VSCSE Summer School 2008

Accelerators for Science and Engineering Applications: GPUs and Multi-cores

Lecture 8: Application Case Study – Accelerating Molecular Dynamics Experimentation

Acknowledgement

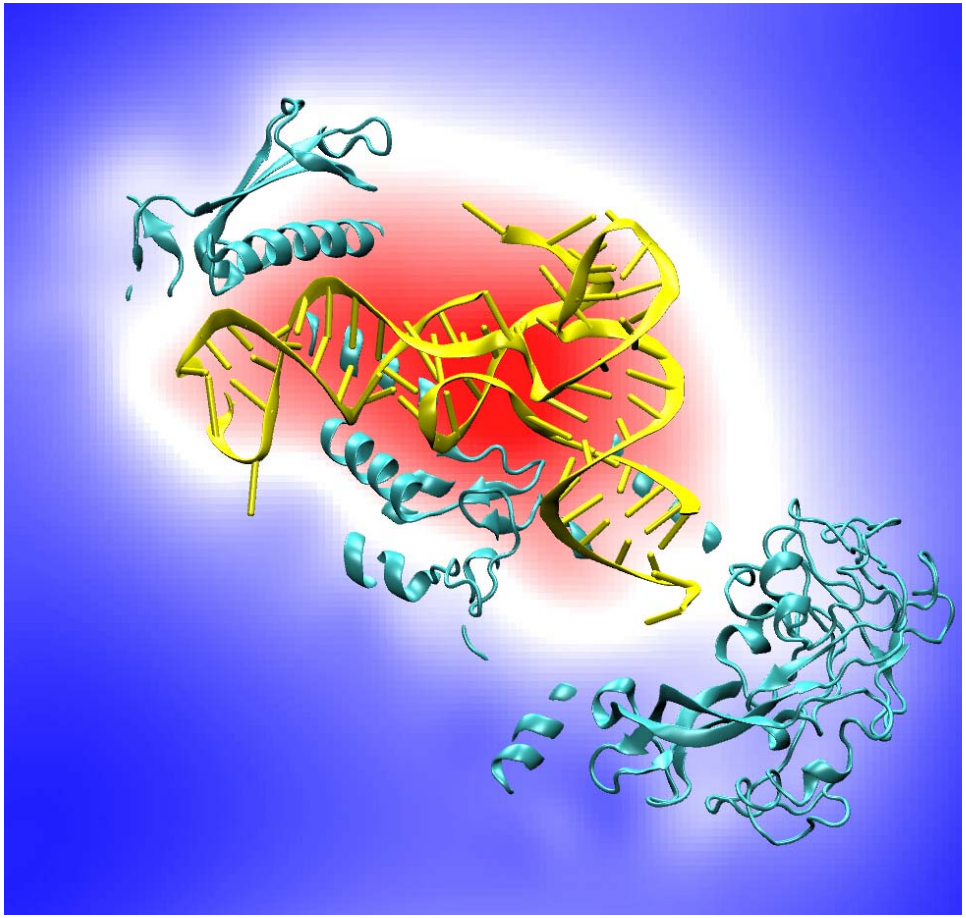
- Additional Information and References:
  - **http://www.ks.uiuc.edu/Research/gpu/**
- Acknowledgement, questions, source code requests:
  - John Stone (johns@ks.uiuc.edu)
  - Klaus Schulten, Jim Phillips, David Hardy
  - Theoretical and Computational Biophysics Group, NIH Resource for Macromolecular Modeling and Bioinformatics Beckman Institute for Advanced Science and Technology
- NIH support: P41-RR05969

Outline

- Explore CUDA algorithms for computing electrostatic fields around molecules
  - Detailed look at a few CUDA implementations of a simple direct Coulomb summation algorithm
  - Multi-GPU direct Coulomb summation
  - Cutoff (range-limited) summation algorithm
- CUDA acceleration of parallel molecular dynamics simulation on GPU clusters with NAMD

Calculating Electrostatic Potential Maps

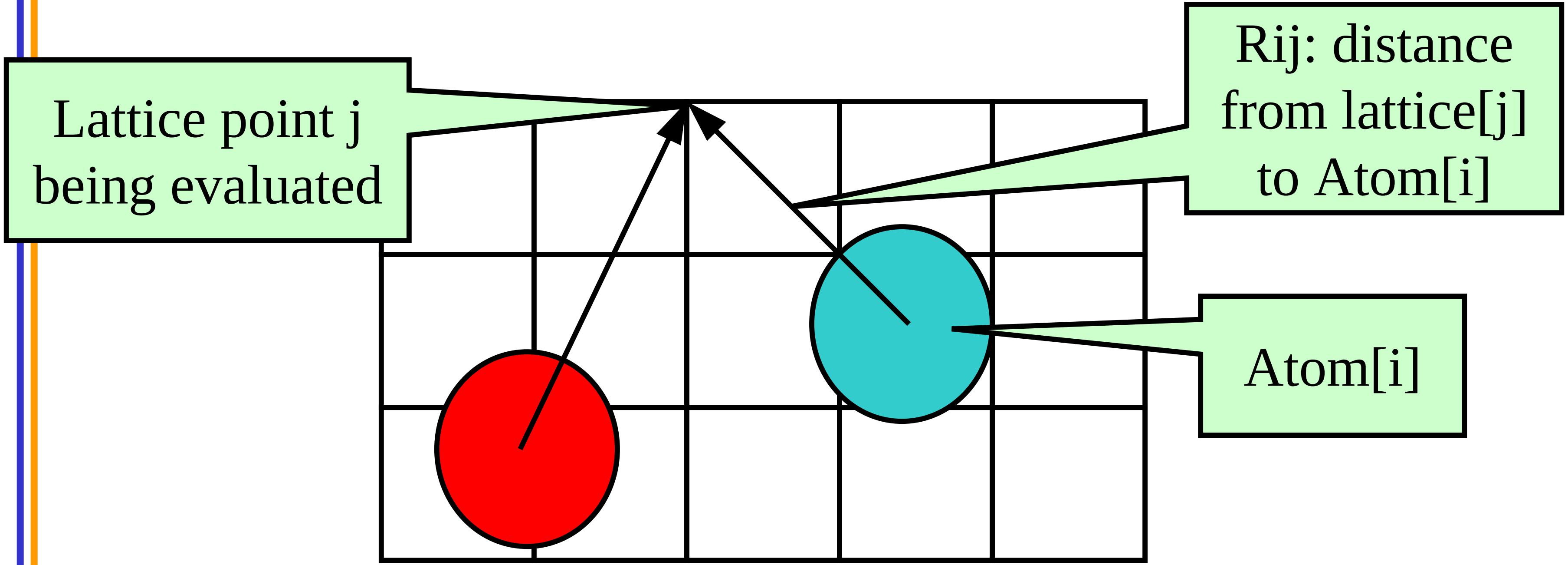
- Used in molecular structure building, analysis, visualization, simulation
- Electrostatic potentials evaluated on a uniformly spaced 3-D lattice
- Each lattice point contains sum of electrostatic contributions of all atoms



Direct Coulomb Summation

- At each lattice point, sum potential contributions for all atoms in the simulated structure:

  potential[j] += charge[i] / Rij



Lattice point j being evaluated

Rij: distance from lattice[j] to Atom[i]

Atom[i]

DCS Algorithm Design Observations

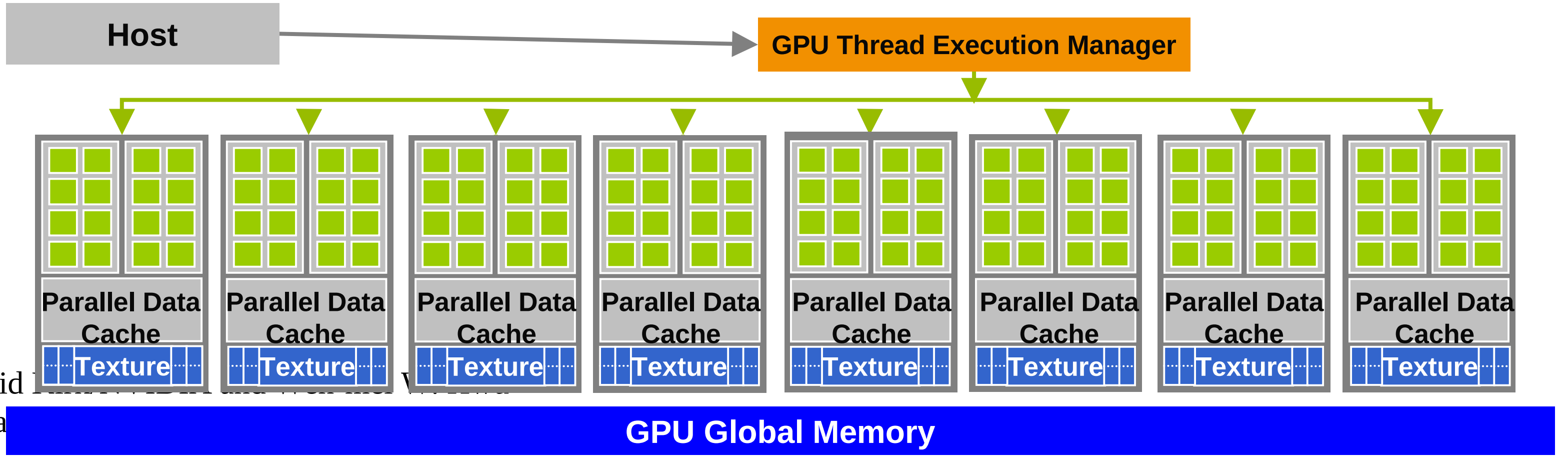
- Atom list has the smallest memory footprint, best choice for the inner loop (both CPU and GPU)
- Lattice point coordinates are computed on-the-fly
- Atom coordinates are made relative to the origin of the potential map, eliminating redundant arithmetic
- Arithmetic can be significantly reduced by precalculating and reusing distance components, e.g. create a new array containing X, Q, and dy^2 + dz^2, updated on-the-fly for each row (CPU)
- Vectorized CPU versions benefit greatly from SSE instructions

