

# Programming in CUDA: the Essentials, Part 2

John E. Stone

Theoretical and Computational Biophysics Group  
Beckman Institute for Advanced Science and Technology  
University of Illinois at Urbana-Champaign

**<http://www.ks.uiuc.edu/Research/gpu/>**

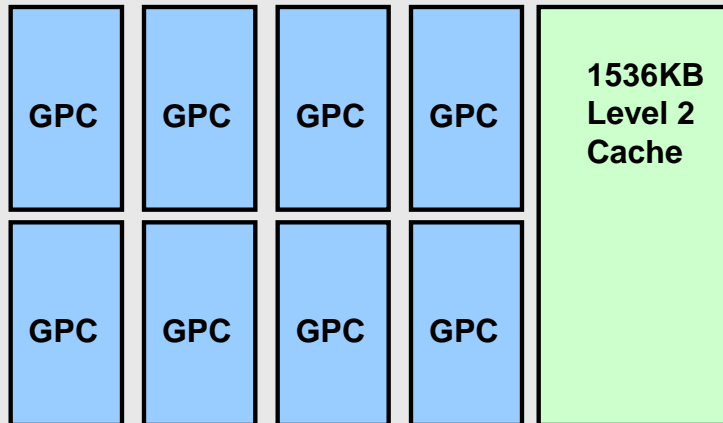
Cape Town GPU Workshop

Cape Town, South Africa, April 30, 2013



# NVIDIA Kepler GPU

~3-6 GB DRAM Memory w/ ECC



## Graphics Processor Cluster

SMX

SMX

## Streaming Multiprocessor - SMX

64 KB Constant Cache

64 KB L1 Cache / Shared Memory

48 KB Tex + Read-only Data Cache



Tex Unit

16 × Execution block =  
192 SP, 64 DP,  
32 SFU, 32 LDST

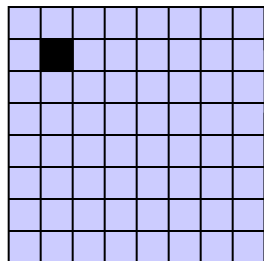
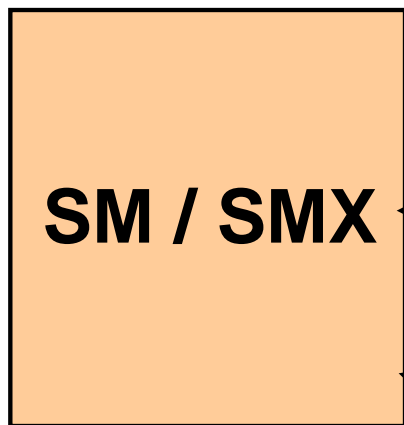
# CUDA Work Abstraction

- Work is expressed as a multidimensional array of independent work items called “**threads**” – not the same thing as a CPU thread
- CUDA Kernels can be thought of as telling a GPU to compute **all iterations** of a set of nested loops **concurrently**
- Threads are dynamically scheduled onto hardware according to a hierarchy of thread groupings

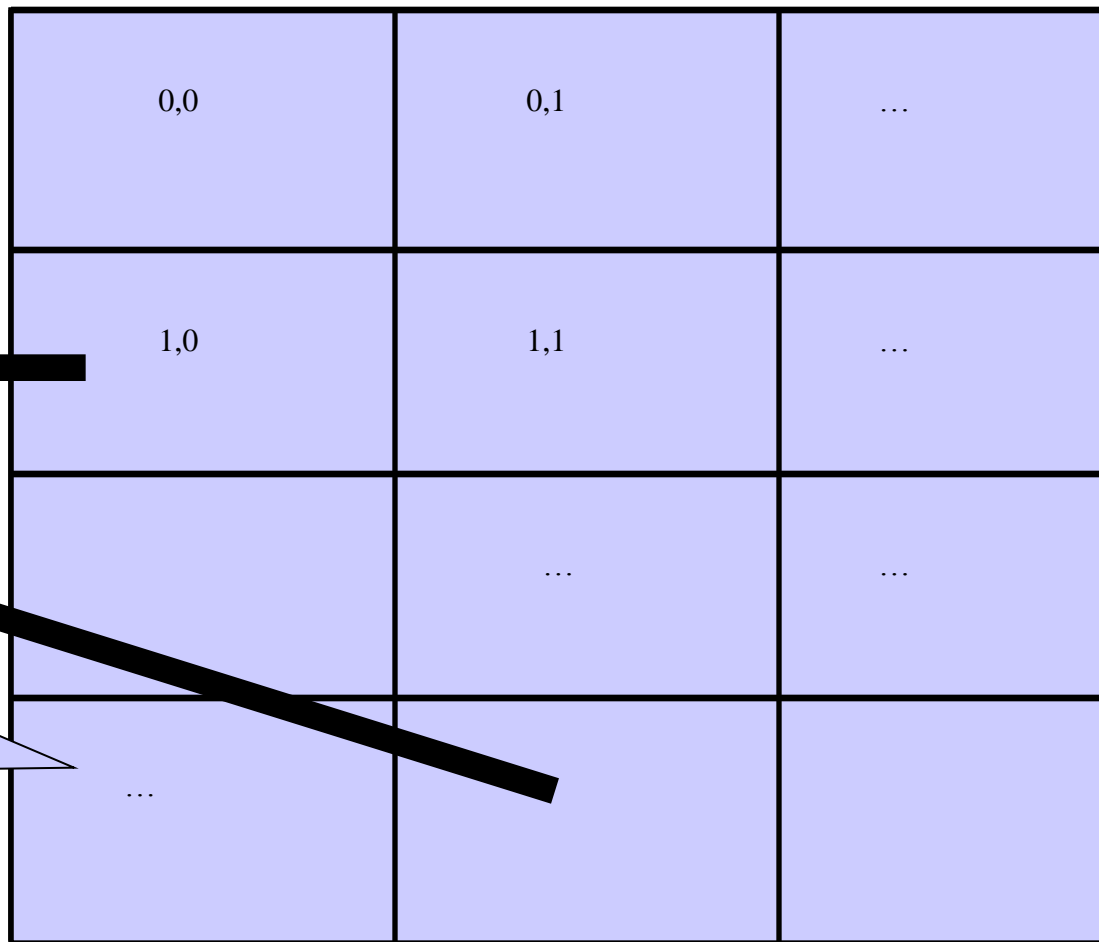
# CUDA Work Abstractions: Grids, Thread Blocks, Threads

1-D, 2-D, or 3-D (SM  $\geq$  2.x)  
Grid of thread blocks:

Thread blocks are  
scheduled onto pool  
of GPU SMs...



