Particle-in-Cell Algorithms for Emerging Computer Architectures: Multiple GPUs

Viktor K. Decyk and Tajendra V. Singh

UCLA

GPUs are graphical processing units originally developed for graphics and games

- Programmable in 2007

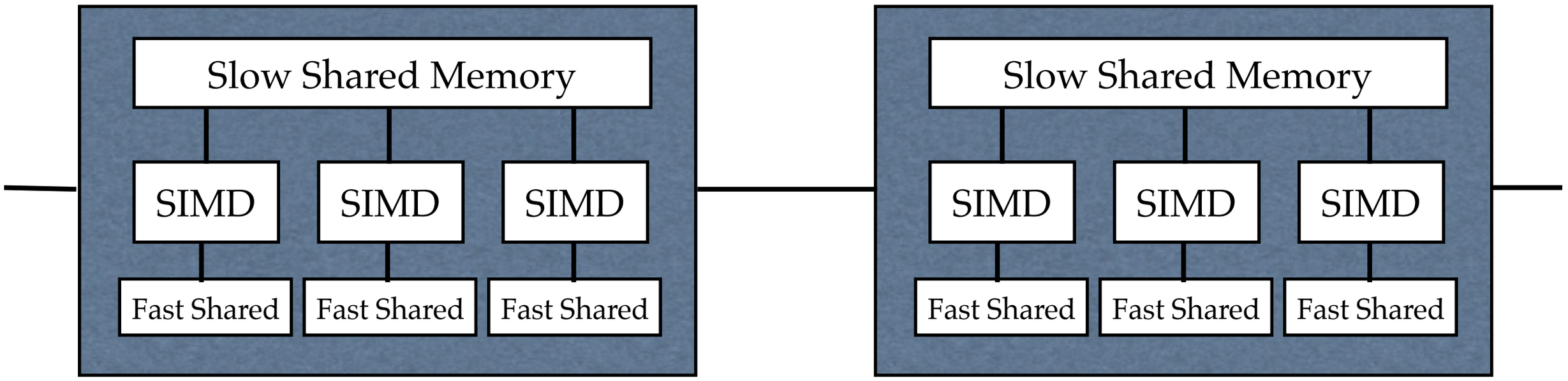
GPUs consist of:

- 12-30 SIMD multiprocessors, each with 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), but for ordered access
- Ability to handle thousands of threads simultaneously, greatly reducing memory "stalls"

Simple Hardware Abstraction for Next Generation Supercomputer



MPI node

MPI node

A distributed memory node consists of

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

Distributed Memory nodes coupled via MPI

Each MPI node is a powerful computer by itself
A supercomputer is a hierarchy of such powerful computers

OpenCL programming model uses such an abstraction for a single node

This hardware model matches a variety of processors

On NVIDIA GPU:
- Vector length = block size (typically 32-128)
- Number of vector processors =12-30
- Fast shared memory = 16-64 KB, plus L2 cache

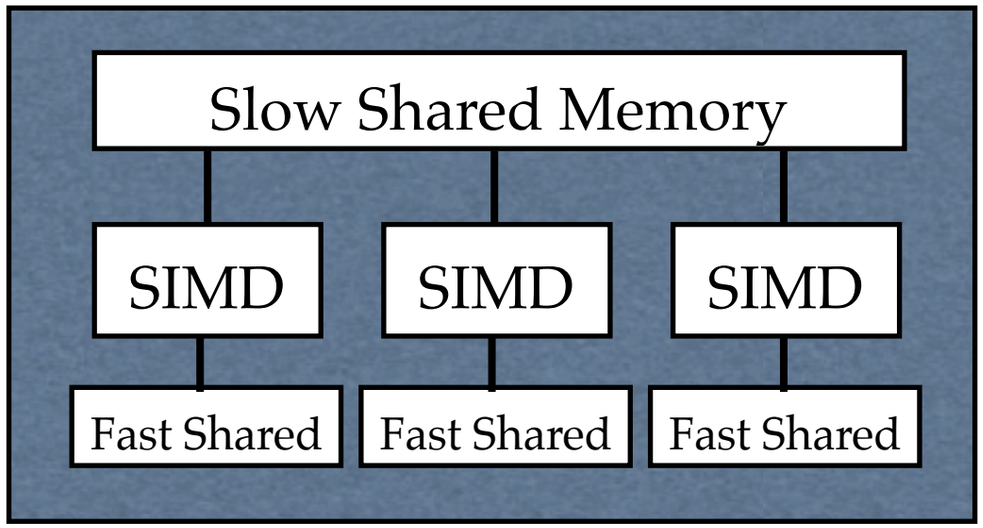
On Intel MIC (PHI):
- Vector length for single precision = 16
- Number of vector processors = 50-60
- Fast shared memory = L1 Cache (32 KB), L2 Cache (512 KB)

On Dual Intel Xeon multicore:
- Vector length for single precision with SSE2 = 4
- Number of vector processors = 4-16
- Fast shared memory = L1, L2 Cache

Designing new algorithms for next generation computers

This abstract machine contains a number of familiar hardware elements

- SIMD (vector) processors
- Small working memory (caches)
- Distributed memories



Scientific programmers have experience with each of these elements

Vector algorithms

- Calculation on a data set whose elements are independent (can be done in any order)
- Long history on Crays, Fijitsu, NEC supercomputers

Blocking (tiling) algorithms

- When data will be read repeatedly, load into faster memory and calculate in chunks
- Long history on RISC processors

Domain decomposition algorithms

- Partition memory so different threads work on different data
- Long history on distributed memory computers

Designing new algorithms for next generation computers

Programming these new machines uses many familiar elements,
but put together in a new way.

But some features are unfamiliar:

Languages (CUDA, OpenCL, OpenACC, OpenMP) have new features

GPUs require many more threads than physical processors

- Hides memory latency
- Hardware can use master-slave model for automatic load balancing among blocks

Optimizing data movement is very critical

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.