Single Slice DCS: Simple (Slow) C Version

```
void cenergy(float *energygrid, dim3 grid, float gridspacing, float z, const float *atoms, int
    numatoms) {
  int i,j,n;
  int atomarrdim = numatoms * 4;
  for (j=0; j<grid.y; j++) {
    float y = gridspacing * (float) j;
    for (i=0; i<grid.x; i++) {
      float x = gridspacing * (float) i;
      float energy = 0.0f;
      for (n=0; n<atomarrdim; n+=4) {     // calculate potential contribution of each atom
        float dx = x - atoms[n    ];
        float dy = y - atoms[n+1];
        float dz = z - atoms[n+2];
        energy += atoms[n+3] / sqrtf(dx*dx + dy*dy + dz*dz);
      }
      energygrid[grid.x*grid.y*k + grid.x*j + i] = energy;
    }
  }
}
```

Direct Coulomb Summation on the GPU

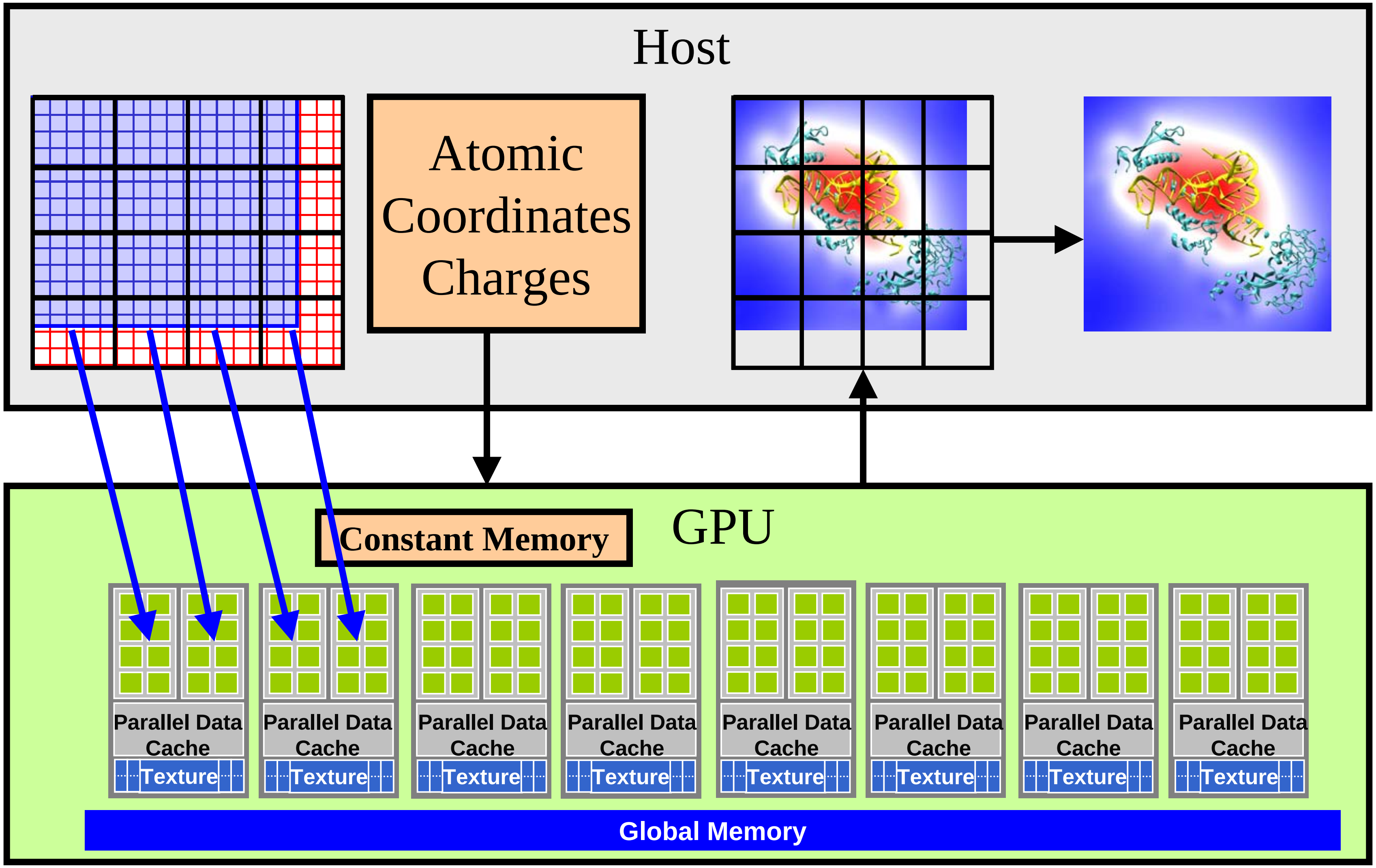
- GPU outruns a CPU core by 44x
- Work is decomposed into tens of thousands of independent threads, multiplexed onto hundreds of GPU processor cores
- Single-precision FP arithmetic is adequate for intended application
- Numerical accuracy can be further improved by compensated summation, spatially ordered summation groupings, or accumulation of potential in double-precision
- Starting point for more sophisticated algorithms



DCS Observations for GPU Implementation

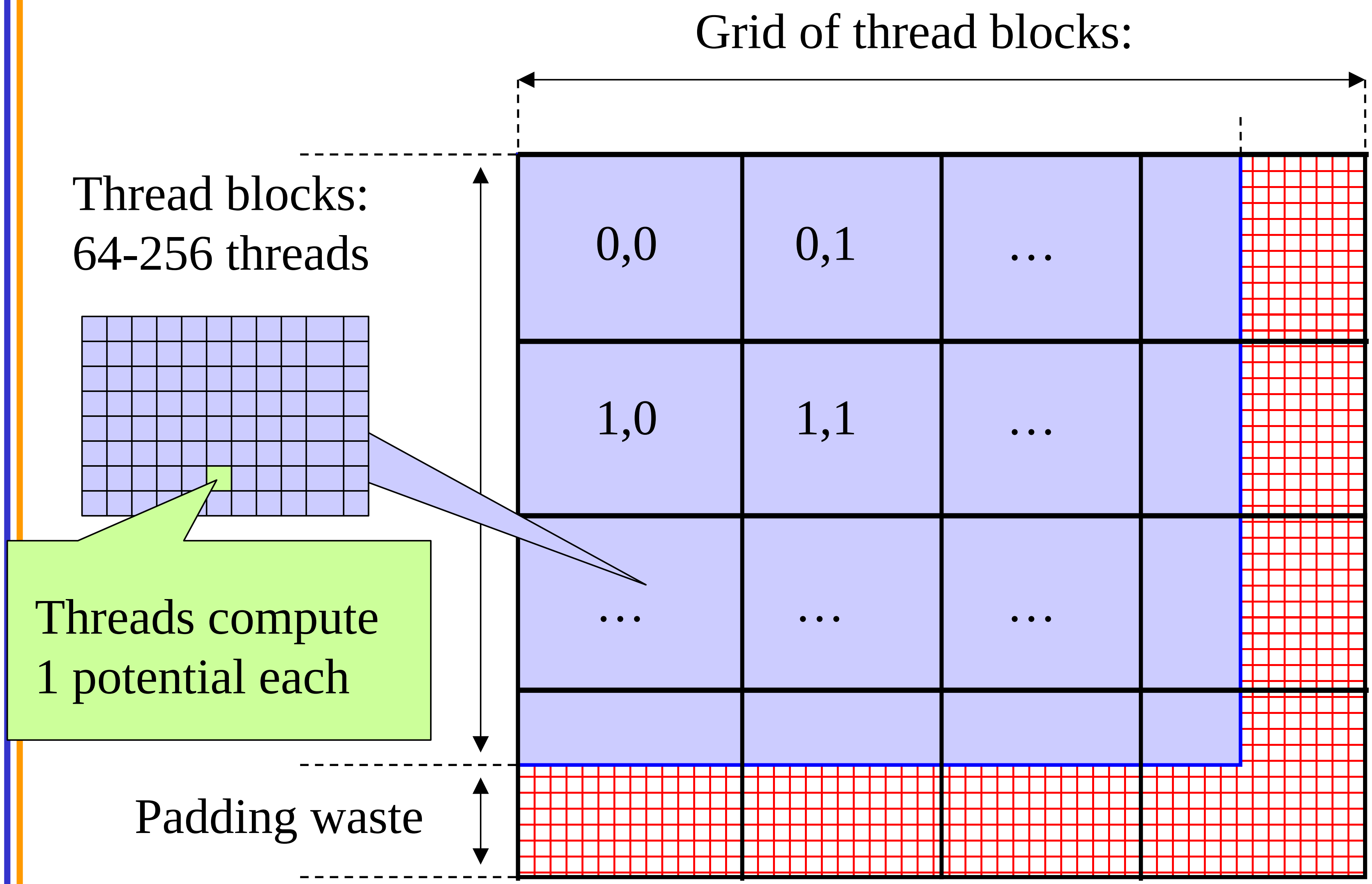
- Straightforward implementation has a low ratio of FLOPS to memory operations (for a GPU…)
- The innermost loop will consume operands VERY quickly
- Since atoms are read-only, they are ideal candidates for texture memory or constant memory
- GPU implementations must access constant memory efficiently, avoid shared memory bank conflicts, and overlap computations with global memory latency
- Map is padded out to a multiple of the thread block size:
  - Eliminates conditional handling at the edges, thus also eliminating the possibility of branch divergence
  - Assists with memory coalescing

