GPU as a Prototype for Next Generation Supercomputer

Viktor K. Decyk and Tajendra V. Singh
UCLA

Abstract

The next generation of supercomputers will likely consist of a hierarchy of parallel computers. If we can define each node as a parameterized abstract machine, then it is possible to design algorithms even if the actual hardware varies. Such an abstract machine is defined by the OpenCL language to consist of a collection of vector (SIMD) processors, each with a small shared memory, communicating via a larger global memory. This abstraction fits a variety of hardware, such as Graphical Processing Units (GPUs), and multi-core processors with vector extensions. To program such an abstract machine, one can use ideas familiar from the past: vector algorithms from vector supercomputers, blocking or tiling algorithms from cache-based machines, and domain decomposition from distributed memory computers. Using the OpenCL language itself is not necessary. Examples from our GPU Particle-in-Cell code will be shown to illustrate this approach.

Outline of Presentation

- Abstraction of future computer hardware
- Short Tutorial on Cuda Programming
- Particle-in-Cell Algorithm for GPUs
- Performance Results on GPU
- Other Parallel Languages

Revolution in Hardware

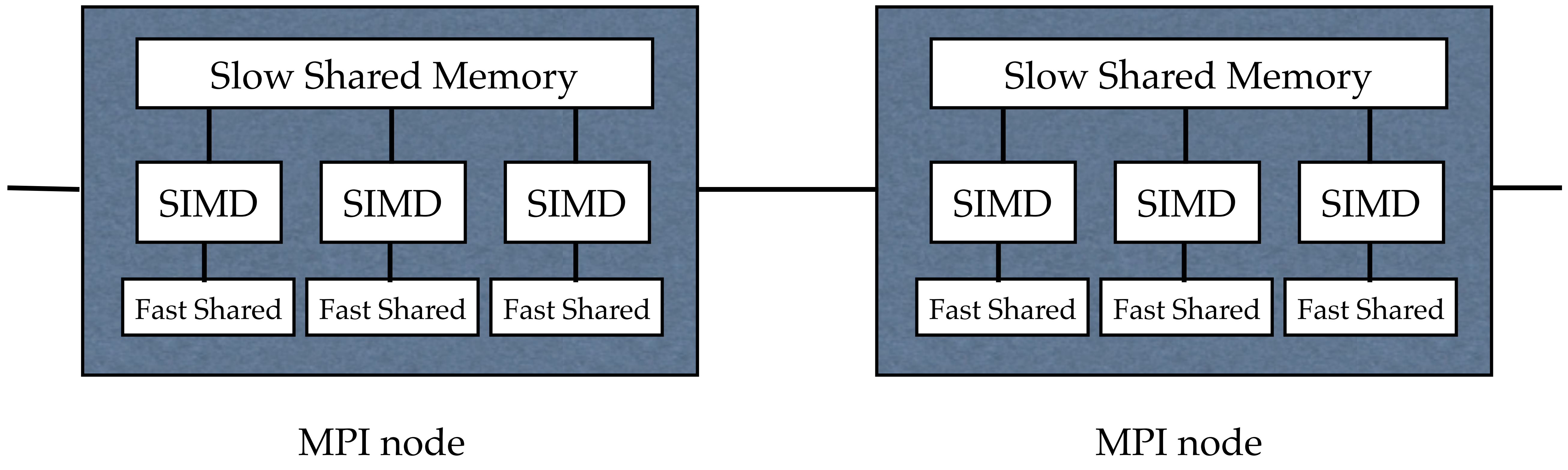
Many new architectures
- Multi-core processors
- SIMD accelerators, GPUs
- Low power embedded processors, FPGAs
- Heterogeneous processors (co-design)

Driven by:
- Increasing shared memory parallelism, since clock speed is not increasing
- Low power computing is now a requirement

How does one cope with this variety?

