Particle-in-Cell Simulations on Modern Computing Platforms

Viktor K. Decyk and Tajendra V. Singh

UCLA
Outline of Presentation

- Abstraction of future computer hardware
- PIC on GPUs
- OpenCL and Cuda Fortran
- OpenMP
- Hybrid MPI/OpenMP
- GPU and MPI
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 prosperity?
Unrest in Parallel Software

Two common 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
• Ease of programming a major concern

Can we avoid different programming models for different hardware?
How will it all turn out?
Coping Strategy: Program to Simple Hardware Abstraction with 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
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
This abstract 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
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: GPU

GPU PIC Algorithm
- Particles kept ordered each time step by use of sorting cells
- Sorting cell can contain multiple grids
- Adaptable with 4 adjustable parameters to match architecture
- Domain decomposition used to avoid data dependencies

First written in OpenMP (loop) style, in both Fortran and C.

Minimal changes in translating to CUDA
- 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 host functions which invoke kernel subroutine on GPU

Reference:


See also: http://www.idre.ucla.edu/hpc/research/
Evaluating New Particle-in-Cell (PIC) Algorithms on GPU: Electromagnetic Case

Warm Plasma results with $c/v_{th} = 10$, $dt = 0.04$

<table>
<thead>
<tr>
<th></th>
<th>Intel Nehalem</th>
<th>Fermi C2050</th>
<th>Tesla C1060</th>
<th>GTX 280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push</td>
<td>81.7 ns.</td>
<td>0.89 ns.</td>
<td>1.13 ns.</td>
<td>1.08 ns.</td>
</tr>
<tr>
<td>Deposit</td>
<td>40.7 ns.</td>
<td>0.78 ns.</td>
<td>1.06 ns.</td>
<td>1.04 ns.</td>
</tr>
<tr>
<td>Reorder</td>
<td>0.5 ns.</td>
<td>0.57 ns.</td>
<td>1.13 ns.</td>
<td>0.97 ns.</td>
</tr>
<tr>
<td>Total Particle</td>
<td>122.9 ns.</td>
<td>2.24 ns.</td>
<td>3.32 ns.</td>
<td>3.09 ns.</td>
</tr>
</tbody>
</table>

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

Cold Plasma (asymptotic) results with $v_{th} = 0$, $dt = 0.025$

<table>
<thead>
<tr>
<th></th>
<th>Intel Nehalem</th>
<th>Fermi C2050</th>
<th>Tesla C1060</th>
<th>GTX 280</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push</td>
<td>78.5 ns.</td>
<td>0.51 ns.</td>
<td>0.79 ns.</td>
<td>0.74 ns.</td>
</tr>
<tr>
<td>Deposit</td>
<td>37.3 ns.</td>
<td>0.58 ns.</td>
<td>0.82 ns.</td>
<td>0.81 ns.</td>
</tr>
<tr>
<td>Reorder</td>
<td>0.4 ns.</td>
<td>0.10 ns.</td>
<td>0.16 ns.</td>
<td>0.15 ns.</td>
</tr>
<tr>
<td>Total Particle</td>
<td>116.2 ns.</td>
<td>1.20 ns.</td>
<td>1.77 ns.</td>
<td>1.70 ns.</td>
</tr>
</tbody>
</table>

The time reported is per particle/time step.
The total speedup on the Fermi C2050 was 97x,
on the Tesla 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
• no FFT available on NVIDIA’s OpenCL (solver turned off in this test)
• 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
• FFT available from Apple’s Web site, but only works on Apple computers

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

OpenCL is portable, but painful and slow
• Will be future be OpenCL or CUDA, or something else?