Direct Coulomb Summation on the GPU



DCS CUDA Block/Grid Decomposition

(non-unrolled)



DCS Version 1: Const+Precalc 187 GFLOPS, 18.6 Billion Atom Evals/Sec

- Pros:
  - Pre-compute dz^2 for entire slice
  - Inner loop over read-only atoms, const memory ideal
  - If all threads read the same const data at the same time, performance is similar to reading a register
- Cons:
  - Const memory only holds ~4000 atom coordinates and charges
  - Potential summation must be done in multiple kernel invocations per slice, with const atom data updated for each invocation
  - Host must shuffle data in/out for each pass

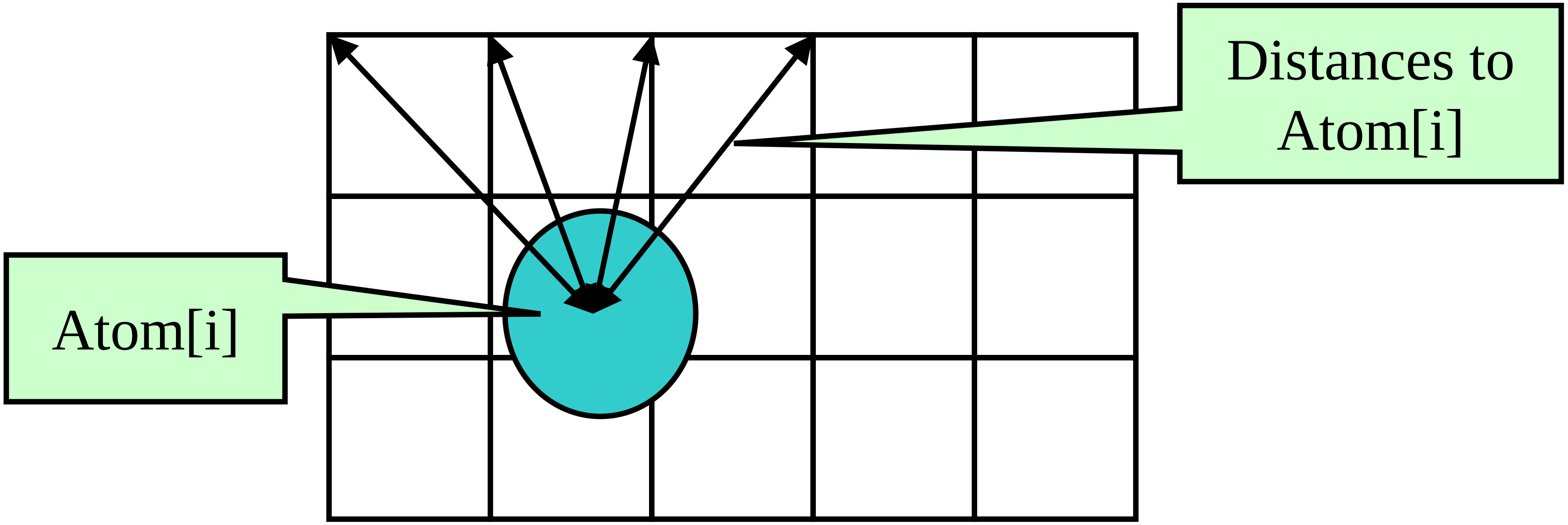
DCS Version 1: Kernel Structure

```
…
 float curenergy = energygrid[outaddr];  // start global mem read very early
 float coorx = gridspacing * xindex;
 float coory = gridspacing * yindex;
 int atomid;
 float energyval=0.0f;

 for (atomid=0; atomid<numatoms; atomid++) {
   float dx = coorx - atominfo[atomid].x;
   float dy = coory - atominfo[atomid].y;
   energyval += atominfo[atomid].w * rsqrtf(dx*dx + dy*dy + atominfo[atomid].z);
 }
 energygrid[outaddr] = curenergy + energyval;
```

DCS CUDA Algorithm: Unrolling Loops

- Reuse atom data and partial distance components multiple times
- Add each atom’s contribution to several lattice points at a time, where distances only differ in one component



DCS Inner Loop (Unroll and Jam)

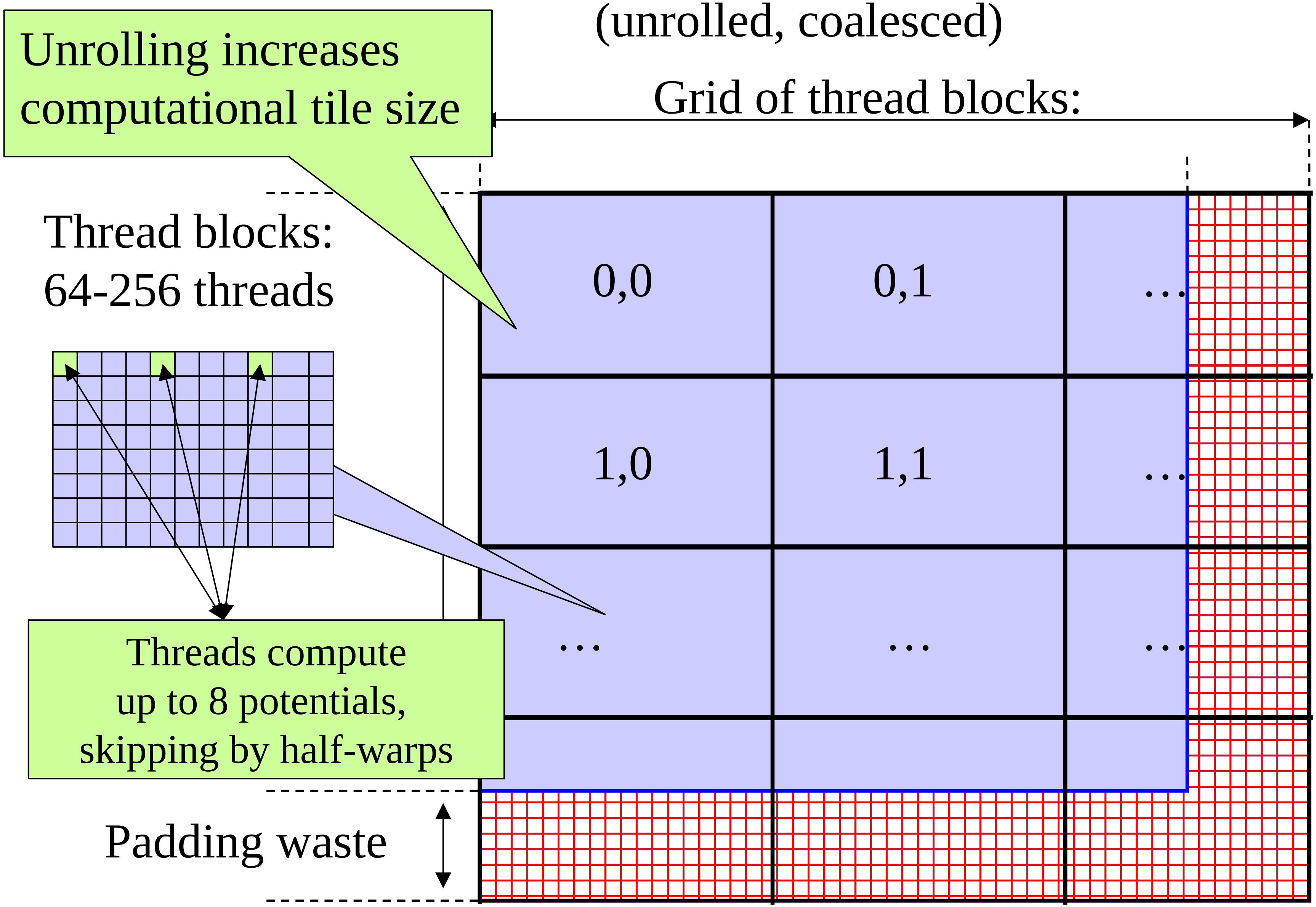
```
…
 for (atomid=0; atomid<numatoms; atomid++) {
   float dy = coory - atominfo[atomid].y;
   float dysqpdzsq = (dy * dy) + atominfo[atomid].z;
   float dx1 = coorx1 - atominfo[atomid].x;
   float dx2 = coorx2 - atominfo[atomid].x;
   float dx3 = coorx3 - atominfo[atomid].x;
   float dx4 = coorx4 - atominfo[atomid].x;
   energyvalx1 += atominfo[atomid].w * rsqrtf(dx1*dx1 + dysqpdzsq);
   energyvalx2 += atominfo[atomid].w * rsqrtf(dx2*dx2 + dysqpdzsq);
   energyvalx3 += atominfo[atomid].w * rsqrtf(dx3*dx3 + dysqpdzsq);
   energyvalx4 += atominfo[atomid].w * rsqrtf(dx4*dx4 + dysqpdzsq);
 }
…
```

DCS CUDA Block/Grid Decomposition (unrolled)

- This optimization technique (unroll and jam) consumes more registers in trade for increased arithmetic intensity
- Kernel variations that calculate more than one lattice point per thread, result in larger computational tiles:
  - Thread count per block must be decreased to reduce computational tile size as unrolling is increased
  - Otherwise, tile size gets bigger as threads do more than one lattice point evaluation, resulting on a significant increase in padding and wasted computations at edges

DCS CUDA Block/Grid Decomposition

(unrolled, coalesced)

