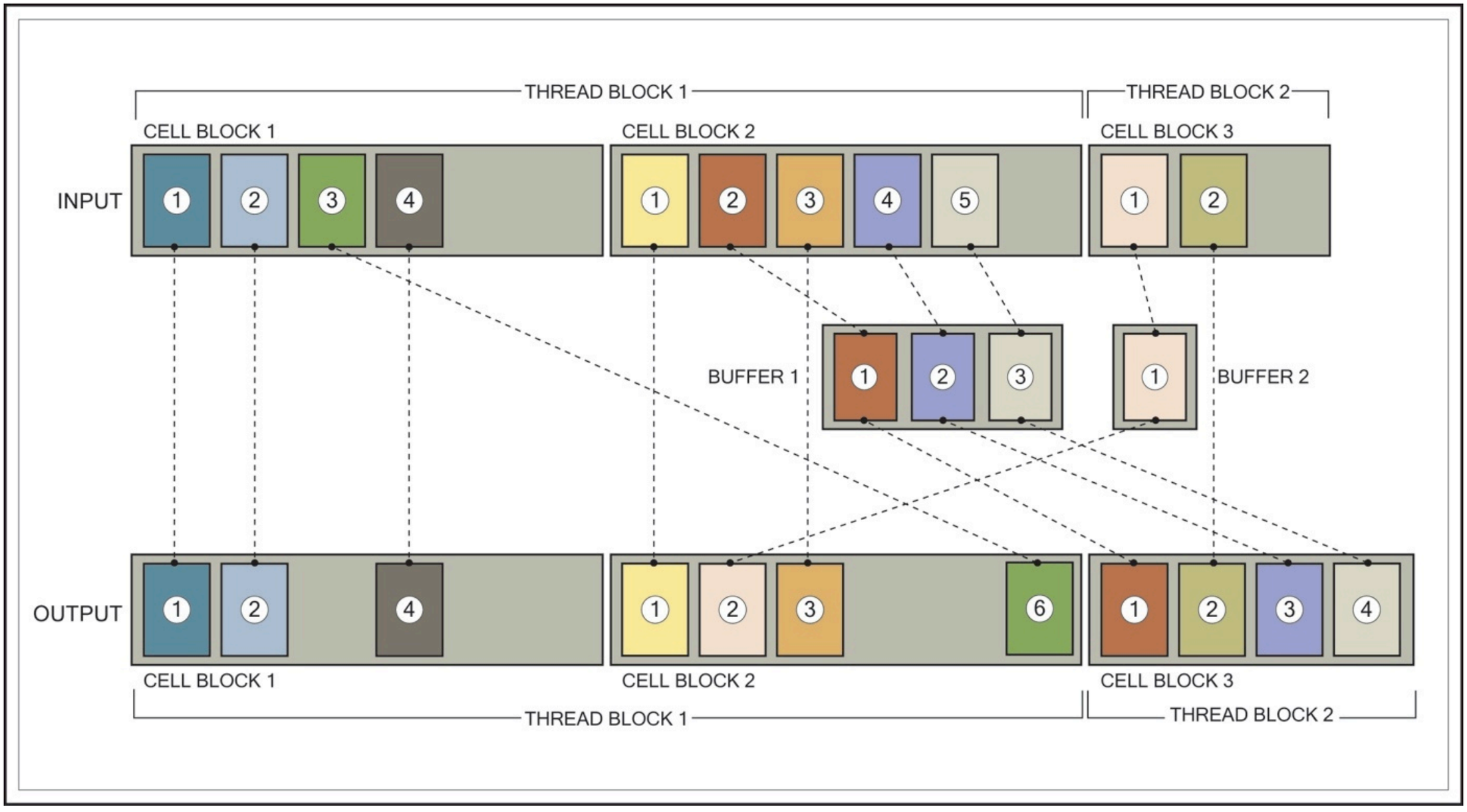
```
dimension partc(nthreads,idimp,nppmax), kpic(nthreads2,nxyp)
dimension q(nthreads,nxys), kcell(nthreads,3)

   do kth = 1, nthreads   ! outermost loop over threads kth
   kmin = kcell(kth,1)    ! minimum cell number for thread
   kmax = kcell(kth,2)    ! maximum cell number for thread
   km = kmax - kmin + 1   ! number of cells for this thread
   nxs = km + 1           ! next row of guard cells
   k2 = 0
   do k = 1, km           ! outer loop over cells in thread
      k2 = k2 + 1         ! increment grid address
      sqll = 0.0
      squl = 0.0
      sqlu = 0.0
      squu = 0.0
      joff = kpic(kth,2,k)     ! offset for this group
      do j = 1, kpic(kth,1,k)  ! inner loop over particles
! find interpolation weights
         dxp = qm*(partc(kth,1,j+joff))
         dyp = partc(kth,2,j+joff)
         amx = qm - dxp
         amy = 1. - dyp
! sum charge
         squu = squu + dxp*dyp
         sqlu = sqlu + amx*dyp
         squl = squl + dxp*amy
         sqll = sqll + amx*amy
      enddo
! deposit charge
      q(kth,k2) = q(kth,k2) + sqll
      q(kth,k2+1) = q(kth,k2+1) + squl
      q(kth,k2+nxs) = q(kth,k2+nxs) + sqlu
      q(kth,k2+nxs+1) = q(kth,k2+nxs+1) + squu
   enddo
   enddo
```