Largest calculations:

- ~3 trillion interacting particles
- ~1.6 million processors (on Sequoia)

Particle-in-Cell Codes

Simplest plasma model is electrostatic:

1. Calculate charge density on a mesh from particles:

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

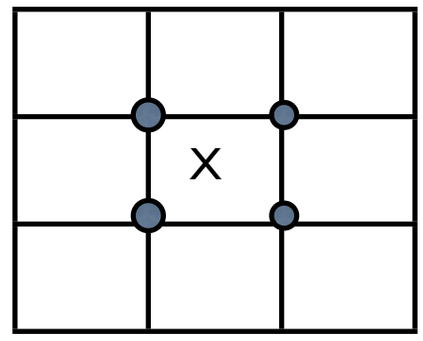
2. Solve Poisson's equation:

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

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.

When running in parallel, data collisions might occur

Distributed Memory Programming (MPI) for PIC

This is dominant technique used on today's supercomputers

Domain decomposition is used to assign which data reside on which processor
- No shared memory, data between nodes sent with message-passing (MPI)
- Most parallel PIC codes are written this way

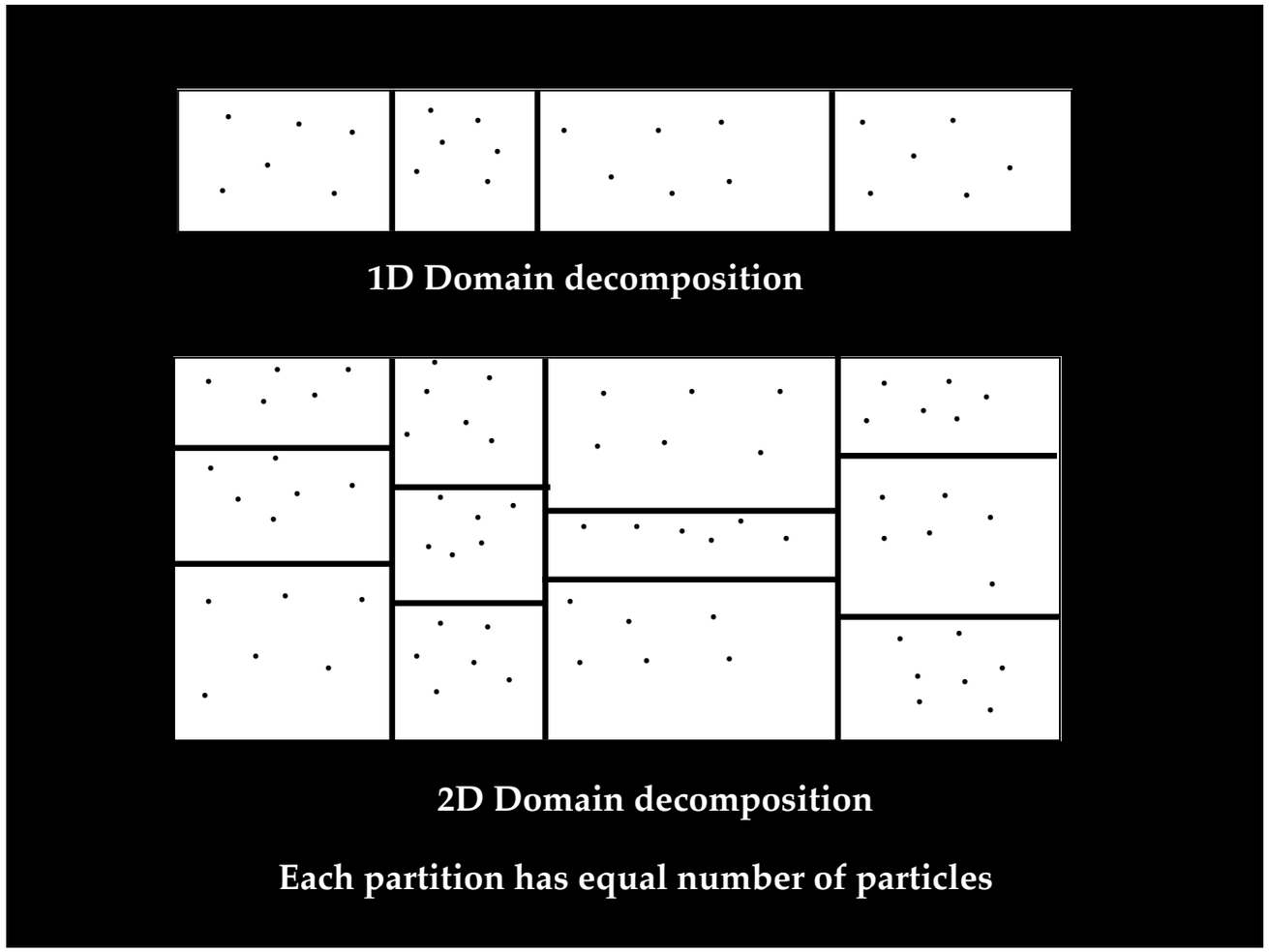
Details can be found at:

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

Sample codes can be found at:
https://idre.ucla.edu/hpc/parallel-plasma-pic-codes

Distributed Memory Programming for PIC

Domain decomposition with MPI



Primary Decomposition has non-uniform partitions to load balance particles

- Sort particles according to spatial location
- Same number of particles in each non-uniform domain
- Scales to many thousands of processors

Particle Manager responsible for moving particles

- Particles can move across multiple nodes

GPU Programming for PIC

Most important bottleneck is memory access

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

Memory access can be optimized with a streaming algorithm (global data read only once)

PIC codes can implement a streaming algorithm by keeping particles ordered by tiles.