In the midst of a revolution, it is hard to know which path is best

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

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

GPUs are graphical processing units which 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)
- Ability to handle thousands of threads simultaneously, greatly reducing memory “stalls”

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

This hardware model (abstraction) 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

Intel MIC (Knights Corner)

- Vector Length = 16

Some convergence of architectures

- Intel increasing vector length, NVIDIA adding cache

Designing new algorithms for next generation computers

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

But some elements are unfamiliar**:**

Languages (CUDA, OpenCL, OpenACC) have new features

Programming GPUs

Programming Massively Parallel Processors: A Hands-on Approach
by David B. Kirk and Wen-mei W. Hwu [Morgan Kaufmann, 2010].

CUDA by Example: An Introduction to General-Purpose GPU Programming
Jason Sanders and Edward Kandrot [Addison-Wesley, 2011].

http://developer.nvidia.com/cuda-downloads

Programming GPUs with CUDA

Let’s consider a very simple example: adding two 1D arrays

```
   subroutine fadd(a,b,c,nx)
! add two vectors
   integer :: nx
   real, dimension(nx) :: a, b, c
   integer :: j
!$OMP PARALLEL DO PRIVATE(j)
   do j = 1, nx
      a(j) = b(j) + c(j)
   enddo
!$OMP END PARALLEL DO
   end subroutine
```

```
void cadd(float a[],float b[],float c[],int nx) {
   int j;
#pragma omp parallel for private(j)
   for (j = 0; j < nx; j++) {
      a[j] = b[j] + c[j];
   }
   return;
}
```

Programming GPUs: CUDA Fortran

The single subroutine with a loop is replaced by a pair of subroutines

The first subroutine contains the interior of the loop (kernel):

loop index j replaced by thread id

```
   attributes(global) subroutine gadd(a,b,c,nx)
! Vector add kernel
   integer, value :: nx
   real :: a(nx), b(nx), c(nx)
   integer :: j
   j = threadIdx%x+blockDim%x*(blockIdx%x-1)
   if (j <= nx) a(j) = b(j) + c(j)
   end subroutine
```

The second subroutine contains the loop parameters:

loop instruction replaced by procedure for launching kernels

```
   subroutine gpadd(a,b,c,nx)
! Vector add interface
   integer :: nx
   real, device, dimension(:) :: a(nx), b(nx), c(nx)
   integer :: nblock_size = 64, crc
   type (dim3) :: dimBlock, dimGrid
! set up loop parameters
   dimBlock = dim3(nblock_size,1,1)
   dimGrid = dim3((nx-1)/nblock_size+1,1,1)
! launch kernel (loop interior)
   call gadd<<<dimGrid,dimBlock>>>(a,b,c,nx)
   crc = cudaThreadSynchronize()
   end subroutine
```

Programming GPUs: CUDA C

The single subroutine with a loop is replaced by a pair of subroutines

The first subroutine contains the interior of the loop (kernel):

```
__global__ void gadd(float a[],float b[],float c[],int nx) {
/* Vector add kernel */
   int j;
   j = threadIdx.x+blockDim.x*blockIdx.x;
   if (j < nx)
      a[j] = b[j] + c[j];
   return;
}
```

The second subroutine contains the loop parameters:

```
extern "C" void gpadd(float a[],float b[],float c[],int nx) {
/* Vector add Interface */
   int nblock_size = 64;
/* set up loop parameters */
   dim3 dimBlock(nblock_size);
   dim3 dimGrid((nx - 1)/nblock_size + 1);
/* launch kernel (loop interior) */
   gadd<<<dimGrid,dimBlock>>>(a,b,c,nx);
   cudaThreadSynchronize();
   return;
}
```

Programming GPUs: CUDA Fortran

In addition, you must initialize memory on the GPU

```
   real, dimension(:), allocatable :: a, b, c
   real, device, dimension(:), allocatable :: g_a, g_b, g_c
! initialize host data
   allocate(a(nx),b(nx),c(nx))
! allocate data on GPU, using Fortran90 array syntax
   allocate(g_a(nx),g_b(nx),g_c(nx))
```

Copy from host to GPU

```
 ! Copy data to GPU, using Fortran90 array syntax
    g_b = b; g_c = c
```

Execute the subroutine:

```
! Execute on GPU: g_a = g_b + g_c
      call gpadd(g_a,g_b,g_c,nx)
```