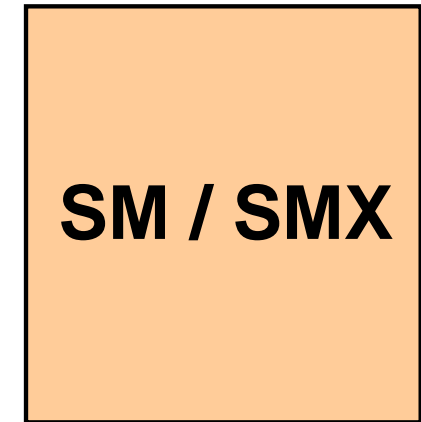
1-D, 2-D, 3-D  
thread block:



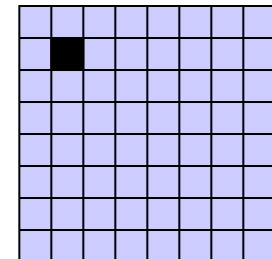
# GPU Thread Block Execution

- Thread blocks are decomposed onto hardware in **32-thread “warps”**
- Hardware execution is scheduled in units of warps – an SM can execute warps from several thread blocks
- Warps run in SIMD-style execution:
  - All threads execute the same instruction in lock-step
  - If one thread stalls, the entire warp stalls...
  - A branch taken by a thread has to be taken by all threads... (**divergence is bad!**)

**Thread blocks are multiplexed onto pool of GPU SMs...**



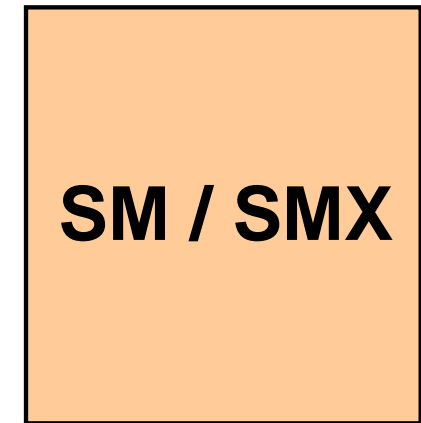
1-D, 2-D, 3-D  
thread block:



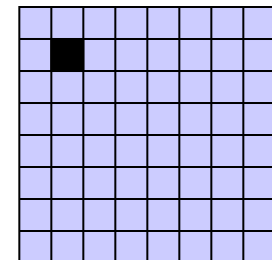
# GPU Warp Branch Divergence

- Branch divergence: when not all threads take the same branch, the entire warp has to execute both sides of the branch
- GPU blocks memory writes from disabled threads in the “if then” branch, then inverts all thread enable states and runs the “else” branch
- GPU hardware detects warp reconvergence and then runs normally...
- Not unique to GPUs, an attribute of all SIMD hardware designs...
- In the case of the GPU, we are at least benefiting from a completely hardware-based implementation...

**Thread blocks are multiplexed onto pool of GPU SMs...**



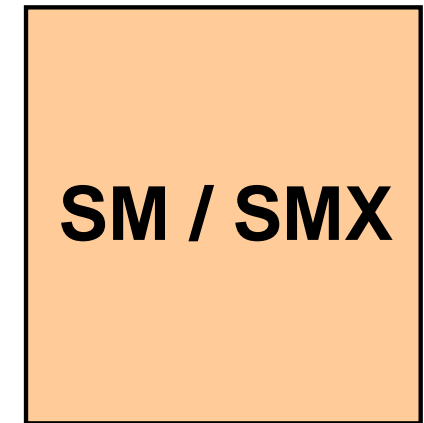
1-D, 2-D, 3-D  
thread block:



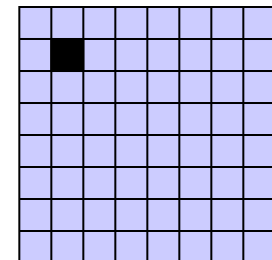
# GPU Thread Block Collective Operations

- Threads within the same thread block can communicate with each other in fast on-chip **shared memory**
- Once scheduled on an SM, **thread blocks run until completion**
- Because the order of thread block execution is arbitrary and they can't be stopped, **they cannot communicate or synchronize with other thread blocks**

**Thread blocks are multiplexed onto pool of GPU SMs...**

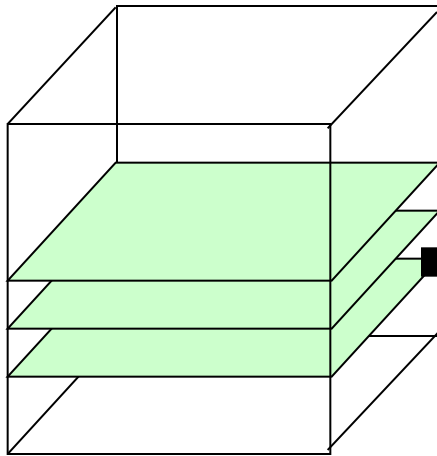


1-D, 2-D, 3-D  
thread block:

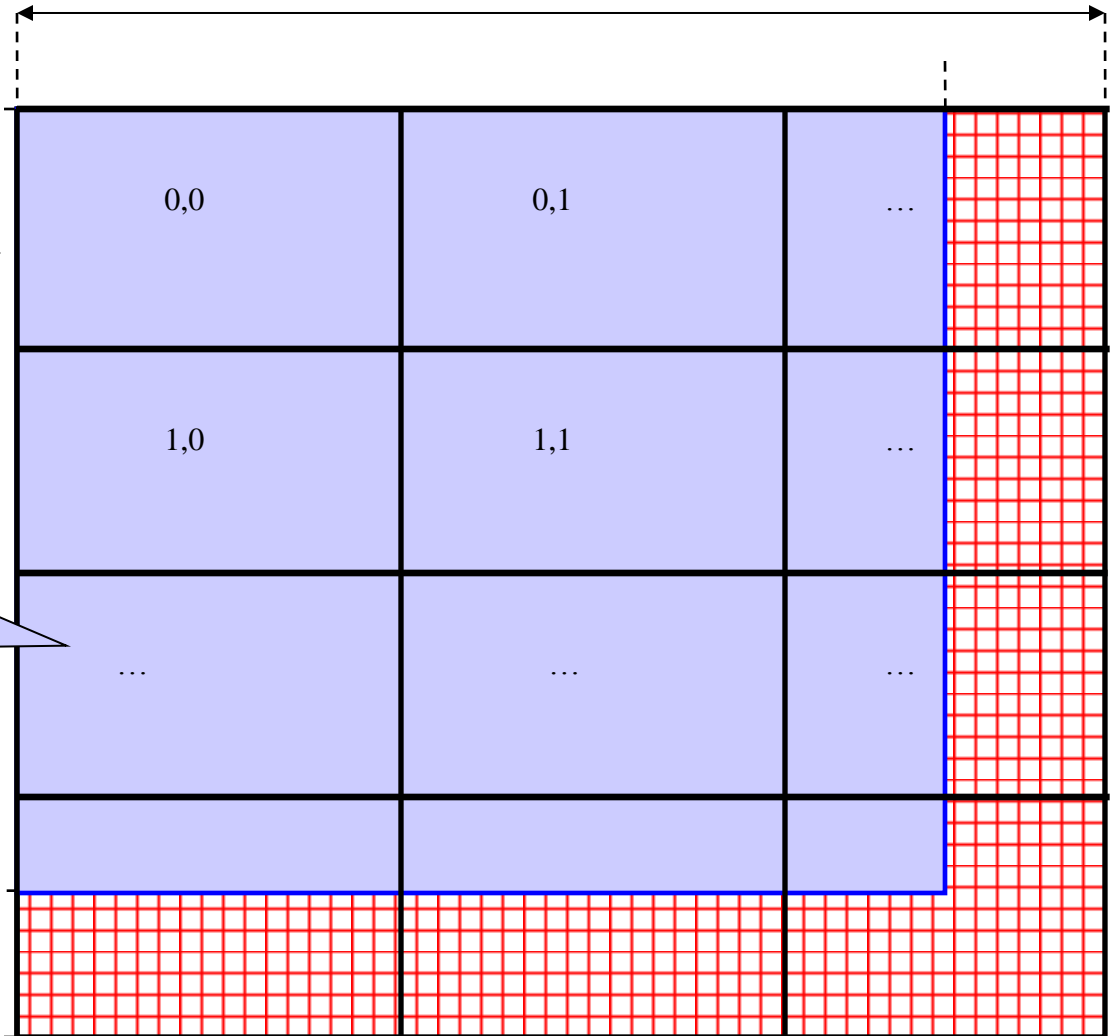


# CUDA Grid/Block/Thread Decomposition

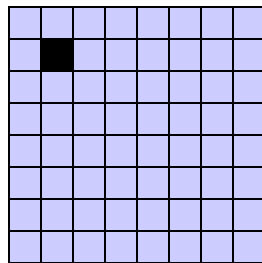
**1-D, 2-D, or 3-D  
Computational Domain**