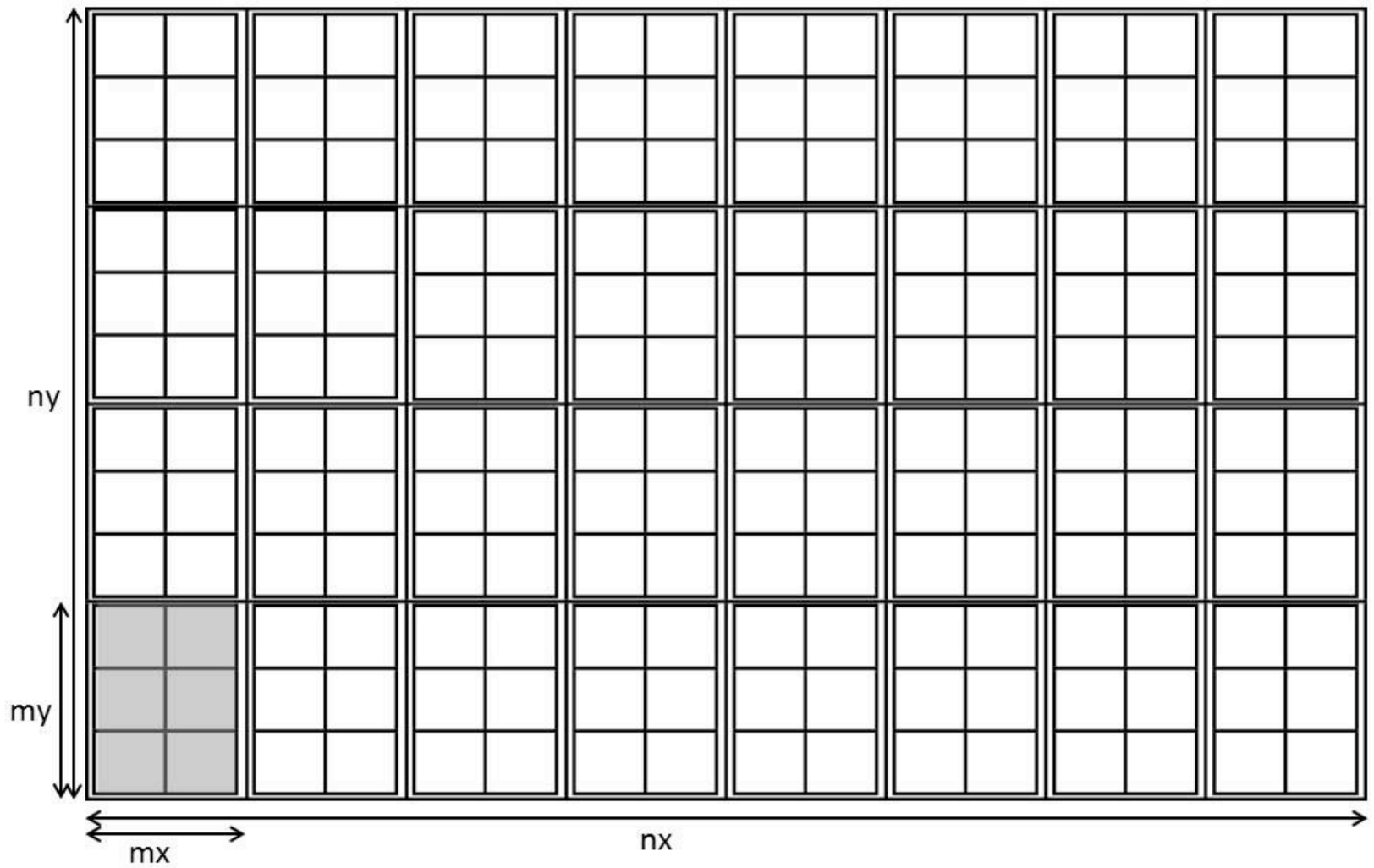
- Minimizes global memory access since field elements need to be read only once.
- global gather/scatter can be avoided.
- Deposit and particles update can have optimal memory access.
- Single precision can be used for particles

Challenge: optimizing particle reordering

Designing New Particle-in-Cell (PIC) Algorithms:
Particles ordered by tiles, varying from 2 x 2 to 16 x 16 grid points

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

```
dimension ppart(idimp,npmax,num_tiles)
```



Designing New Particle-in-Cell (PIC) Algorithms: **Push/Deposit Procedures**:

Within a tile, all particles read or write the same block of fields.

- Before pushing particles, copy fields to fast memory
- After depositing charge to fast memory, write to global memory
- Different tiles can be done in parallel.

Each tile contains data for the grids in the tile, plus guard cells: an extra column of grids on the right, and an extra row of grids on the bottom for linear interpolation.

For push, parallelization is easy, each particle is independent of others, no data hazards

- Similar to MPI code, but with tiny partitions

For deposit, parallelization is also easy if each tile is controlled by one thread

- This avoids data collisions where two threads try to update the same memory

However, if each tile is controlled by a vector of threads, data collisions are possible.

- Atomic updates (which treat an update as an uninterruptible operation) are one approach