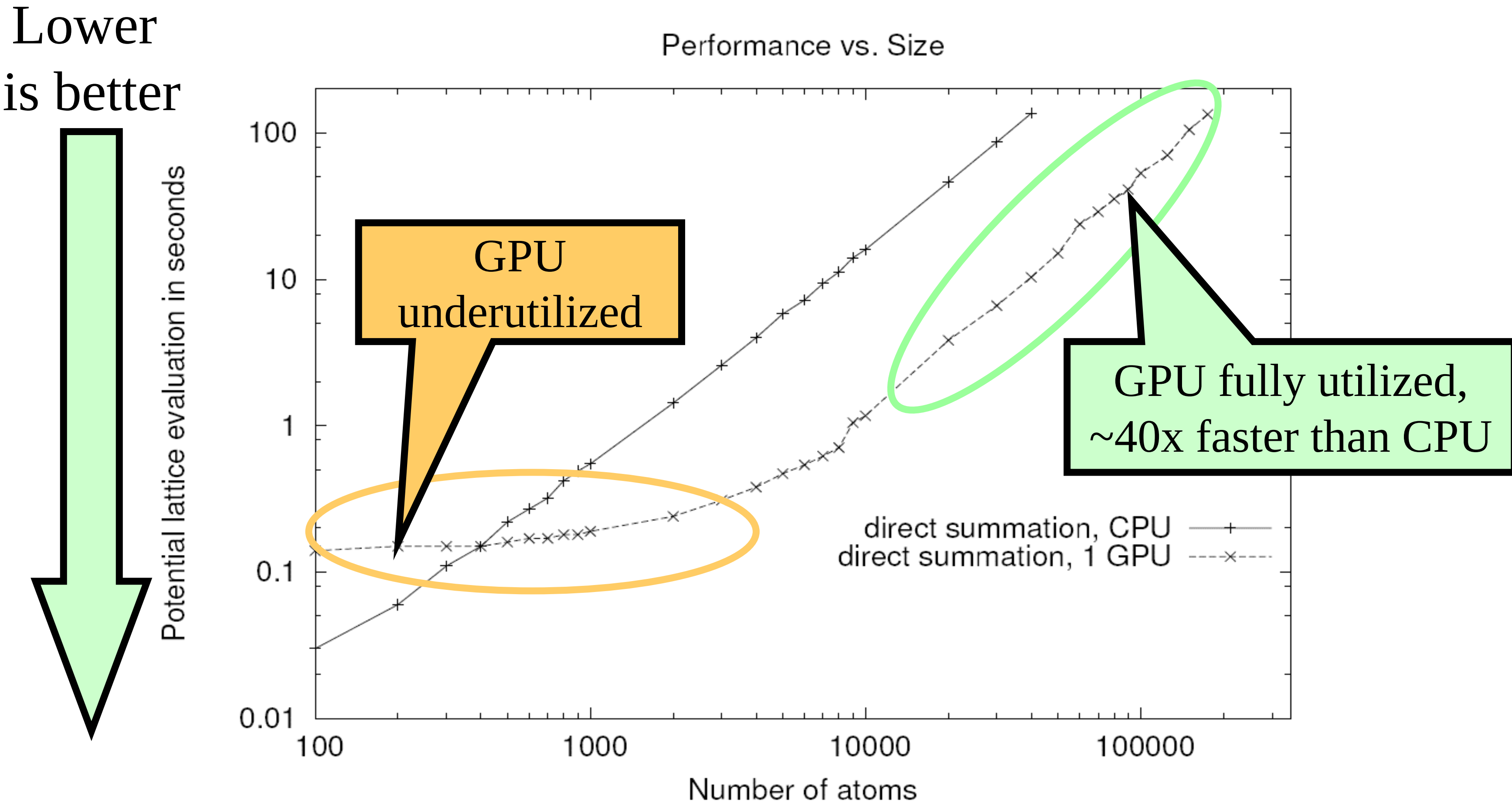


DCS Version 4: Kernel Structure

291.5 GFLOPS, 39.5 Billion Atom Evals/Sec

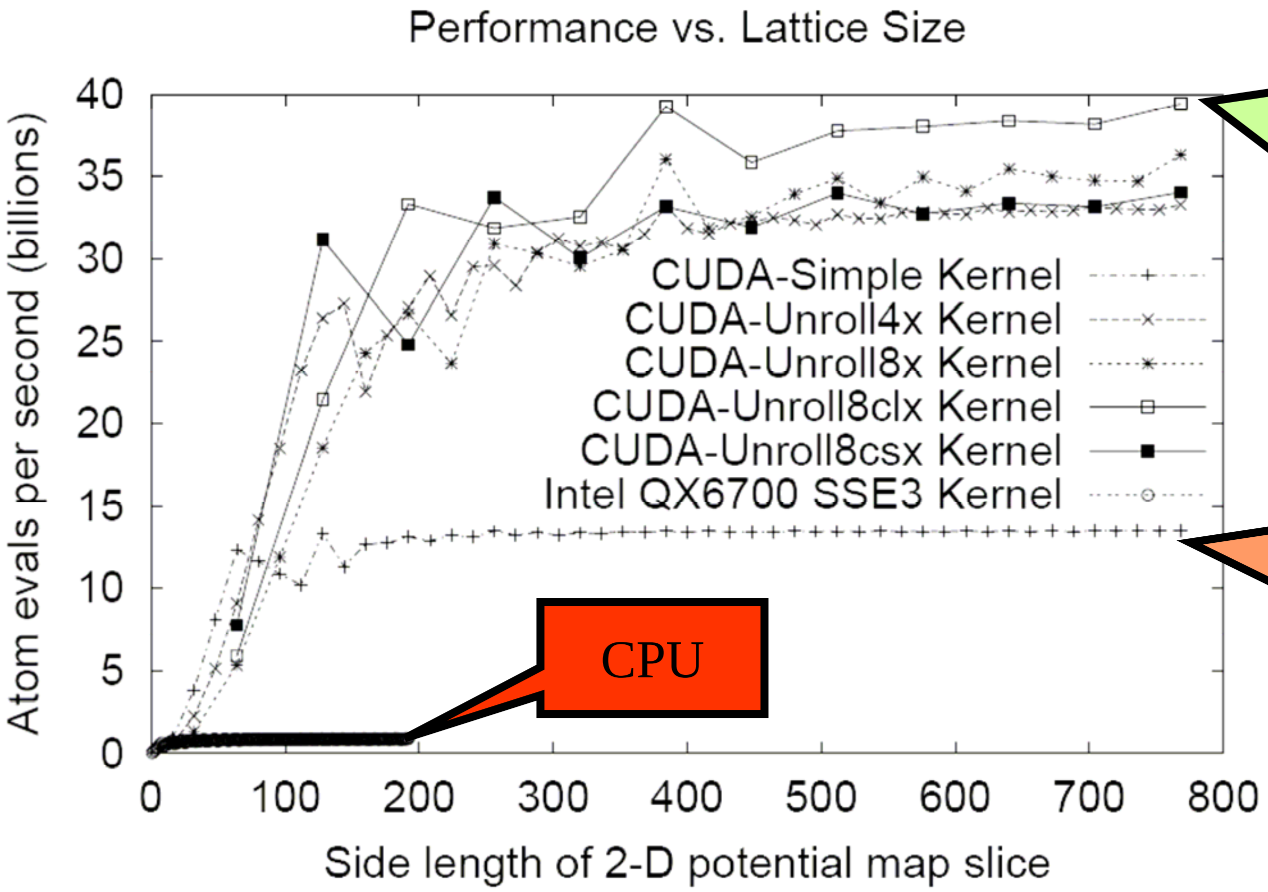
- Processes 8 lattice points at a time in the inner loop
- Subsequent lattice points computed by each thread are offset to guarantee coalesced memory accesses
- Loads and increments 8 potential map lattice points from global memory at completion of of the summation, avoiding register consumption
- Code is too long to show, but is available by request

Direct Coulomb Summation Runtime



Accelerating molecular modeling applications with graphics processors.
J. Stone, J. Phillips, P. Freddolino, D. Hardy, L. Trabuco, K. Schulten.
J. Comp. Chem., 28:2618-2640, 2007.

Direct Coulomb Summation Performance



CPU

CUDA-Unroll8clx: fastest GPU kernel, 44x faster than CPU, 291 GFLOPS on GeForce 8800GTX

CUDA-Simple: 14.8x faster, 33% of fastest GPU kernel

GPU computing. J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips. *Proceedings of the IEEE*, 96:879-899, 2008.