Particle-in-Cell (PIC) Algorithms

Parallel Push: Reading and updating particles is similar to depositing them.

Parallel Sort: Particles leaving a thread group of cells are buffered.
When sort is complete, buffered particles are copied to new locations.

- This is essentially **message-passing**.



Particle-in-Cell (PIC) Algorithms

We have ported the entire PIC loop to the GPU (6 subroutines):

- Charge deposit
- Field solver, including guard cells and transpose, using CUDA FFT
- Particle push and sort

Porting these subroutines to the GPU required:

- First translating subroutines into C
- Replace the main loop over threads: "for (kth = 0; kth < nthreads; n++)"
  with the CUDA construct: "kth = blockIdx.X*blockDim.x+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

2D ES Benchmark with 256x512 grid, 4,718,592 particles, 36 particles/cell, dt = .025

NVIDA Tesla (C1060) compared to the 2.66 GHz Intel Nehalem (W3520) Host:

	Cray C-90	Intel Nehalem	Tesla (C1060)	Speedup
Deposit:		8.2 nsec.	0.18 nsec.	46
Push:		19.9 nsec.	0.54 nsec.	37
Sort:		-	0.60 nsec.	--
Total:	944 nsec.	30.0 nsec.	1.55 nsec.	19

Times for Push and Deposit are about 55% of memory bandwidth limit
Lightweight threads really work!

Memory bandwidth limit = time when memory latency and FLOPs are ignored.
Optimal number of threads turned out to be 8,192, or about 16 grids per thread group.

The **Asymptotic limit** is given for a frozen plasma (vth = 0)

	Intel Nehalem	Tesla (C1060)	Speedup
Deposit:	8.2 nsec.	0.13 nsec.	63
Push:	19.9 nsec.	0.50 nsec.	40
Total:	30.0 nsec.	0.85 nsec.	35

Target application:
3D EM Relativistic code has more FLOPs (around 300) and more memory accesses.
and higher computational intensity.

If the 3D code continues to be memory dominated and have the same memory scaling,
we expect a 3D EM PIC code to run about 3 nsec/particle/time step.

A 3D EM Benchmark on a 2.4 GHz AMD Opteron (Atlas at LLNL) runs about
345 nsec/particle/time step.

=> We expect speedups of about 100 compared to the machines we are currently using.

For multiple GPUs, we think MPI will still be the right model:
Memory to network bandwidths are comparable to today, which works well.

Problem areas:

- Very difficult to debug, emulator not very faithful.
- Sum reductions are currently difficult.
- Occasional incorrect result.

To debug, we run a Fortran code on the host simultaneously.

- We can run either the CUDA or Fortran routine at any point
- Copy out from CUDA and compare

Software development should improve in future

- Emerging standards should help: OpenCL , co-Array Fortran.
- More libraries becoming available: BLAS, FFT, CUDPP
- Vectorizing compiler technology could be exploited.
- OpenMP not adequate, especially with memory hierarchies.
- Non-standard features and extra manual labor should disappear.

Although GPUs and NVIDIA may not survive in long term,
I believe the hardware elements here and the new algorithms needed will survive.