**1-D, 2-D, or 3-D (SM  $\geq$  2.x)  
Grid of thread blocks:**



**1-D, 2-D, 3-D  
thread block:**



Padding arrays out to full blocks  
optimizes global memory performance  
by guaranteeing memory coalescing



# Indexing Work

- Within a CUDA kernel:
  - Grid: `gridDim.[xyz]`
  - Block: `blockDim.[xyz]` and `blockIdx.[xyz]`
  - Thread: `threadIdx.[xyz]`
- Example CUDA kernel with 1-D Indexing:  

```
__global__ void cuda_add(float *c, float *a, float *b) {  
    int idx = (blockIdx.x * blockDim.x) + threadIdx.x;  
    c[idx] = a[idx] + b[idx];  
}
```

# Running a GPU kernel:

```
int sz = N * sizeof(float);
```

```
...
```

```
cudaMalloc((void**) &a_gpu, sz);
```

```
cudaMemcpy(a_gpu, a, sz, cudaMemcpyHostToDevice);
```

```
... // do the same for 'b_gpu', allocate 'c_gpu'
```

```
int Bsz = 256; // 1-D thread block size
```

```
cuda_add<<<N/Bsz, Bsz>>>(c, a, b);
```

```
cudaDeviceSynchronize(); // make CPU wait for completion
```

```
...
```

```
cudaMemcpy(c, c_gpu, sz, cudaMemcpyDeviceToHost);
```

```
cudaFree(a_gpu);
```

```
... // free 'b_gpu', and 'c_gpu'...
```

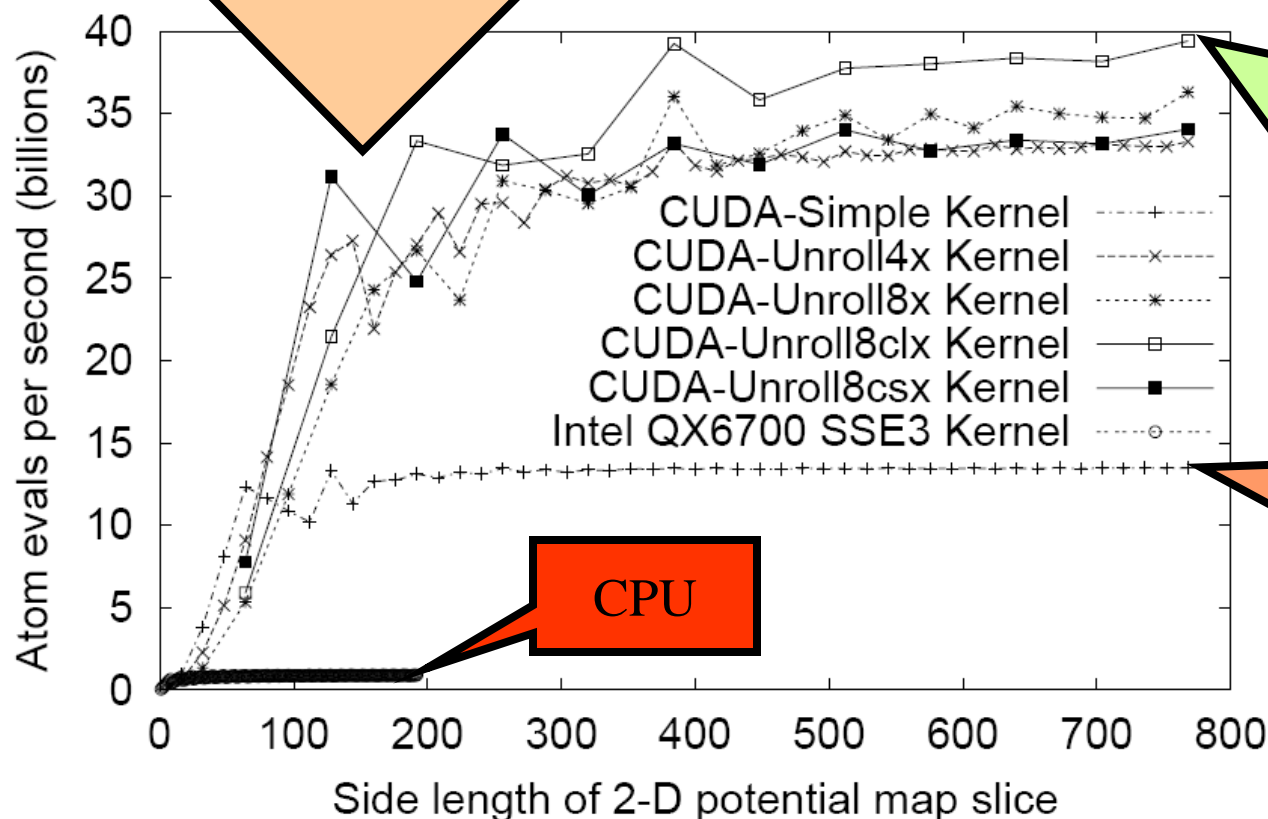
# What if Work Size Isn't an Integer Multiple of the Thread Block Size?

- Threads must check if they are “in bounds”:

```
__global__ void cuda_add(float *c, float *a, float *b, int N) {  
    int idx = (blockIdx.x * blockDim.x) + threadIdx.x;  
    if (idx < N) {  
        c[idx] = a[idx] + b[idx];  
    }  
}
```

# Direct Coulomb Summation Performance

Number of thread blocks modulo number of SMs results in significant performance variation for small workloads



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.

# An Approach to Writing CUDA Kernels

- Find an algorithm that can expose **substantial parallelism**, we'll ultimately need thousands of independent threads...
- Identify **appropriate** GPU memory or texture subsystems used to store data used by kernel
- Are there trade-offs that can be made to exchange computation for **more parallelism**?
  - Though counterintuitive, past successes resulted from this strategy
  - “Brute force” methods that expose significant parallelism do surprisingly well on GPUs
- Analyze the real-world use case for the problem and select a specialized kernel for the problem sizes that will be heavily used

# Getting Performance From GPUs

- Don't worry (much) about counting arithmetic operations...at least until you have nothing else left to do
- GPUs provide tremendous memory bandwidth, but even so, **memory bandwidth often ends up being the performance limiter**
- Keep/reuse data in **registers** as long as possible
- The main consideration when programming GPUs is **accessing memory efficiently**, and storing operands in the **most appropriate memory system** according to data size and access pattern

# Avoid Output Conflicts, Conversion of Scatter to Gather

- Many CPU codes contain algorithms that “scatter” outputs to memory, to reduce arithmetic
- Scattered output can create bottlenecks for GPU performance due to bank conflicts
- On the GPU, it’s often better to do **more arithmetic**, in exchange for a **regularized output pattern**, or to convert “scatter” algorithms to “gather” approaches