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

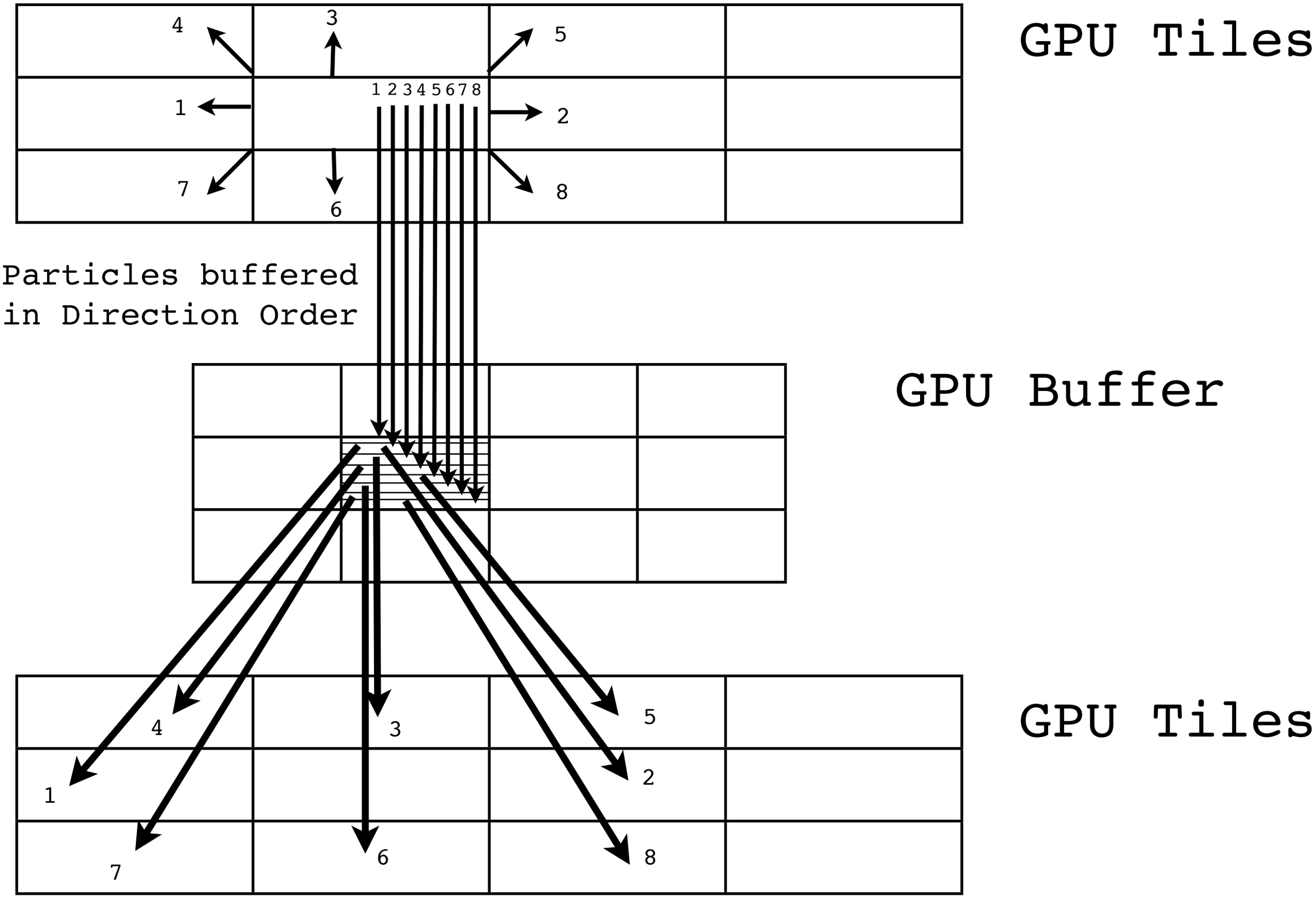
Three steps:
1. Create a list of particles which are leaving a tile, and where they are going
2. Using list, each thread places outgoing particles into an ordered buffer it controls
3. Using lists, each tile copies incoming particles from buffers into particle array

- Less than a full sort, low overhead if particles already in correct tile
- Can be done in parallel
- Essentially message-passing, except buffer contains multiple destinations

In the end, the particle array belonging to a tile has no gaps
- Incoming 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

Straightforward to implement with one thread per tile
- Much more complex with multiple threads per tile

GPU Particle Reordering



Evaluating New Particle-in-Cell (PIC) Algorithms on GPU: **Electrostatic Case**
2D ES Benchmark with 2048x2048 grid, 150,994,944 particles, 36 particles/cell
optimal block size = 128, optimal tile size = 16x16. Single precision

GPU algorithm also implemented in OpenMP

```
Hot Plasma results with dt = 0.1
              CPU:Intel i7    GPU:Fermi M2090
Push             20.7 ns.        0.552 ns.
Deposit           9.4 ns.        0.247 ns.
Reorder           1.6 ns.        0.114 ns.
Total Particle   31.7 ns.        0.914 ns.

The time reported is per particle/time step.
The total particle speedup on the Fermi M2090 was 35x compared to 1 CPU.
Field solver takes an additional 12% on GPU, 11% on CPU.
```

Evaluating New Particle-in-Cell (PIC) Algorithms on GPU: **Electromagnetic Case**
2-1/2D EM Benchmark with 2048x2048 grid, 150,994,944 particles, 36 particles/cell
optimal block size = 128, optimal tile size = 16x16. Single precision

GPU algorithm also implemented in OpenMP

```
Hot Plasma results with dt = 0.04, c/vth = 10, relativistic
              CPU:Intel i7    GPU:Fermi M2090
Push             68.4 ns.        0.980 ns.
Deposit          43.2 ns.        1.327 ns.
Reorder           0.6 ns.        0.068 ns.
Total Particle  112.2 ns.        2.375 ns.

The time reported is per particle/time step.
The total particle speedup on the Fermi M2090 was 47x compared to 1 CPU.
Field solver takes an additional 16% on GPU, 11% on CPU.
```

Designing New Particle-in-Cell (PIC) Algorithms: **Multiple GPUs**

Multiple GPUs can be controlled with MPI

- Merge MPI and GPU algorithms

We started with existing 2D MPI codes from UPIC Framework

- Replacing MPI push/deposit with GPU version was no major challenge

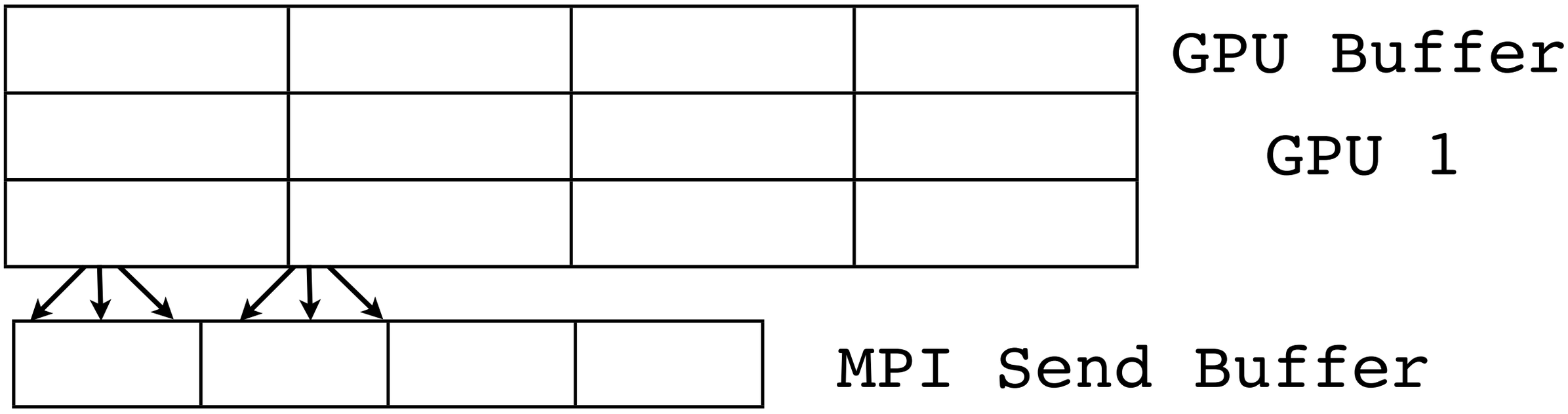
With multiple GPUs, we need to integrate two different particle partitions