Copy from GPU back to host

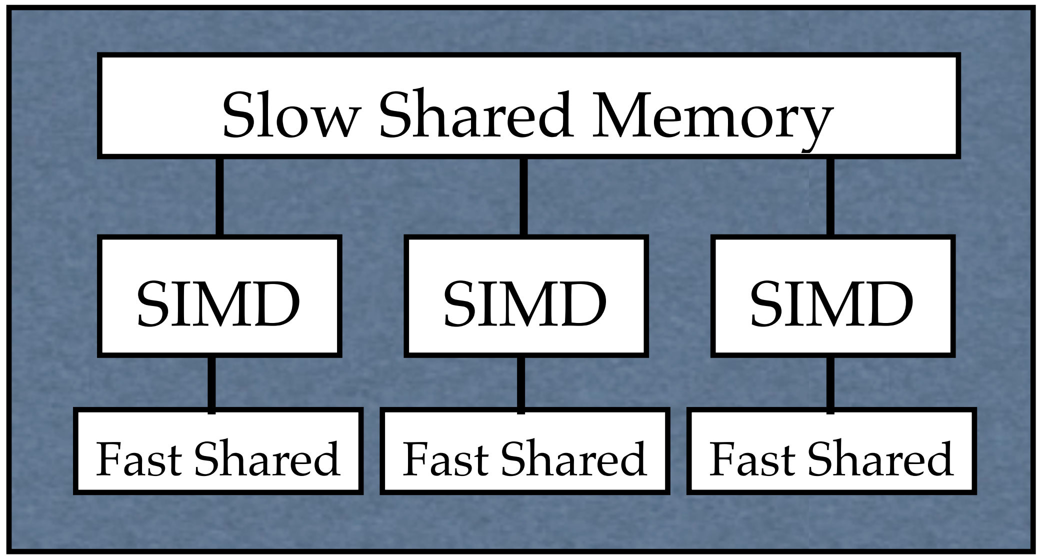
```
! Copy data from GPU, using Fortran90 array syntax
   a = g_a
```

CUDA C is similar but more complex (no array syntax, separate memory spaces)

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

Vector algorithms

In PIC code, often appears in field solvers

- Stride 1 memory access optimal (to avoid memory bank conflicts)

Here is a code fragment of a vectorized Poisson solver, $\mathbf{k}\bullet\mathbf{E} = 4\pi qn(k)$

- Usually inner loops were vectorized

```
complex qt(nyv,nxvh), fxyt(nyv,2,nxvh), ffct(nyhd,nxhd)

do j = 2, nxh
   dkx = dnx*real(j - 1)
   do k = 2, nyh
      k1 = ny2 - k
      at1 = real(ffct(k,j))*aimag(ffct(k,j))
      at2 = dkx*at1
      at3 = dny*real(k - 1)*at1
      zt1 = cmplx(aimag(qt(k,j)),-real(qt(k,j)))
      zt2 = cmplx(aimag(qt(k1,j)),-real(qt(k1,j)))
      fxyt(k,1,j) = at2*zt1
      fxyt(k,2,j) = at3*zt1
      fxyt(k1,1,j) = at2*zt2
      fxyt(k1,2,j) = -at3*zt2
   enddo
enddo
```

Vector algorithms

Here is a code fragment of a Poisson solver written for CUDA Fortran

- The changes were minimal: loop index replaced by thread ids
- If test to check loop bounds
- Both loops are running in parallel

```
k = threadIdx%x + blockDim%x*(blockIdx%x - 1)
j = blockIdx%y

dkx = dnx*real(j - 1)
if ((k > 1) .and. (k < nyh1) .and. (j > 1)) then
   k1 = ny2 - k
   at1 = real(ffct(k,j))*aimag(ffct(k,j))
   at2 = dkx*at1
   at3 = dny*real(k - 1)*at1
   zt1 = cmplx(aimag(qt(k,j)),-real(qt(k,j)))
   zt2 = cmplx(aimag(qt(k1,j)),-real(qt(k1,j)))
   fxyt(k,1,j) = at2*zt1
   fxyt(k,2,j) = at3*zt1
   fxyt(k1,1,j) = at2*zt2
   fxyt(k1,2,j) = -at3*zt2
endif
```