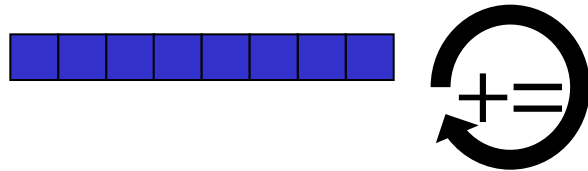
# Avoid Output Conflicts: Privatization Schemes

- ***Privatization***: use of private work areas for workers
  - Avoid/reduce the need for thread synchronization barriers
  - Avoid/reduce the need atomic increment/decrement operations during work, use **parallel reduction** at the end...
- By working in separate memory buffers, workers **avoid read/modify/write conflicts** of various kinds
- Huge GPU thread counts make it impractical to privatize data on a per-thread basis, so GPUs must use **coarser granularity: warps, thread-blocks**
- Use of the **on-chip shared memory** local to each SM can often be considered a form of privatization



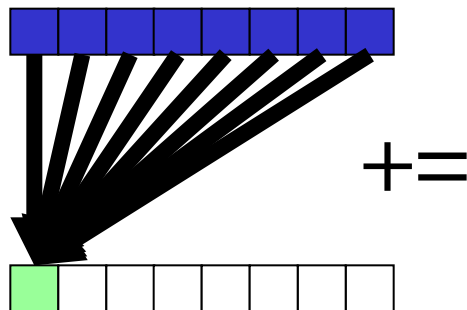
# Example: avoiding output conflicts when summing numbers among threads in a block

**Accumulate sums in thread-local registers before doing any reduction among threads**

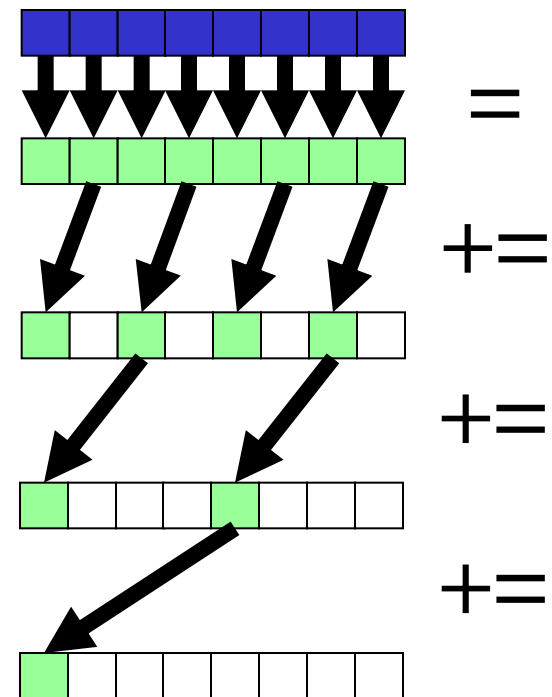


**N-way output conflict:**

Correct results require **costly barrier synchronizations** or **atomic memory operations ON EVERY ADD** to prevent threads from overwriting each other...



**Parallel reduction:** no output conflicts,  $\text{Log}_2(N)$  barriers

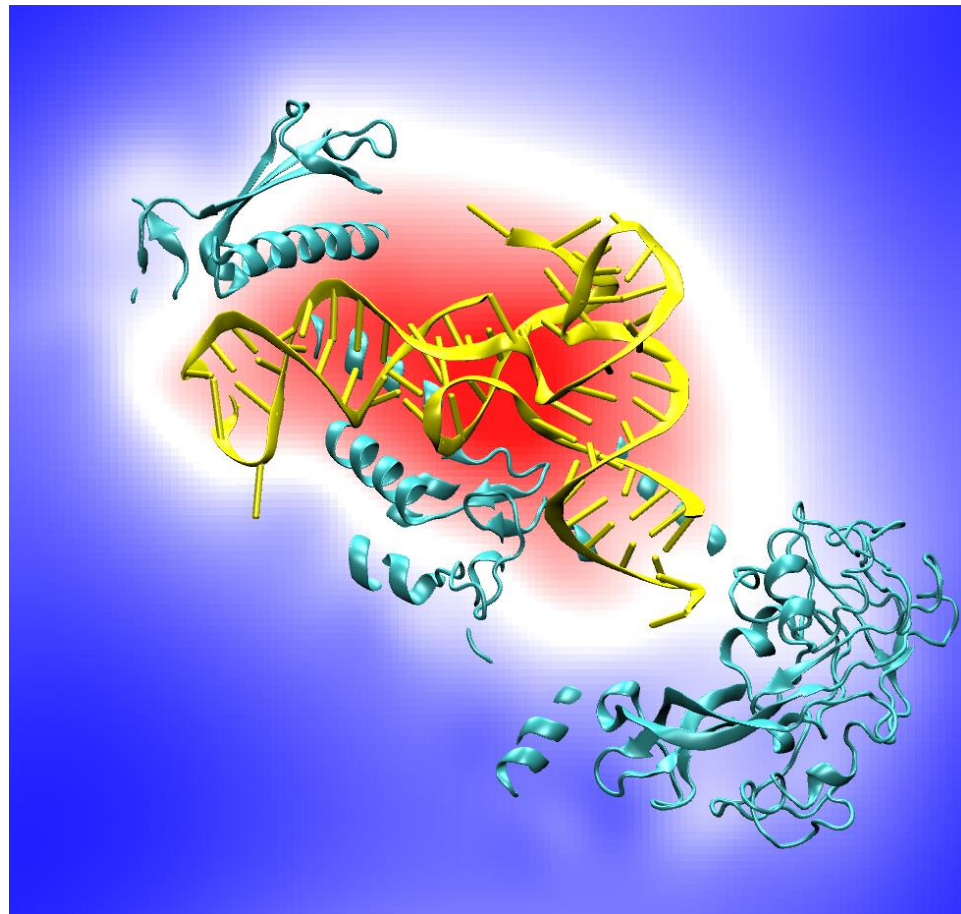


# Electrostatic Potential Maps

- Electrostatic potentials evaluated on 3-D lattice:

$$V_i = \sum_j \frac{q_j}{4\pi\epsilon_0 |\mathbf{r}_j - \mathbf{r}_i|}$$

- Applications include:
  - Ion placement for structure building
  - Time-averaged potentials for simulation
  - Visualization and analysis



Isoleucine tRNA synthetase

# Overview of Direct Coulomb Summation (DCS) Algorithm

- One of several ways to compute the electrostatic potentials on a grid, ideally suited for the GPU
- Methods such as multilevel summation can achieve much higher performance at the cost of additional complexity
- Begin with DCS for computing electrostatic maps:
  - conceptually simple algorithm well suited to the GPU
  - easy to fully explore
  - requires very little background knowledge, unlike other methods
- DCS: for each lattice point, sum potential contributions for all atoms in the simulated structure:

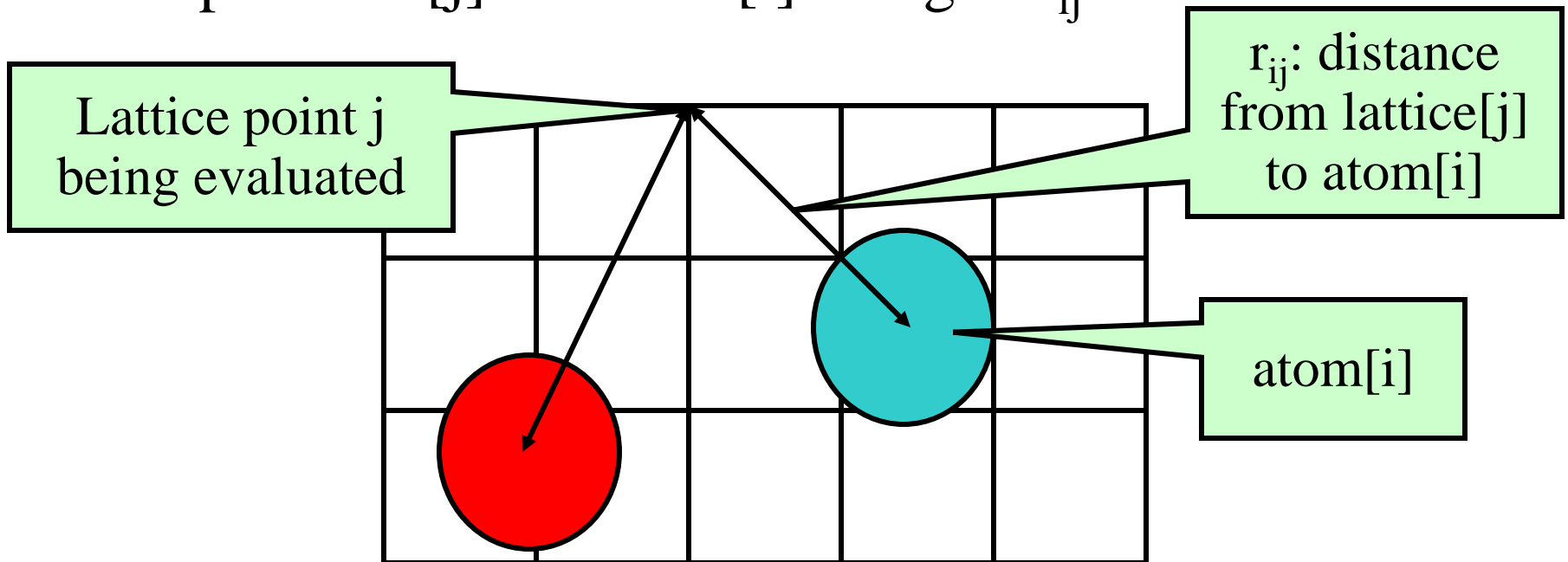
$$\text{potential}[j] += \text{atom}[i].\text{charge} / r_{ij}$$

# Direct Coulomb Summation (DCS)

## Algorithm Detail

- Each lattice point accumulates electrostatic potential contribution from all atoms:

$$\text{potential}[j] += \text{atom}[i].\text{charge} / r_{ij}$$



# DCS Computational Considerations

- Attributes of DCS algorithm for computing electrostatic maps:
  - Highly data parallel
  - Starting point for more sophisticated algorithms
  - Single-precision FP arithmetic is adequate for intended uses
  - Numerical accuracy can be further improved by compensated summation, spatially ordered summation groupings, or with the use of double-precision accumulation
  - Interesting test case since potential maps are useful for various visualization and analysis tasks
- Forms a template for related spatially evaluated function summation algorithms in CUDA

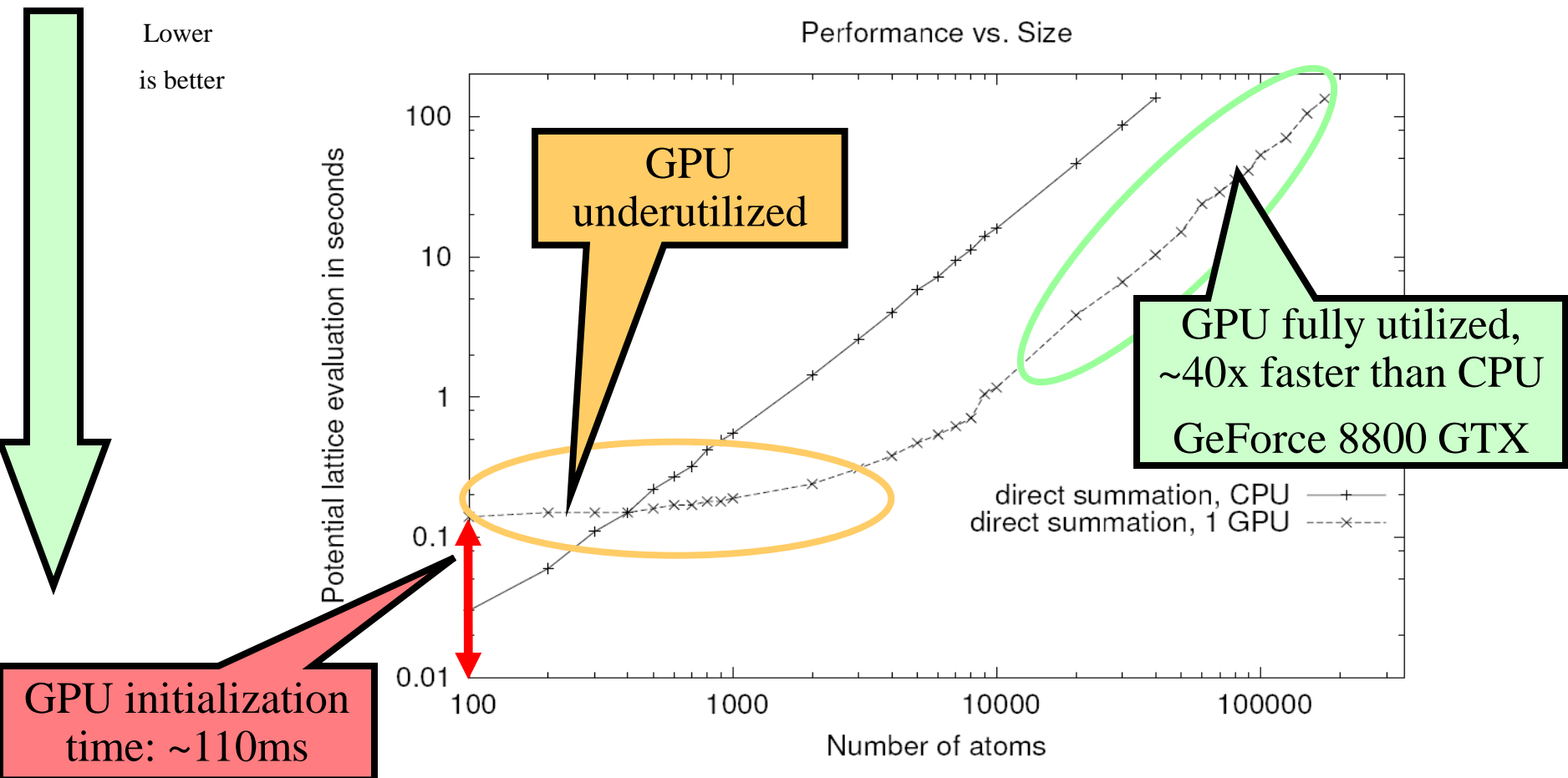
# Single Slice DCS: Simple (Slow) C Version

```
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;
        }
    }
}
```

# DCS Algorithm Design Observations

- Electrostatic maps used for ion placement require evaluation of ~20 potential lattice points per atom for a typical biological structure
- 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