Multi-GPU DCS Algorithm:

- One host thread is created for each CUDA GPU, attached according to host thread ID:
  - First CUDA call binds that thread's CUDA context to that GPU for life
- Map slices are decomposed cyclically onto the available GPUs
- Map slices are usually larger than the host memory page size, so false sharing and related effects are not a problem for this application

Multi-GPU Direct Coulomb Summation

- Effective memory bandwidth scales with the number of GPUs utilized
- PCIe bus bandwidth not a bottleneck for this algorithm
- 117 billion evals/sec
- 863 GFLOPS
- 131x speedup vs. CPU core
- Power: 700 watts during benchmark

Quad-core Intel QX6700

Three NVIDIA GeForce 8800GTX

Multi-GPU Direct Coulomb Summation

- 4-GPU (2 Quadroplex) Opteron node at NCSA
- 157 billion evals/sec
- 1.16 TFLOPS
- 176x speedup vs. Intel QX6700 CPU core w/ SSE

NCSA GPU Cluster

http://www.ncsa.uiuc.edu/Projects/GPUcluster/

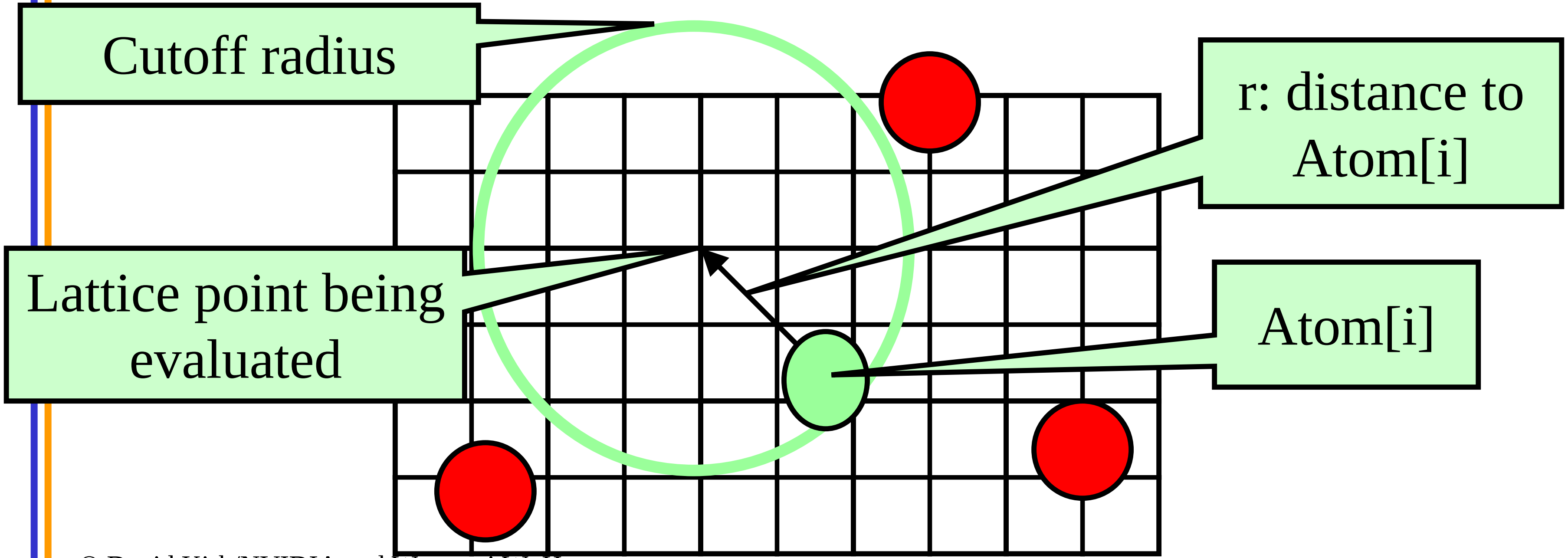
- 4-GPU (GT200)
- 241 billion evals/sec
- 1.78 TFLOPS
- 271x speedup vs. Intel QX6700 CPU core w/ SSE

Cutoff Summation

- At each lattice point, sum potential contributions for atoms within cutoff radius:

```
if (distance to atom[i] < cutoff)
  potential += (charge[i] / r) * s(r)
```

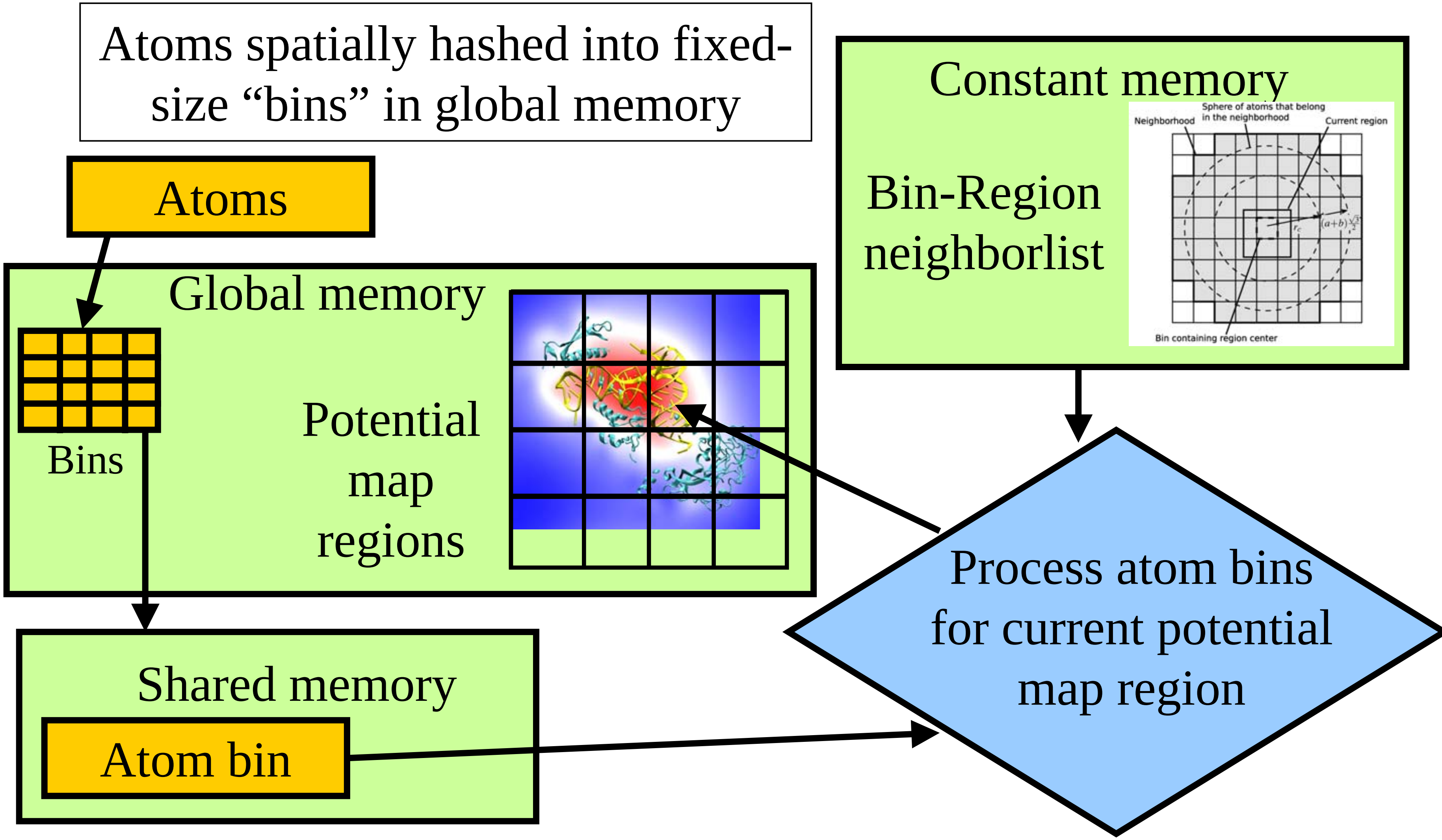
- Smoothing function s(r) is algorithm dependent



Infinite vs. Cutoff Potentials

- Infinite range potential:
  - All atoms contribute to all lattice points
  - Summation algorithm has quadratic complexity
- Cutoff (range-limited) potential:
  - Atoms contribute within cutoff distance to lattice points
  - Summation algorithm has linear time complexity
  - Has many applications in molecular modeling:
    - Replace electrostatic potential with shifted form
    - Short-range part for fast methods of approximating full electrostatics
    - Used for fast decaying interactions (e.g. Lennard-Jones, Buckingham)