- MPI and GPU each have their own particle managers to maintain particle order

Only the first/last row or column of tiles on GPU interacts with neighboring MPI node

- Particles in row/column of tiles collected in MPI send buffer
- Table of outgoing particles are also sent
- Table is used to determine where incoming particles must be placed

GPU-MPI Particle Reordering



FFT requires an all-to-all transpose among GPUs

Data is first transposed on GPU into buffers gs
GPU buffers gs are copied to host buffers s
MPI on host sends buffers s, receives data in buffers r.
Host buffers are copied to GPU buffers gr
Buffer gr is further transposed on GPU.

Synchronous example with 4 nodes

Copy GPU data to host: gs(1:4)->s(1:4)

MPISend s(1)	MPISend s(2)	MPISend s(3)	MPISend s(4)
MPIRecv r(1)	MPIRecv r(2)	MPIRecv r(3)	MPIRecv r(4)

Copy host data to GPU: r(1:4)->gr(1:4)

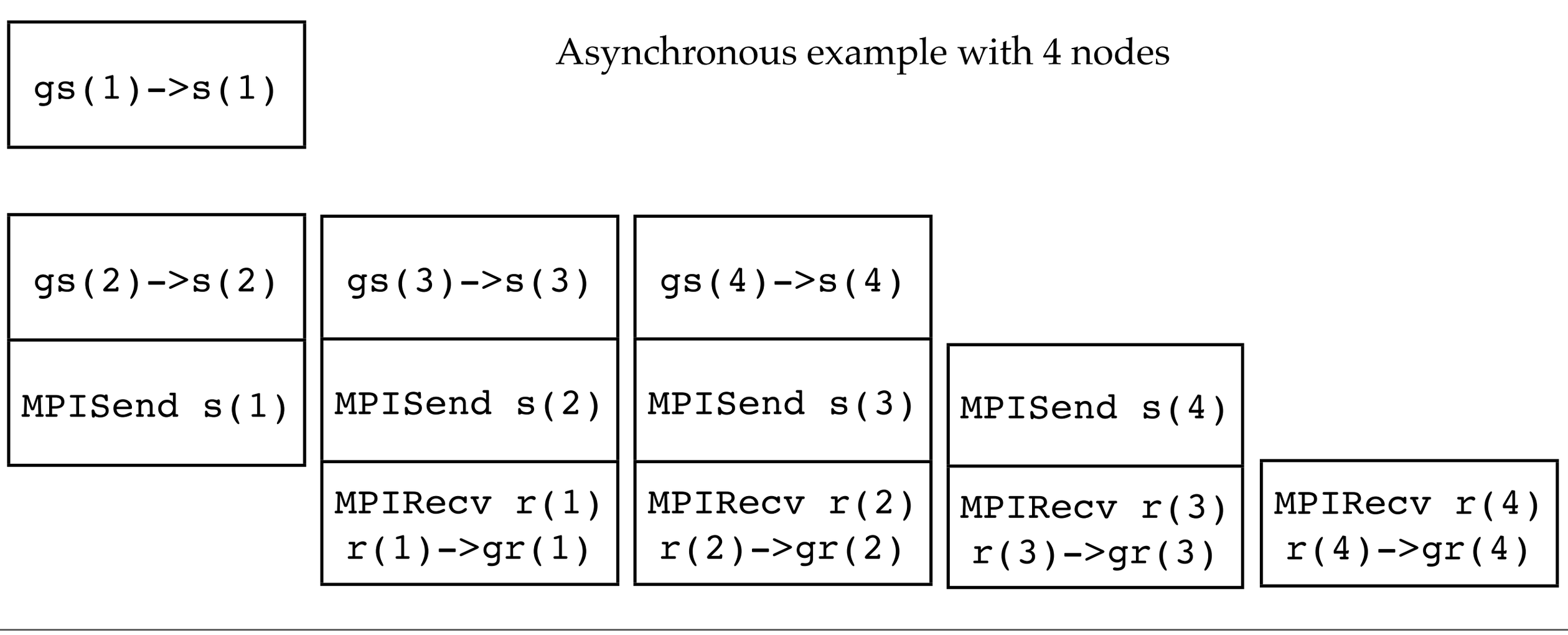
All-to-all transpose among GPUs: asynchronous version

Overlap the copying between GPUs and host with MPI messages with three buffers

- One buffer sends a segment from GPU to host
- Another buffer sends a second segment between GPUs
- A third buffer sends a third segment from host to GPU.

Improvement in performance is modest, typically less than 10%

Asynchronous example with 4 nodes



Designing New Particle-in-Cell (PIC) Algorithms: **Multiple GPUs**

One MPI node is used to control a single GPU. If multiple GPUs can exist on a single host, a procedure is needed to assign each MPI node a different GPU device id.

This was done by making use of the MPI_GET_PROCESSOR_NAME procedure:

The host name of each MPI node is sent to every other node. If the incoming name matches the local name, the remote processor id is added to a local list. This list is then locally sorted and is used to assign a unique device id to the MPI nodes on the same host.

Evaluating New Particle-in-Cell (PIC) Algorithms on GPU: **Electrostatic Case**
2D ES Benchmark with 2048x2048 grid, 150,994,944 particles, 36 particles/cell
optimal block size = 128, optimal tile size = 16x16. Single precision. Fermi M2090 GPU

Hot Plasma results with dt = 0.1

	CPU:Intel i7	1 GPU	24 GPUs	108 GPUs
Push	22.1 ns.	0.327 ns.	13.4 ps.	3.46 ps.
Deposit	8.5 ns.	0.233 ns.	11.0 ps.	2.60 ps.
Reorder	0.4 ns.	0.442 ns.	19.7 ps.	5.21 ps.
Total Particle	31.0 ns.	1.004 ns.	49.9 ps.	13.10 ps.