# 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.



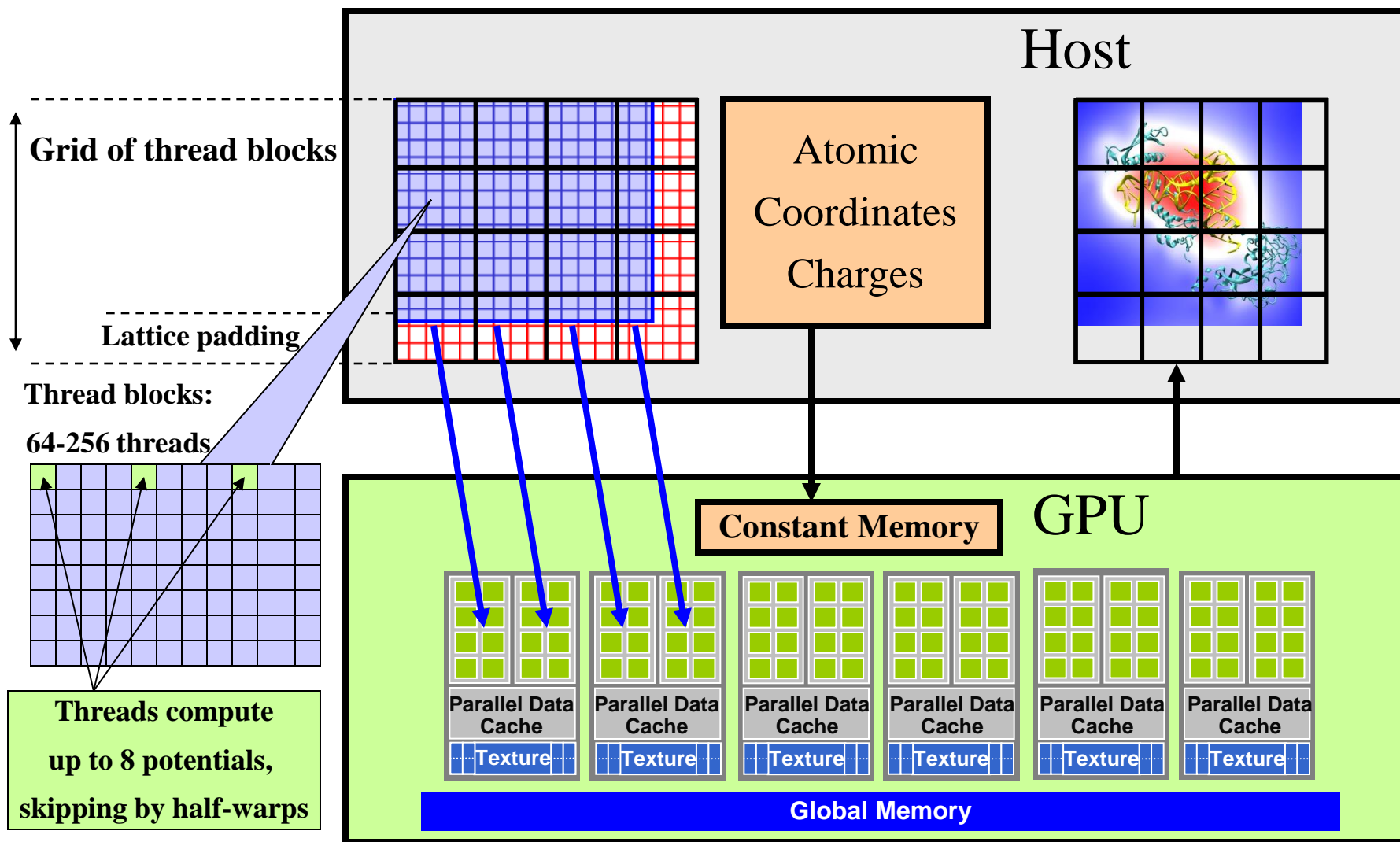
# DCS Observations for GPU Implementation

- Naive implementation has a low ratio of FP arithmetic operations to memory transactions (at least 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, coalesce global memory accesses, and overlap arithmetic 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

# CUDA DCS Implementation Overview

- Allocate and initialize potential map memory on host CPU
- Allocate potential map slice buffer on GPU
- Preprocess atom coordinates and charges
- Loop over slices:
  - Copy slice from host to GPU
  - Loop over groups of atoms until done:
    - Copy atom data to GPU
    - Run CUDA Kernel on atoms and slice resident on GPU accumulating new potential contributions into slice
  - Copy slice from GPU back to host
- Free resources

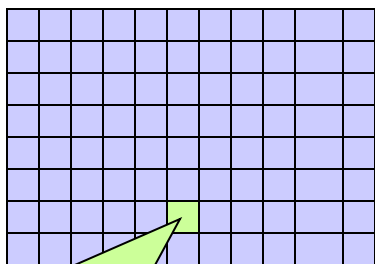
# Direct Coulomb Summation on the GPU



# DCS CUDA Block/Grid Decomposition (non-unrolled)

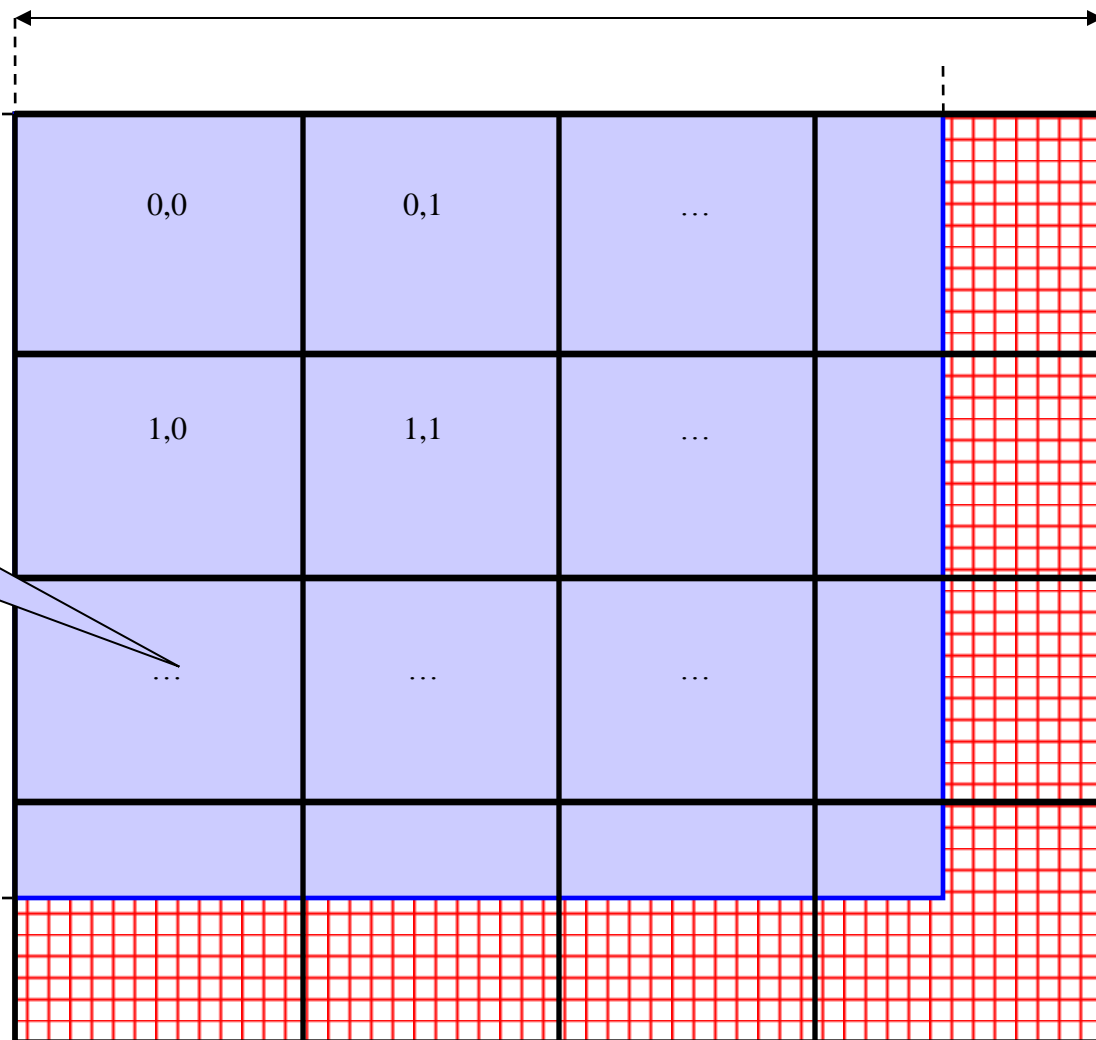
Grid of thread blocks:

Thread blocks:  
64-256 threads



Threads compute  
1 potential each

Padding waste



# DCS CUDA Block/Grid Decomposition (non-unrolled)

- 16x16 CUDA thread blocks are a nice starting size with a satisfactory number of threads
- Small enough that there's not much waste due to padding at the edges

# Notes on Benchmarking CUDA Kernels:

## Initialization Overhead

- When a host thread initially binds to a CUDA context, there is a small ( $\sim 100\text{ms}$ ) delay during the first CUDA runtime call that touches state on the device
- The first time each CUDA kernel is executed, there's a small delay while the driver compiles the device-independent PTX intermediate code for the physical device associated with the current context
- In most real codes, these sources of one-time initialization overhead would occur at application startup and should not be a significant factor.
- The exception to this is that newly-created host threads incur overhead when they bind to their device, so it's best to re-use existing host threads than to generate them repeatedly

# Notes on Benchmarking CUDA Kernels: Power Management, Async Operations

- Modern GPUs (and of course CPUs) incorporate power management hardware that reduces clock rates and/or powers down functional units when idle
- In order to benchmark peak performance of CUDA kernels, both the GPU(s) and CPU(s) must be awoken from their respective low-power modes
- In order to get accurate and repeatable timings, do a “warm up” pass prior to running benchmark timings on your kernel and any associated I/O
- Call `cudaThreadSynchronize()` prior to stopping timers to verify that any outstanding kernels and I/Os have completed

# DCS Version 1: Const+Precalc

## 187 GFLOPS, 18.6 Billion Atom Evals/Sec

(GeForce 8800 GTX)