CUDA C does not natively support complex arithmetic

Blocking (tiling) algorithms

- Stride 1 memory access optimal (to avoid reading many cache lines)

Here is a code fragment of a complex transpose, used by FFT

```
complex f(nxv,ny), g(nyv,nx)
do k = 1, ny
   do j = 1, nx
      g(k,j) = f(j,k)
   enddo
enddo
```

f is read with optimal stride 1,
but g is written with large jumps in memory

Break up loops into blocks, where block is typically size of L1 cache

```
   complex f(nxv,ny), g(nyv,nx), buff(nblok+1,nblok)
! loop over tiles
   do kb = 1, ny/nblok
      koff = nblok*(kb - 1)
      do jb = 1, nx/nblok
         joff = nblok*(jb - 1)
! copy data into tile, with optimal stride
         do k = 1, nblok
            do j = 1, nblok
               buff(j,k) = f(j+joff,k+koff)
            enddo
         enddo
! copy data out of tile with optimal stride
         do j = 1, nblok
            do k = 1, nblok
               g(k+koff,j+joff) = buff(j,k)
            enddo
         enddo
      enddo
   enddo
```

f , g are accessed with optimal stride 1,
buff is accessed in fast memory with
smaller jumps in memory

Blocking (tiling) algorithms

Here is a code fragment of a complex transpose written for CUDA C

- Changes were minimal: loop index replaced by thread ids
- If test to check loop bounds
- Synchronization needed

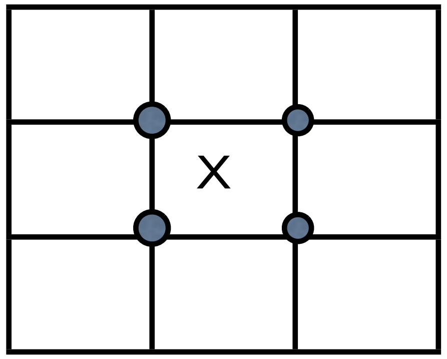
```
      extern __shared__ cuComplex buff[];

      koff = nblok*blockIdx.y;
      joff = nblok*blockIdx.x;
   /* copy data into tile, with optimal stride */
      k = threadIdx.y;
      ky = k + koff;
      j = threadIdx.x;
      jx = j + joff;
      if ((jx < nx) && (ky < ny)) {
         buff[j+(nblok+1)*k] = f[jx+nxv*ky)];
       }
   /* make sure all threads are done */
       __syncthreads();
   /* copy data out of tile with optimal stride */
      j = threadIdx.y;
      jx = j + joff;
      k = threadIdx.x;
      ky = k + koff;
      if ((jx < nx) && (ky < ny)) {
         g[ky+nyv*jx] = buff[j+(nblok+1)*k];
      }
```

Domain decomposition algorithms
- Partition memory so different threads work on different data

Parallel Charge Deposit in PIC code

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 collision**.



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 (domain decomposition).
- This avoids any data hazards, widely used in distributed memory computers

Domain decomposition algorithms

- Useful when different threads might write to same memory locations

The original charge deposit loop in 2D

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

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:

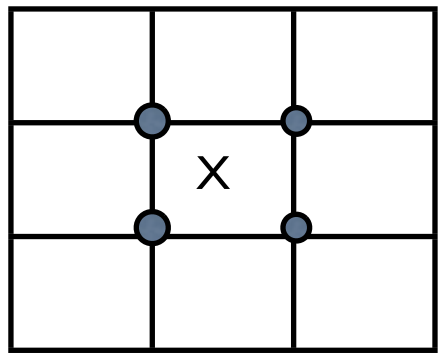
$$\nabla \bullet \boldsymbol{E} = 4\pi\rho$$

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.