Cutoff Summation on the GPU



Atoms spatially hashed into fixed-size “bins” in global memory

Atoms

Global memory

Bins

Potential map regions

Constant memory

Bin-Region neighborlist

Process atom bins for current potential map region

Shared memory

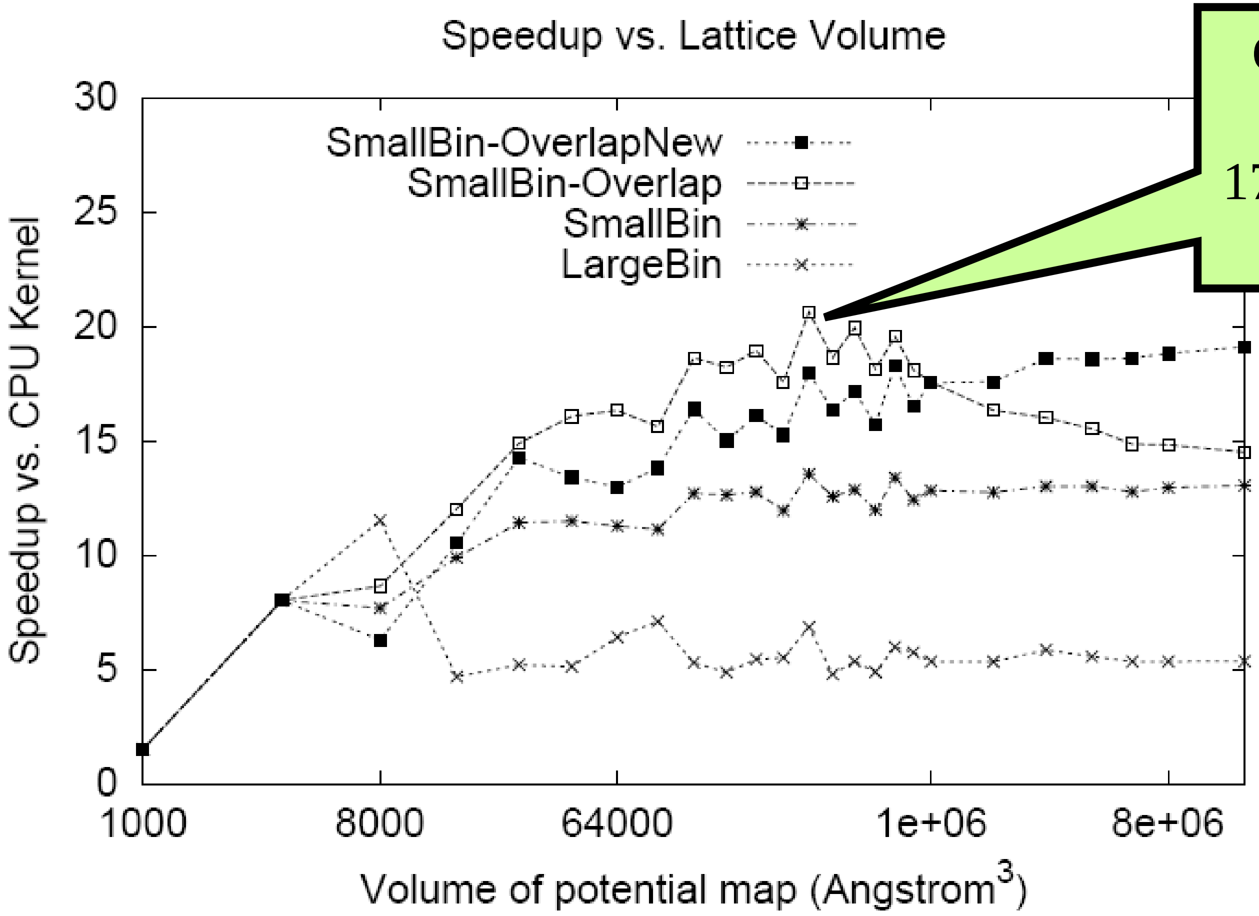
Atom bin

Sphere of atoms that belong in the neighborhood
Neighborhood
Current region
r_c
$(a+b)\frac{\sqrt{3}}{2}$
Bin containing region center

Using the CPU to Improve GPU Performance

- GPU performs best when the work evenly divides into the number of threads/processing units
- Optimization strategy:
  - Use the CPU to “regularize” the GPU workload
  - Handle exceptional or irregular work units on the CPU while the GPU processes the bulk of the work
  - On average, the GPU is kept highly occupied, attaining a much higher fraction of peak performance

Cutoff Summation Runtime



GPU cutoff with CPU overlap: 17x-21x faster than CPU core

GPU acceleration of cutoff pair potentials for molecular modeling applications. C. Rodrigues, D. Hardy, J. Stone, K. Schulten, W. Hwu. *Proceedings of the 2008 Conference On Computing Frontiers*, pp. 273-282, 2008.

NAMD Parallel Molecular Dynamics

Kale et al., J. Comp. Phys. 151:283-312, 1999.

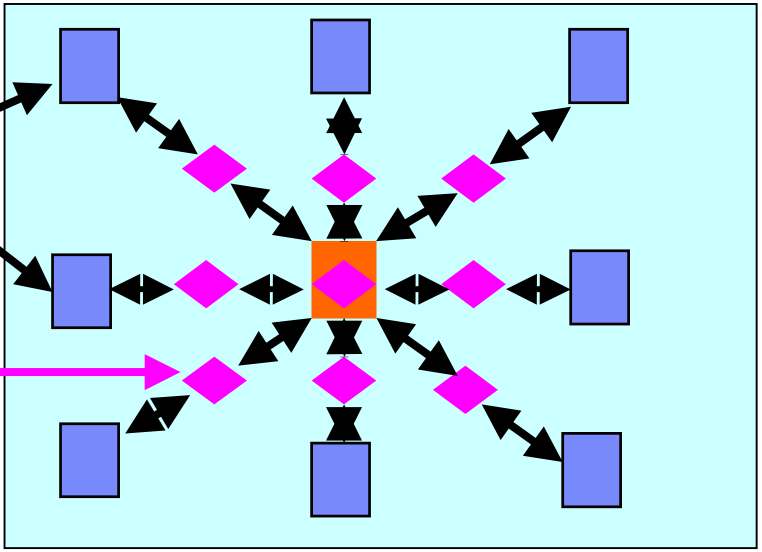
- Designed from the beginning as a parallel program
- Uses the Charm++ philosophy:
  - Decompose computation into a large number of objects
  - Intelligent Run-time system (Charm++) assigns objects to processors for dynamic load balancing with minimal communication

Hybrid of spatial and force decomposition:

•Spatial decomposition of atoms into cubes (called patches)

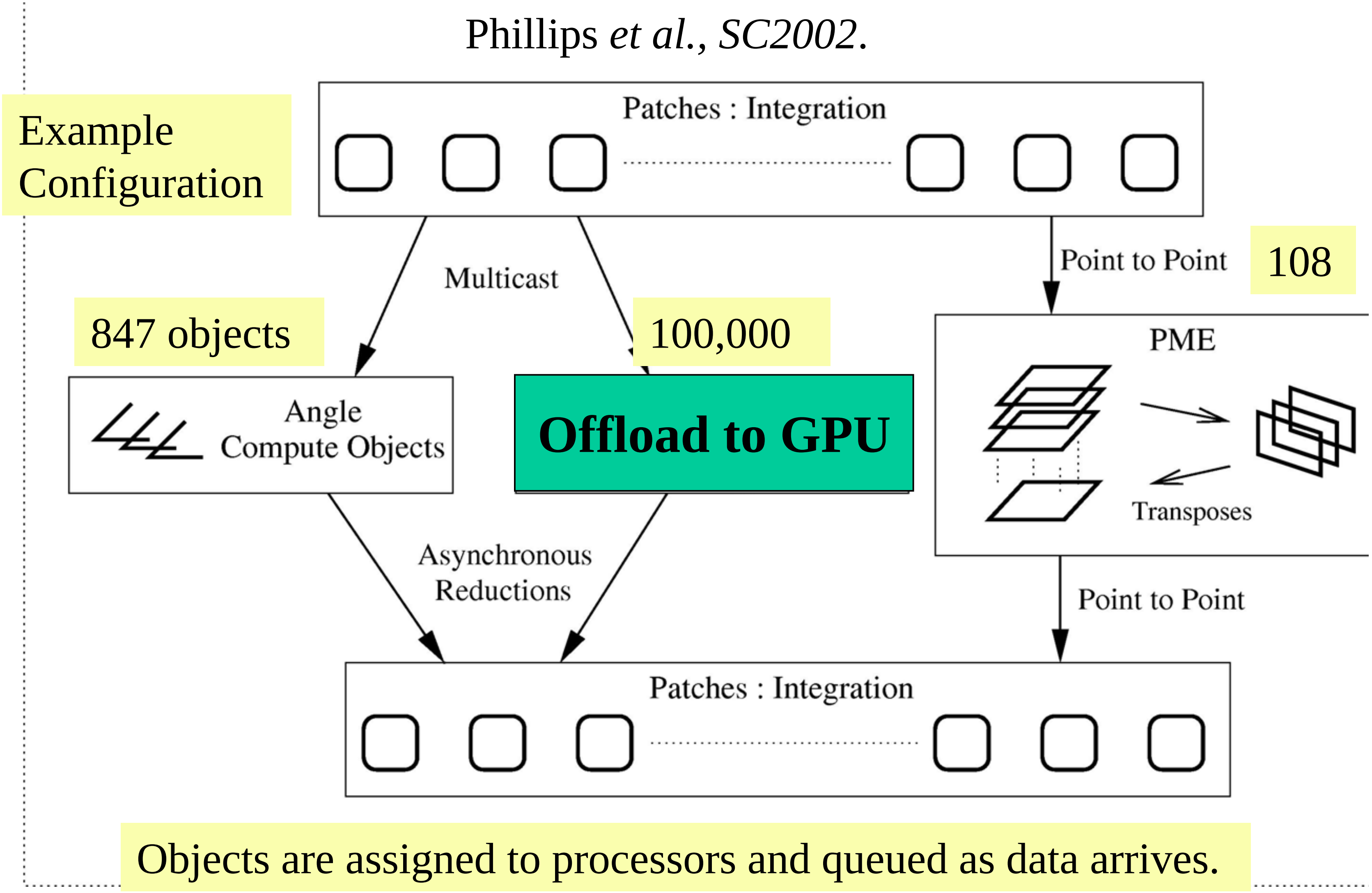
•For every pair of interacting patches, create one object for calculating electrostatic interactions