The time reported is per particle/time step.
The total particle speedup on the 108 Fermi M2090s compared to 1 GPU was 77x,

Field solver takes an additional 5% on 1 GPU, 45% on 2 GPUs, and 73% on 108 GPUs

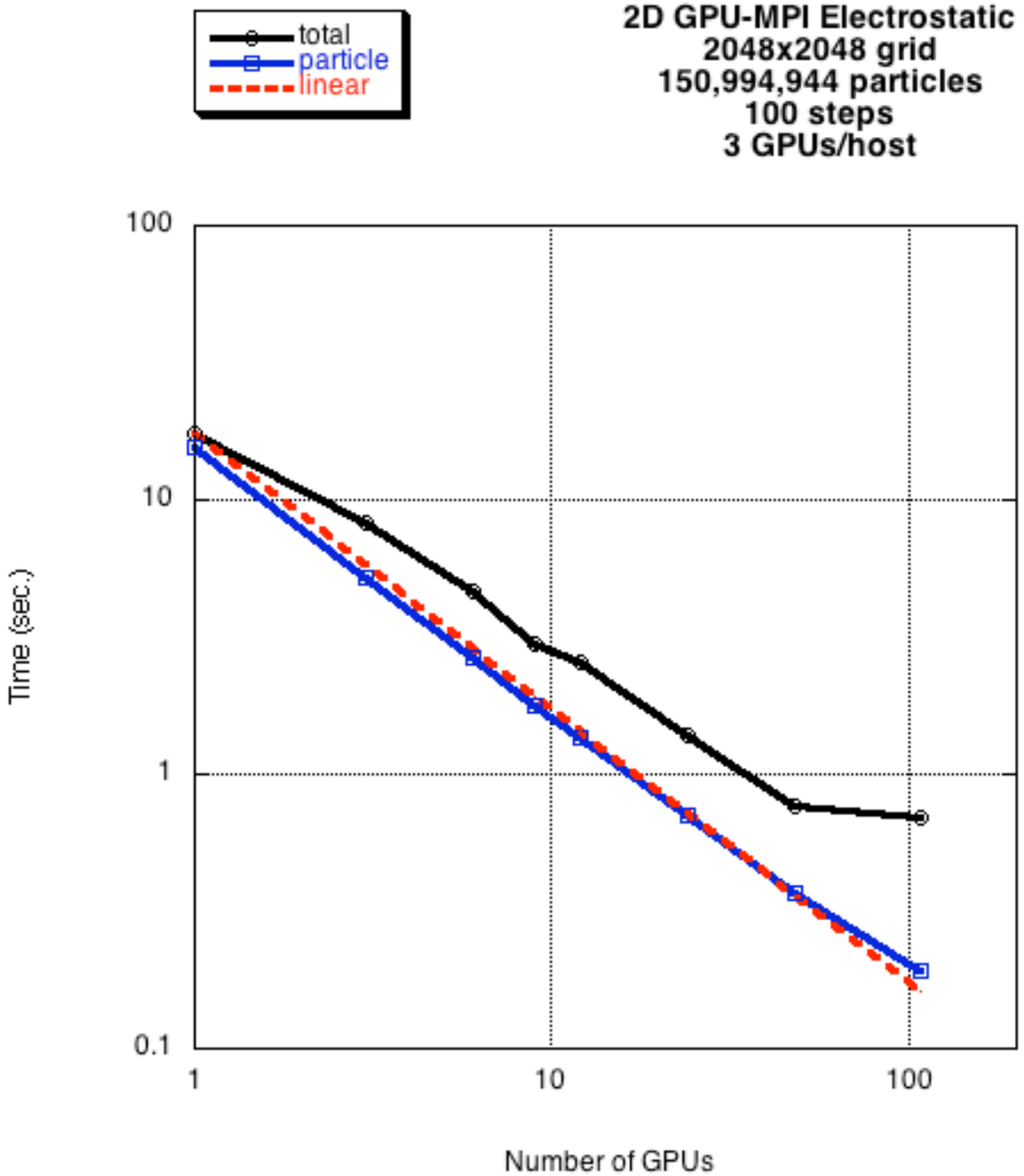
Evaluating New Particle-in-Cell (PIC) Algorithms on GPU: **Electromagnetic Case**
2-1/2D EM Benchmark with 2048x2048 grid, 150,994,944 particles, 36 particles/cell
optimal block size = 128, optimal tile size = 16x16. Single precision. Fermi M2090 GPU