GPU Programming for PIC

GPUs require many more threads than physical processors

- Very different from traditional MPI programming
- Hides memory latency
- Hardware uses master-slave model for automatic load balancing among blocks

Optimizing data movement is very critical

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

- Particles reordered every time step to within small tiles
- Particle data is read/written once
- Field data is read/written once and partitioned to fit within small tiles
- Charge deposit can avoid data collisions
- Memory access is now regular

Reordering particles efficiently was largest challenge

- Memory access not regular, but few particles need to be reordered

GPU Programming for PIC: Maintaining Particle Order is Main Challenge

Stream compaction: reorder elements into substreams of like elements

- Less than a full sort, low overhead if already ordered

Two cases

- 1: Particle moves to another thread in same vector or thread block
  Use fast memory
- 2: Particle moves to another thread in a different vector or thread block
  Essentially message-passing, except buffer contains multiple destinations

The reordering algorithm does not match the architecture well

- Does not have stride 1 access (CUDAspeak: poor data coalescing)
- Does not run in lockstep (CUDAspeak: has warp divergence)
- But few particles need to be reordered

GPU Programming for PIC

Algorithm is hierarchical (layered)
- Lowest level is vector (thread block), same instruction on all units
- Next higher level is GPU kernels (more threads than cores)
- Multiple GPUs (MPI and/or OpenMP)

Algorithm is adaptable to match architecture
- Vector length (thread block size) adjustable
- Size of tiles adjustable

Reference:

V. K. Decyk and T. V. Singh, "Adaptable Particle-in-Cell Algorithms for Graphical Processing Units," Computer Physics Communications, 182, 641, 2011.

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

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 vector (thread block) size = 64, and
On Fermi C2050, tile size: mx = 2, my = 2
On Tesla C1060 and GTX280, mx = 1, my = 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 i7 CPU   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 i7 CPU   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.
```

Other parallel languages: OpenCL

OpenCL is a portable, parallel language
- Kernel procedures map easily between Cuda C and OpenCL.
- Kernel procedures have to be strings, compiled at run time
- Host functions very different, more complex
- Initializing OpenCL: 225 lines of code!

Performance with NVIDIA's CUDA C and OpenCL
- OpenCL about 23% slower than CUDA C
- Overhead in launching kernels about 2-6 x higher in OpenCL than CUDA C

Performance on NVIDIA GPU with Apple OpenCL
- Apple's OpenCL about 60% slower than CUDA C

Performance on AMD GPU with AMD OpenCL
- AMD HD5870 with OpenCL about 20% slower than NVIDA C1060 with Cuda C
- AMD GPU has much less shared memory
- AMD GPU did not allow an array larger than about 134 MB

Conclusion: OpenCL is portable, but painful and slow

Other parallel languages: CUDA Fortran from PGI

CUDA Fortran somewhat easier to use than CUDA C
- Host and GPU memory only differ by attribute
- Copying an array from host to GPU or back is just an assignment.
- Limitation: dynamic shared memory must all be same type

Performance on NVIDIA GPU with CUDA C and CUDA Fortran
- CUDA Fortran about 9% slower than CUDA C
- CUDA's FFT can be called from CUDA Fortran using ISO_C_BIND (F2003 feature)
- Overhead in launching kernels is the same in CUDA Fortran and CUDA C

CUDA Fortran is somewhat easier to use than CUDA C, and gives good performance
- Very strict with types

Other parallel languages: OpenMP

OpenMP is a portable, parallel language for shared memory processors

- Cannot handle the hierarchical structures needed for GPUs (e.g., local synchronization)

Other parallel languages: OpenACC

- Compiler directives in style of OpenMP, but can support GPUs

Parallel languages are still evolving

Conclusions

PIC Algorithms on GPUs 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

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

Scheme should be portable to other architectures with similar hardware abstractions
- OpenCL version allows the code to run on AMD GPU and Intel Multicore

Hierarchical (Nested) Domain Decompositions look promising for GPU cluster
- Also improves performance on shared memory/distributed memory architectures

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

We do not plan to wait 10 years to start developing algorithms for the next generation supercomputer

Dawson2 at UCLA: 96 nodes, ranked 235 in top 500, 70 TFlops on Linpack

- Each node has: 12 Intel G7 X5650 CPUs and 3 NVIDIA M2090 GPUs.
- Each GPU has 512 cores: total GPU cores=147,456 cores, total CPU cores=1152