•Recent: Blue Matter, Desmond, etc. use this idea in some form



NAMD Overlapping Execution

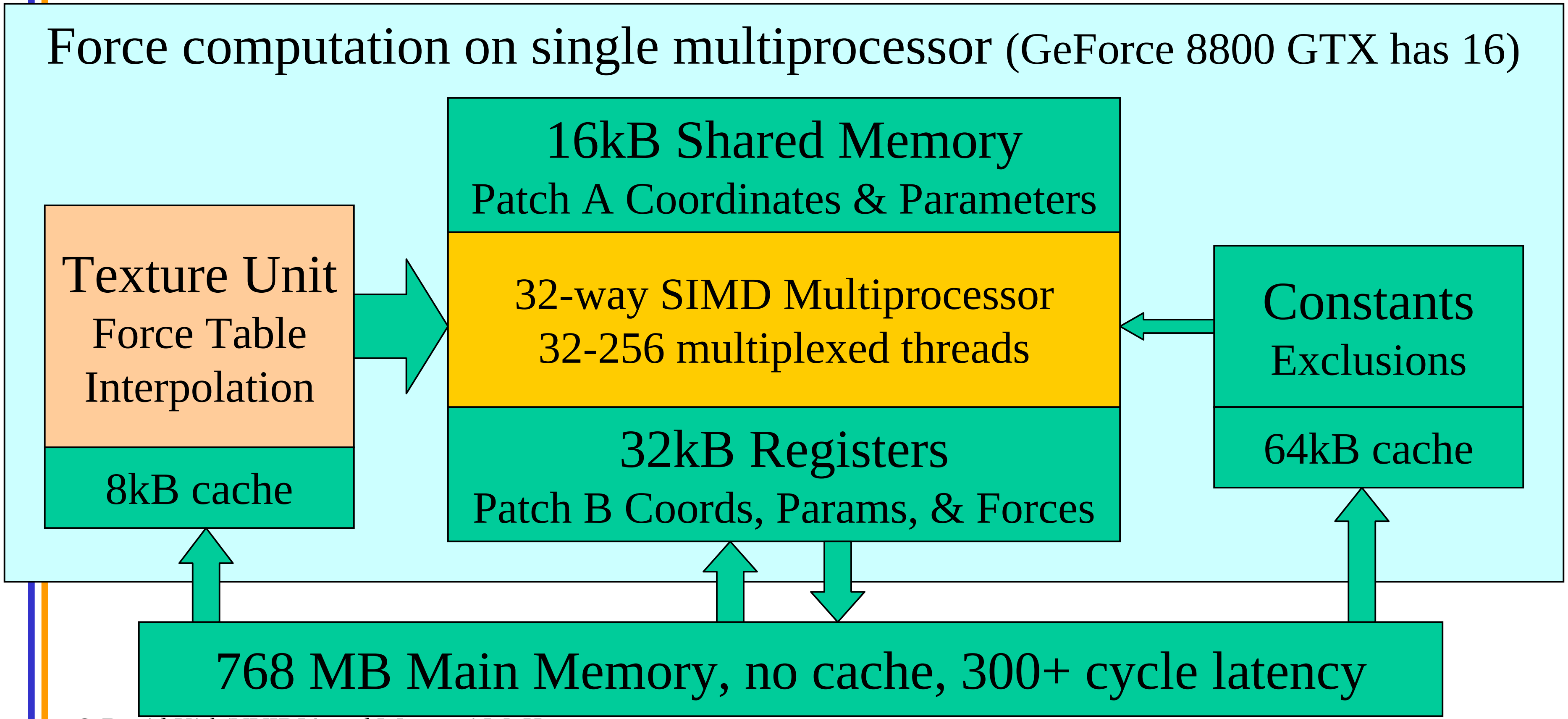
Phillips *et al., SC2002*.



Objects are assigned to processors and queued as data arrives.

Nonbonded Forces on G80 GPU

- Start with most expensive calculation: direct nonbonded interactions.
- Decompose work into pairs of patches, identical to NAMD structure.
- GPU hardware assigns patch-pairs to multiprocessors dynamically.


Force computation on single multiprocessor (GeForce 8800 GTX has 16)
16kB Shared Memory
Patch A Coordinates & Parameters
Texture Unit
Force Table
Interpolation
8kB cache
32-way SIMD Multiprocessor
32-256 multiplexed threads
Constants
Exclusions
64kB cache
32kB Registers
Patch B Coords, Params, & Forces
768 MB Main Memory, no cache, 300+ cycle latency

Stone *et al.*, *J. Comp. Chem.* **28**:2618-2640, 2007.

Nonbonded Forces CUDA Code

```
texture<float4> force_table;
__constant__ unsigned int exclusions[];
__shared__ atom jatom[];
atom iatom;      // per-thread atom, stored in registers
float4 iforce;   // per-thread force, stored in registers
for ( int j = 0; j < jatom_count; ++j ) {
  float dx = jatom[j].x - iatom.x;   float dy = jatom[j].y - iatom.y;  float dz = jatom[j].z - iatom.z;
  float r2 = dx*dx + dy*dy + dz*dz;
  if ( r2 < cutoff2 ) {
```

Force Interpolation

```
    float4 ft = texfetch(force_table, 1.f/sqrt(r2));
```

Exclusions

```
    bool excluded = false;
    int indexdiff = iatom.index - jatom[j].index;
    if ( abs(indexdiff) <= (int) jatom[j].excl_maxdiff ) {
      indexdiff += jatom[j].excl_index;
      excluded = ((exclusions[indexdiff>>5] & (1<<(indexdiff&31))) != 0);
    }
```

Parameters

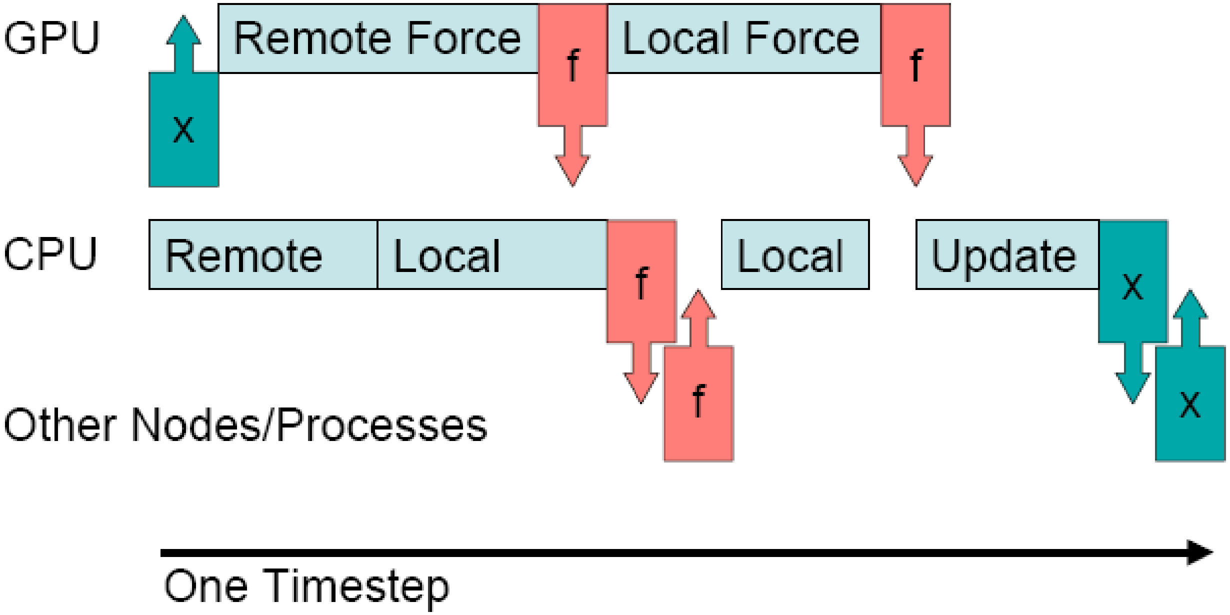
```
    float f = iatom.half_sigma + jatom[j].half_sigma;  // sigma
    f *= f*f;  // sigma^3
    f *= f;  // sigma^6
    f *= ( f * ft.x + ft.y );  // sigma^12 * fi.x - sigma^6 * fi.y
    f *= iatom.sqrt_epsilon * jatom[j].sqrt_epsilon;
    float qq = iatom.charge * jatom[j].charge;
    if ( excluded ) { f = qq * ft.w; }  // PME correction
    else { f += qq * ft.z; }  // Coulomb
```

Accumulation

```
    iforce.x += dx * f;   iforce.y += dy * f;    iforce.z += dz * f;
    iforce.w += 1.f;  // interaction count or energy
  }
}
```

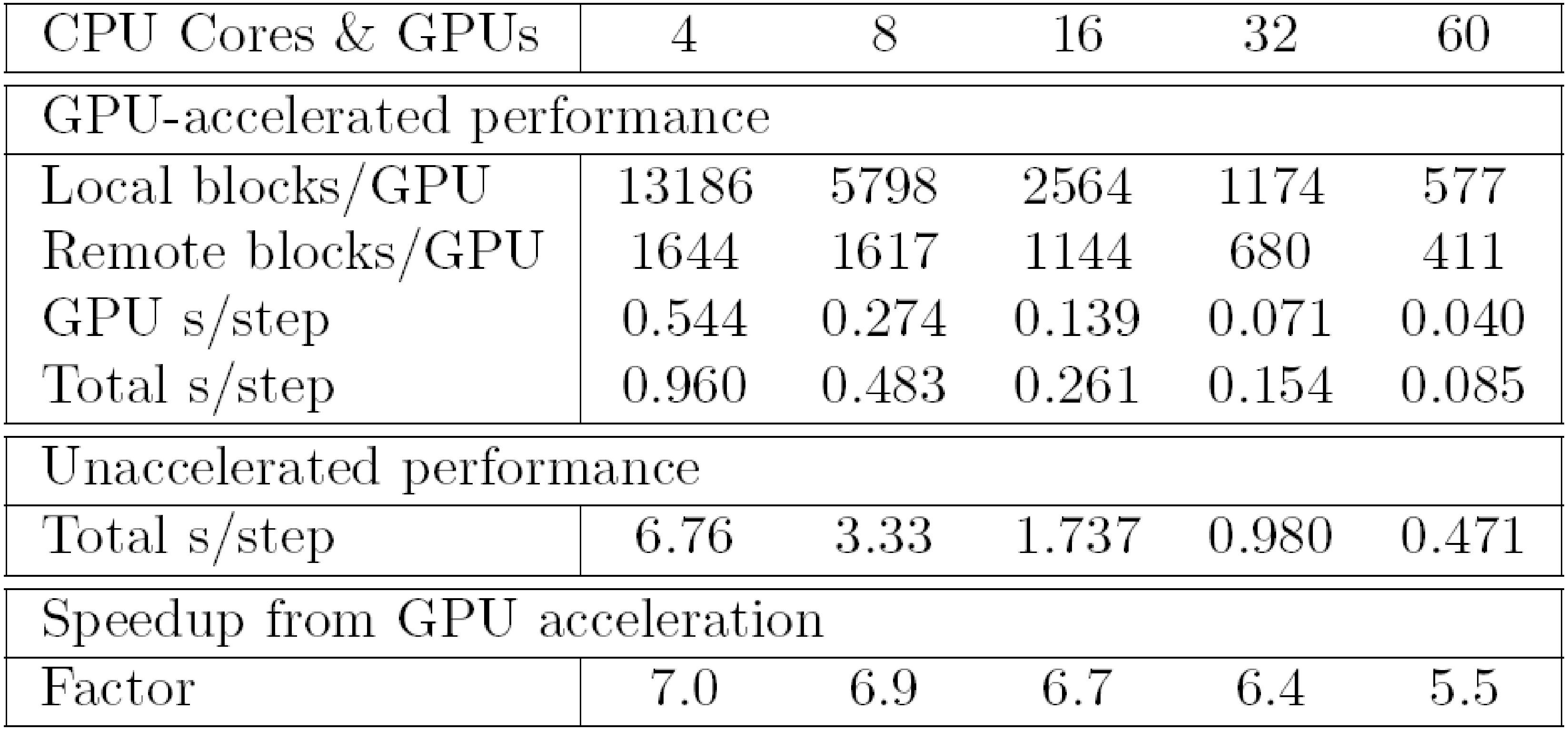
Stone *et al.*, *J. Comp. Chem.* **28**:2618-2640, 2007.