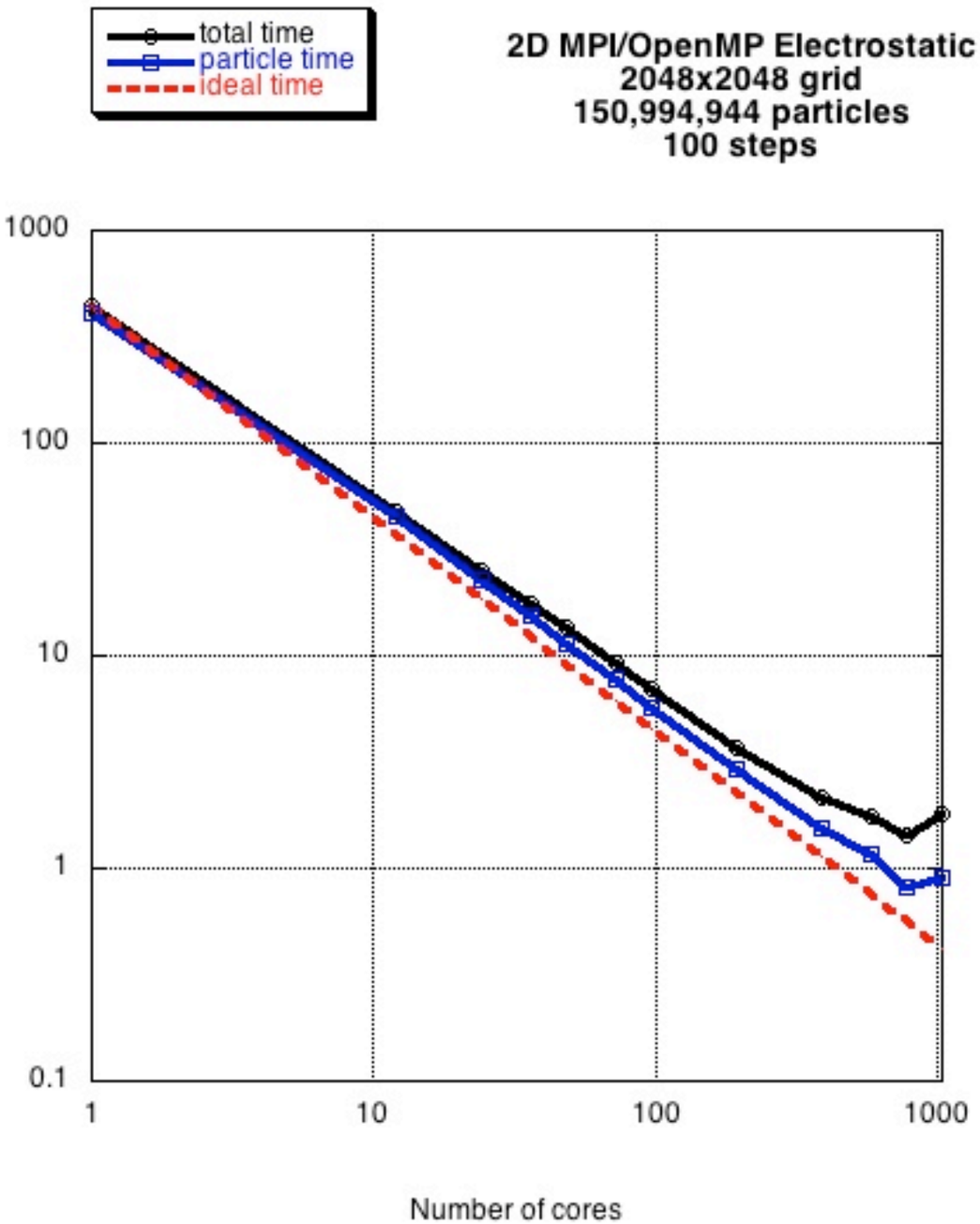
Hot Plasma results with dt = 0.04, c/vth = 10, relativistic

	CPU:Intel i7	1 GPU	24 GPUs	108 GPUs
Push	66.5 ns.	0.422 ns.	18.3 ps.	4.26 ps.
Deposit	36.7 ns.	0.972 ns.	43.8 ps.	11.31 ps.
Reorder	0.4 ns.	0.690 ns.	31.2 ps.	8.33 ps.
Total Particle	103.6 ns.	2.092 ns.	95.5 ps.	29.60 ps.

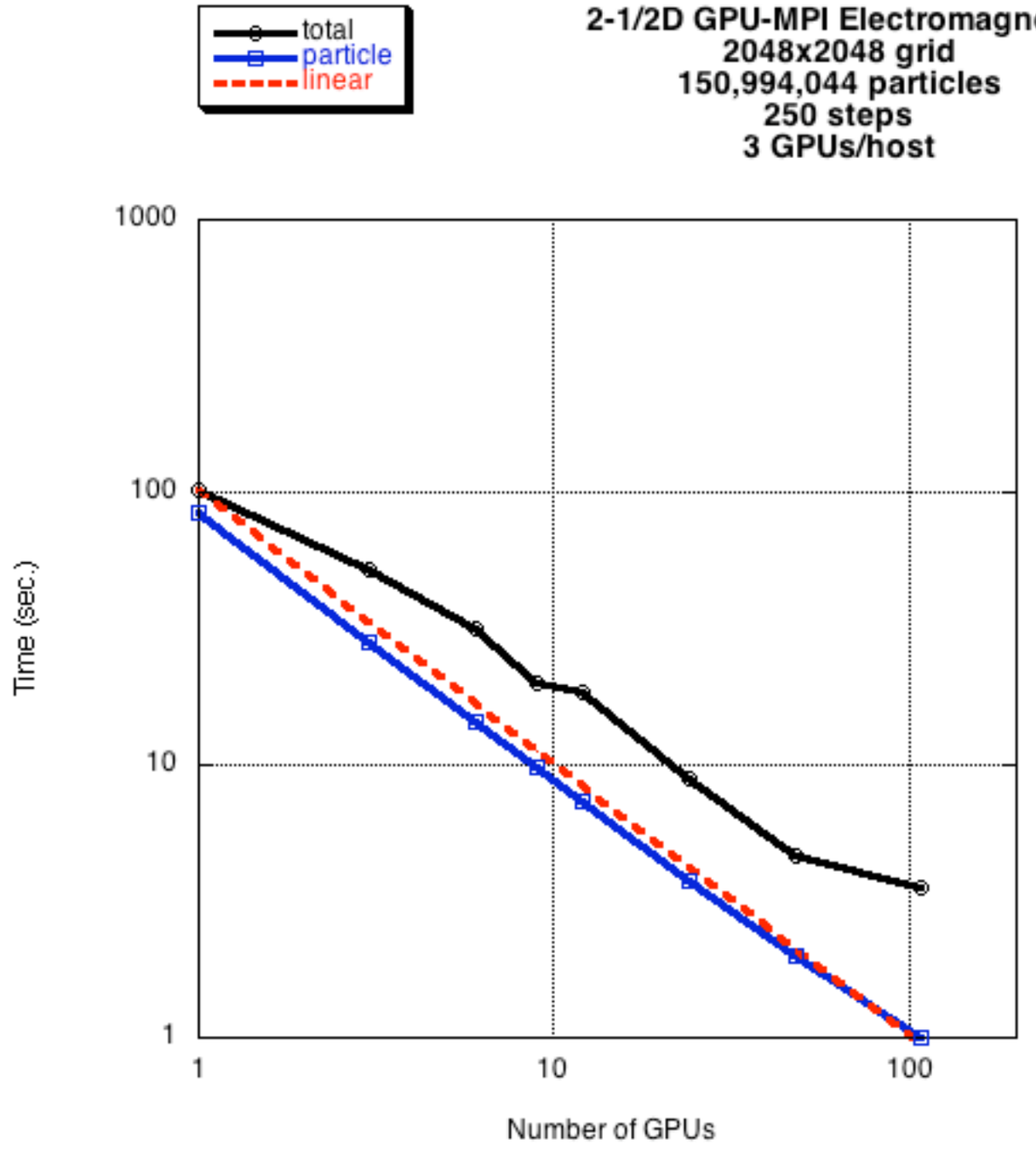
The time reported is per particle/time step.
The total particle speedup on the 108 Fermi M2090s compared to 1 GPU was 71x,