CUDA Fortran is somewhat easier to use than CUDA C, and gives good performance
• Very strict with types
Shared Memory Programming for PIC

Multi-core architectures have increasing numbers of CPUs

New shared memory parallel languages are being developed by computers scientists
  • Computer Scientists are often more interested in ease of programming than performance

Is OpenMP useful in this environment?
  • Historically, MPI has usually outperformed OpenMP
  • OpenMP was usually limited to shared memory machines
  • OpenMP does not have enough features to program GPUs

So why bother with OpenMP?
  • It is a simple language
  • A shared memory PIC code in OpenMP can be used as a starting point for developing PIC codes in other shared memory languages
OpenMP Programming for PIC

Domain decomposition is useful in shared memory architectures
• Avoids data dependencies
• Partitioning data gives better cache performance
• Domain decomposition is scalable
• Proved to be useful on GPUs

We decided to implement in OpenMP, the same domain decomposition used by MPI

There are two styles of programming OpenMP:

Most common is the loop style

```cpp
!$OMP PARALLEL DO
!$OMP& PRIVATE(l,n,m,...)
!$OMP& REDUCTION(+:ke)
    do l = 1, nblok
    ...
enddo
!$OMP END PARALLEL DO
```

In implementing domain decomposition in OpenMP, we set the loop index nblok = nproc
OpenMP Programming for PIC: loop style

In the MPI code, we have a communications utility library with 4 major procedures:
• Add/copy guard cells
• Transpose (used by FFT)
• Particle manager (to move particles to correct processor)
• Field manager (to move field elements between uniform/non-uniform partitions)

The remaining procedures do not use communications:
• Push, deposit, field solver, FFT

Implementing non-communicating procedures in OpenMP was easy
• Add a do loop and an additional index in the all the distributed arrays

In implementing the communications library, new, simpler algorithms were developed
• Processors could directly read the data they need without receiving a message

Some of the lessons learned:
• OpenMP can be tricky: race conditions are easy to trigger
• Be very careful about declaring private data (data which is replicated inside the loop)
OpenMP Programming for PIC: SPMD style

The other style of programming OpenMP is the SPMD style
• Described in Chapman, Jost, and van de Pas, Using OpenMP

In this style, the entire program is a parallel region, but data is still partitioned
• The main program has the declaration:

```
!$OMP PARALLEL DEFAULT(none)
!$OMP& SHARED(part,qe,fxye,,,
!$OMP& PRIVATE(j,l,np,nx,ny,...)
!$OMP& FIRSTPRIVATE(idimp,ipbc,ntpose,...)
    ...
!$OMP END PARALLEL
```

Main program passes a different partition of global arrays to each procedure

Non-communicating procedures look like the MPI procedures
• No parallel loops, no partitioning

Communicating procedures look like OpenMP procedures, except
• omp_get_thread_num() procedure used instead of parallel loop
• barriers, single and critical regions are added if needed
• looks somewhat like GPU programming with CUDA
Data Shows OpenMP can be competitive with MPI, but somewhat slower

SPMD style slightly faster than loop style

Lessons learned:
- OpenMP has more synchronization points than MPI, all synchronizations are global
- Communication procedures are simpler in OpenMP
- SPMD style had fewer synchronization points than loop style
Hybrid MPI/OpenMP Programming for PIC

Used Nested domain decompositions
- Each shared memory node has its own domain
- Edges of shared memory domain define MPI domain

Allows fine grained partitions with reduced communications requirements
- Prototype for GPU-MPI decomposition
FFT gives nearly 3 times better performance with MPI-OpenMP than MPI alone.

Overall, spectral code gets more than 4 times better performance.
Application to GPU

Modularize the 4 hybrid MPI-OpenMP communicating procedures
• Isolate OpenMP calls into separate subroutines
• Replace OpenMP subroutines with CUDA

With replaceable modules, main code can be used with either GPUS or CPUs
• OpenMP subroutines can also be replaced with MPI-2, pthreads, ...
Dawson2 at UCLA: 96 nodes, ranked 148 in top 500, 68 TFlops on Linpack
- Each node has: 12 Intel G7 X5650 CPUs and 3 NVIDIA M2070 GPUs.
- Each GPU has 448 cores: total GPU cores=129,024 cores, total CPU cores=1152
Conclusions

- Programming to Hardware Abstraction leads to common algorithms
- Streaming algorithms optimal for memory-intensive applications
- OpenCL portable, but painful and slow
- Hybrid MPI/OpenMP useful for communication intensive algorithms
- Modular nested domain decompositions appear promising for GPU cluster