NAMD Overlapping Execution with Asynchronous CUDA kernels



GPU kernels are launched asynchronously, CPU continues with its own work, polling for GPU completion periodically. Forces needed by remote nodes are explicitly scheduled to be computed ASAP to improve overall performance.

NAMD Performance on NCSA GPU Cluster, April 2008

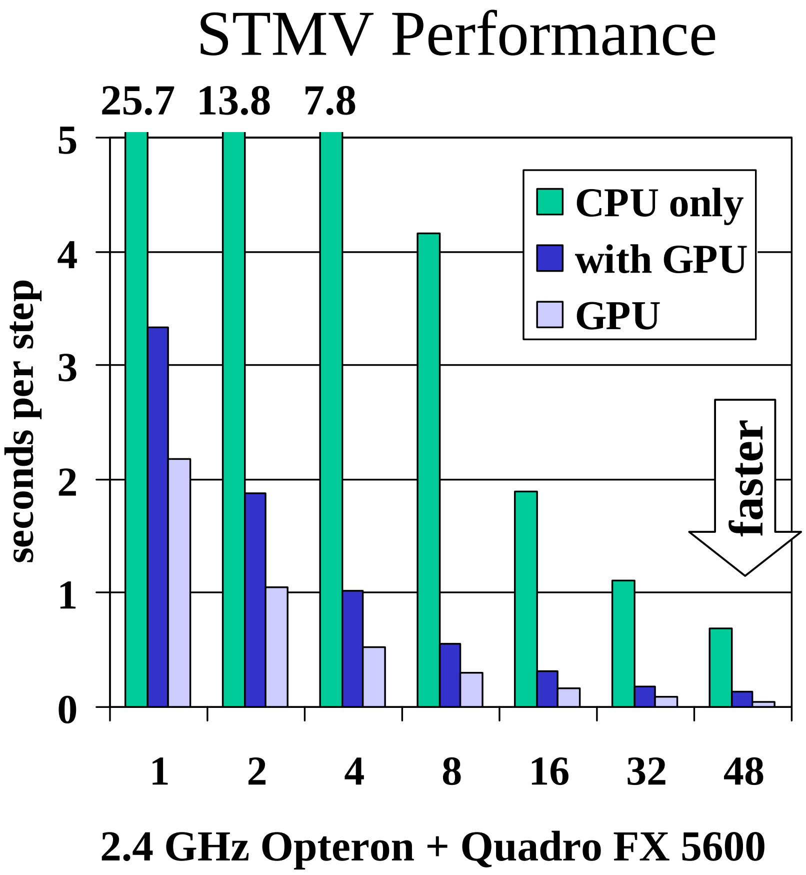


CPU Cores & GPUs	4	8	16	32	60
GPU-accelerated performance					
Local blocks/GPU	13186	5798	2564	1174	577
Remote blocks/GPU	1644	1617	1144	680	411
GPU s/step	0.544	0.274	0.139	0.071	0.040
Total s/step	0.960	0.483	0.261	0.154	0.085
Unaccelerated performance					
Total s/step	6.76	3.33	1.737	0.980	0.471
Speedup from GPU acceleration					
Factor	7.0	6.9	6.7	6.4	5.5

STMV benchmark, 1M atoms,12A cutoff,
PME every 4 steps, running on
2.4 GHz AMD Opteron + NVIDIA Quadro FX 5600

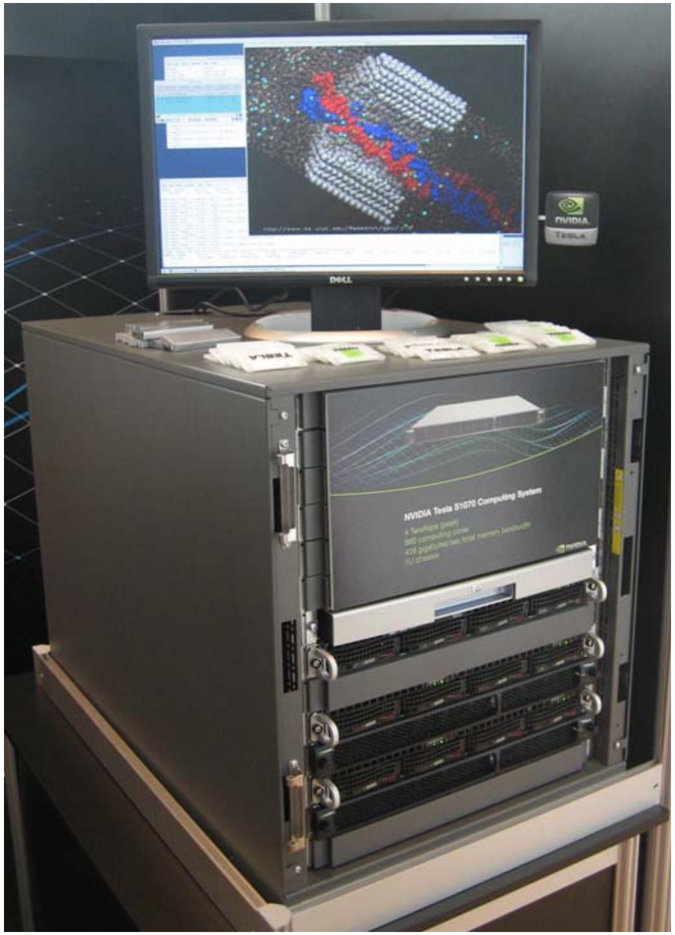
NAMD Performance on NCSA GPU Cluster, April 2008

- STMV virus (1M atoms)
- 60 GPUs match performance of 330 CPU cores
- 5.5-7x overall application speedup w/ G80-based GPUs
- Overlap with CPU
- Off-node results done first
- Plans for better performance
  - Tune or port remaining work
  - Balance GPU load



NAMD Performance on GT200 GPU Cluster, August 2008

- 8 GT200s, 240 SPs @ 1.3GHz:
  - 72x faster than a single CPU core
  - 9x overall application speedup vs. 8 CPU cores
  - 32% faster overall than 8 nodes of G80 cluster
  - GT200 CUDA kernel is 54% faster
  - ~8% variation in GPU load
- Cost of double-precision for force accumulation is minimal: only 8% slower than single-precision



GPU Kernel Performance, May 2008

GeForce 8800GTX w/ CUDA 1.1, Driver 169.09

http://www.ks.uiuc.edu/Research/gpu/

Calculation / Algorithm	Algorithm class	Speedup vs. Intel QX6700 CPU core
Fluorescence microphotolysis	Iterative matrix / stencil	12x
Pairlist calculation	Particle pair distance test	10-11x
Pairlist update	Particle pair distance test	5-15x
Molecular dynamics non-bonded force calc.	N-body cutoff force calculations	10x 20x (w/ pairlist)
Cutoff electron density sum	Particle-grid w/ cutoff	15-23x
MSM short-range	Particle-grid w/ cutoff	24x
MSM long-range	Grid-grid w/ cutoff	22x
Direct Coulomb summation	Particle-grid	44x

Lessons Learned

- GPU algorithms need fine-grained parallelism and sufficient work to fully utilize the hardware
- Fine-grained GPU work decompositions compose well with the comparatively coarse-grained decompositions used for multicore or distributed memory programing
- Much of GPU algorithm optimization revolves around efficient use of multiple memory systems and latency hiding

Lessons Learned (2)

- The host CPU can potentially be used to “regularize” the computation for the GPU, yielding better overall performance
- Overlapping CPU work with GPU can hide some communication and unaccelerated computation

Publications

http://www.ks.uiuc.edu/Research/gpu/

- Adapting a message-driven parallel application to GPU-accelerated clusters. J. Phillips, J. Stone, K. Schulten. *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, *(in press)*
- GPU acceleration of cutoff pair potentials for molecular modeling applications. C. Rodrigues, D. Hardy, J. Stone, K. Schulten, W. Hwu. *Proceedings of the 2008 Conference On Computing Frontiers*, pp. 273-282, 2008.
- GPU computing. J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips. *Proceedings of the IEEE*, 96:879-899, 2008.
- Accelerating molecular modeling applications with graphics processors. J. Stone, J. Phillips, P. Freddolino, D. Hardy, L. Trabuco, K. Schulten. *J. Comp. Chem.*, 28:2618-2640, 2007.
- Continuous fluorescence microphotolysis and correlation spectroscopy. A. Arkhipov, J. Hüve, M. Kahms, R. Peters, K. Schulten. *Biophysical Journal*, 93:4006-4017, 2007.