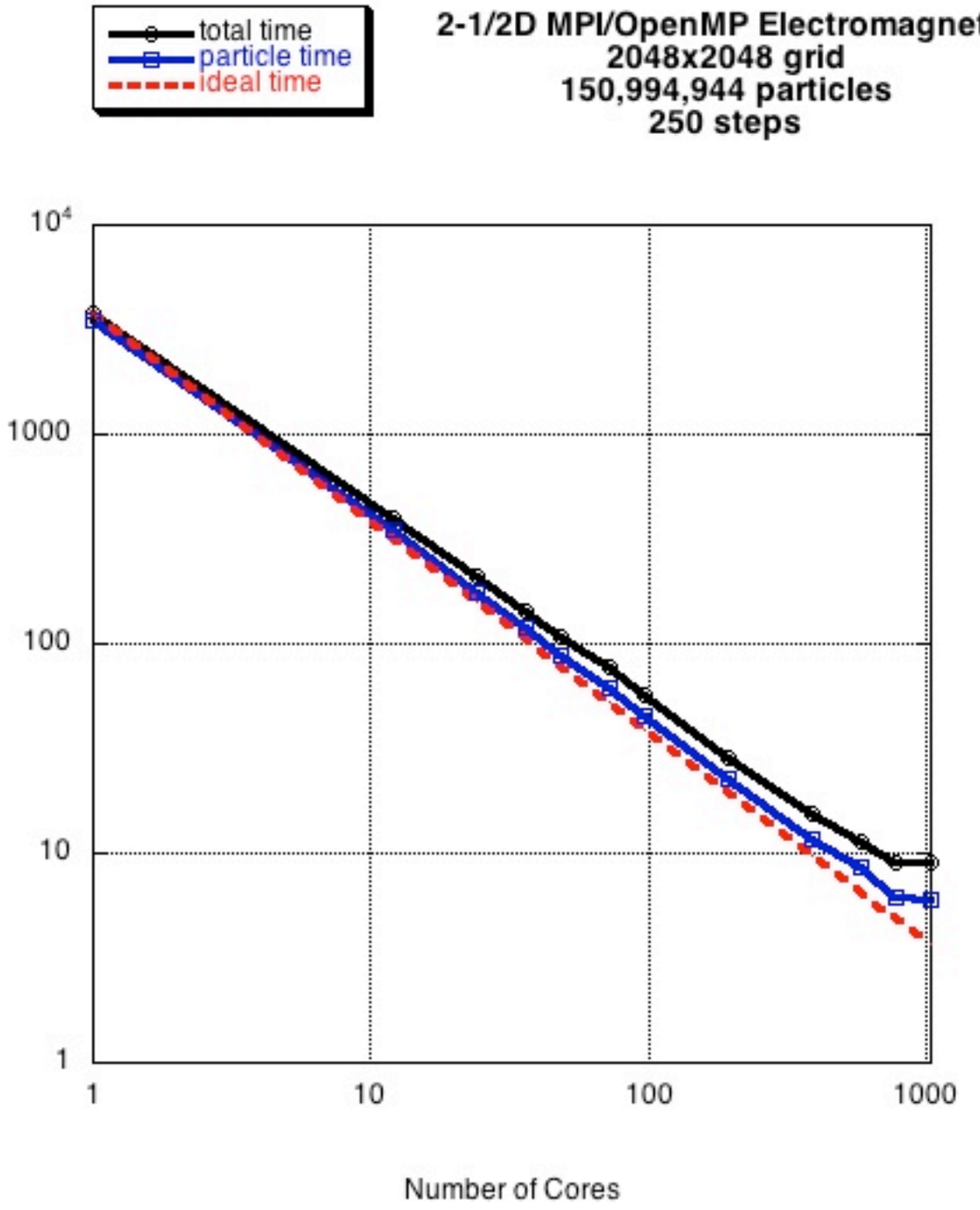
Field solver takes an additional 10% on 1 GPU, 62% on 24 GPUs, and 74% on 108 GPUs





Performance as a function of the number of GPUs (Left). Dashed red line is ideal scaling, blue shows particle time, black shows total time. Degradation of total time is due to the all to all transpose in the FFT, which dominates. The FFT takes about 30-70% of the total time. The MPI/OpenMP code (Right) uses the same algorithm as the GPU code.





Performance as a function of the number of GPUs (Left). Dashed red line is ideal scaling, blue shows particle time, black shows total time. Degradation of total time is due to the all to all transpose in the FFT, which dominates. The FFT takes about 40-70% of the total time. The MPI/OpenMP code (Right) uses the same algorithm as the GPU code.

Conclusions

PIC Algorithms on emerging architectures are largely a combination of previous techniques

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

- Programming to Hardware Abstraction leads to common algorithms
- Streaming algorithms optimal for memory-intensive applications

Scheme should be portable to architectures with similar hardware abstractions

Further information available at:

V. K. Decyk and T. V. Singh, "Particle-in-Cell Algorithms for Emerging Computer Architectures," Computer Physics Communications, 185, 708, (2014), available at http://dx.doi.org/10.1016/j.cpc.2013.10.013.

http://www.idre.ucla.edu/hpc/research/

Source codes available at:
https://idre.ucla.edu/hpc/parallel-plasma-pic-codes/