- 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];  
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;
```

Start global memory reads early. Kernel hides some of its own latency.

Only dependency on global memory read is at the end of the kernel...

# DCS CUDA Block/Grid Decomposition (unrolled)

- Reuse atom data and partial distance components multiple times
- Use “unroll and jam” to unroll the outer loop into the inner loop
- Uses more registers, but increases arithmetic intensity significantly
- Kernels that unroll the inner loop 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

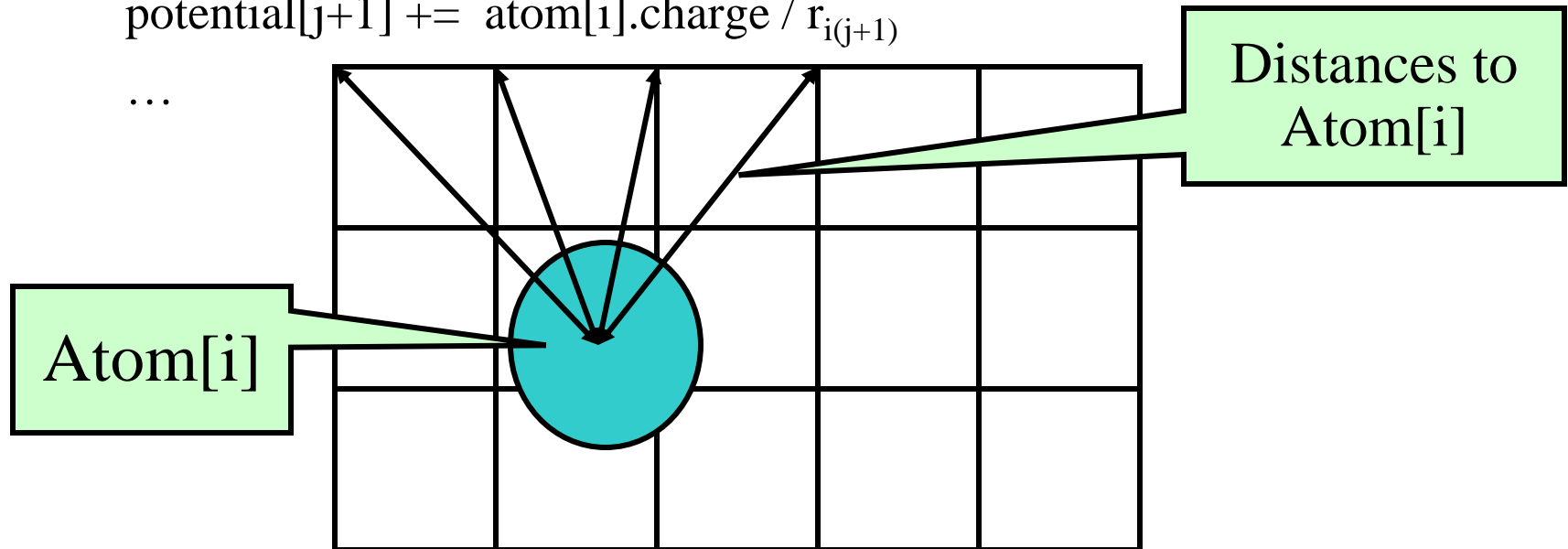
# DCS CUDA Algorithm: Unrolling Loops

- Add each atom's contribution to several lattice points at a time, distances only differ in one component:

potential[j] += atom[i].charge / r<sub>ij</sub>

potential[j+1] += atom[i].charge / r<sub>i(j+1)</sub>

...



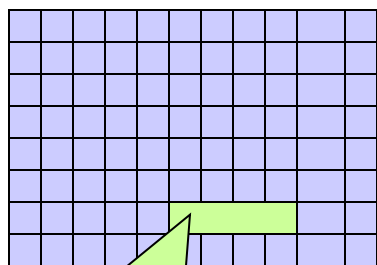
# DCS CUDA Block/Grid Decomposition

(unrolled)

Grid of thread blocks:

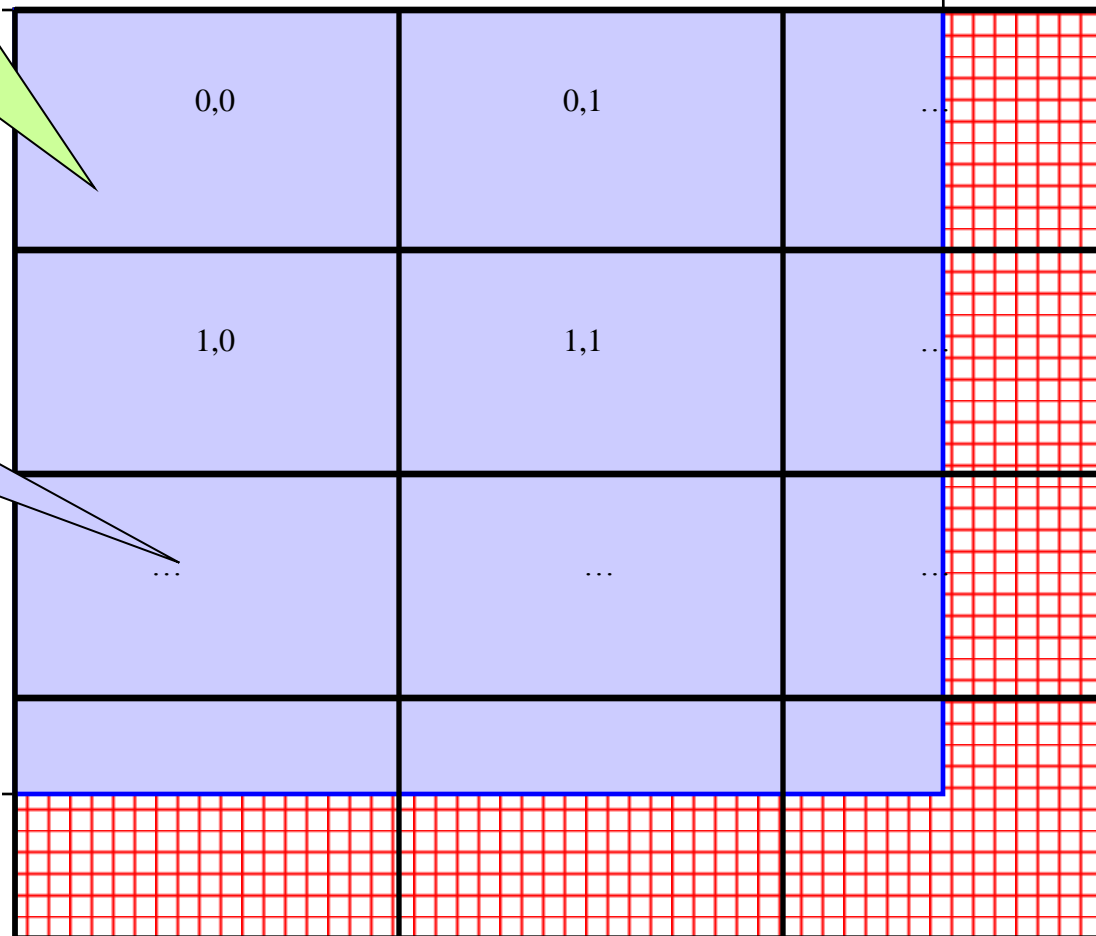
Unrolling increases  
computational tile size

Thread blocks:  
64-256 threads



Threads compute  
up to 8 potentials

Padding waste



# DCS Version 2: Const+Precalc+Loop Unrolling

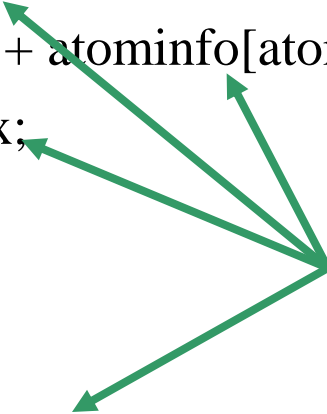
## 259 GFLOPS, 33.4 Billion Atom Evals/Sec

(GeForce 8800 GTX)

- Pros:
  - Although const memory is very fast, loading values into registers costs instruction slots
  - We can reduce the number of loads by reusing atom coordinate values for multiple voxels, by storing in regs
  - By unrolling the X loop by 4, we can compute  $dy^2 + dz^2$  once and use it multiple times, much like the CPU version of the code does
- Cons:
  - Compiler won't do this type of unrolling for us (yet)
  - Uses more registers, one of several finite resources
  - Increases effective tile size, or decreases thread count in a block, though not a problem at this level

# DCS Version 2: Inner Loop

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



Compared to non-unrolled kernel: memory loads are decreased by 4x, and FLOPS per evaluation are reduced, but register use is increased...

# DCS Version 3:

Const+Shared+Loop Unrolling+Precalc  
268 GFLOPS, 36.4 Billion Atom Evals/Sec

- Pros:
  - Loading prior potential values from global memory into shared memory frees up several registers, so we can afford to unroll by 8 instead of 4
  - Using fewer registers allows co-scheduling of more blocks, increasing GPU “occupancy”
- Cons:
  - Bumping against hardware limits (uses all const memory, most shared memory, and a largish number of registers)

# DCS Version 3: Kernel Structure

- Loads 8 potential map lattice points from global memory at startup, and immediately stores them into shared memory before going into inner loop. We would otherwise consume too many registers and lose performance (on GeForce 8800 at least...)
- Processes 8 lattice points at a time in the inner loop
- Additional performance gains are achievable by coalescing global memory reads at start/end



# DCS Version 3: Inner Loop

```
...for (v=0; v<8; v++)
    curenergies[tid + nthr * v] = energygrid[outaddr + v];
float coorx = gridspacing * xindex;
float coory = gridspacing * yindex;
float energyvalx1=0.0f; [.....] float energyvalx8=0.0f;
for (atomid=0; atomid<numatoms; atomid++) {
    float dy = coory - atominfo[atomid].y;
    float dysqpdzsq = (dy * dy) + atominfo[atomid].z;
    float dx = coorx - atominfo[atomid].x;
    energyvalx1 += atominfo[atomid].w * rsqrtf(dx*dx + dysqpdzsq);
    dx += gridspacing;
[...]
```

energyvalx8 += atominfo[atomid].w \* rsqrtf(dx\*dx + dysqpdzsq);

```
}
__syncthreads(); // guarantee that shared memory values are ready for reading by all threads
energygrid[outaddr    ] = energyvalx1 + curenergies[tid    ];
[...]
```

energygrid[outaddr + 7] = energyvalx2 + curenergies[tid + nthr \* 7];

# DCS Version 4:

## Const+Loop Unrolling+Coalescing

### 291.5 GFLOPS, 39.5 Billion Atom Evals/Sec

- Pros:
  - Simplified structure compared to version 3, no use of shared memory, register pressure kept at bay by doing global memory operations only at the end of the kernel
  - Using fewer registers allows co-scheduling of more blocks, increasing GPU “occupancy”
  - Doesn’t have as strict of a thread block dimension requirement as version 3, computational tile size can be smaller
- Cons:
  - The computation tile size is still large, so small potential maps don’t perform as well as large ones

# DCS Version 4: Kernel Structure

- Processes 8 lattice points at a time in the inner loop
- Subsequent lattice points computed by each thread are offset by a half-warp to guarantee coalesced memory accesses
- Loads and increments 8 potential map lattice points from global memory at completion of the summation, avoiding register consumption

# DCS Version 4: Inner Loop

```
...float coory = gridspacing * yindex;  
float coorx = gridspacing * xindex;  
float gridspacing_coalesce = gridspacing * BLOCKSIZE_X;  
int atomid;
```

Points spaced for  
memory coalescing

```
for (atomid=0; atomid<numatoms; atomid++) {
```

```
    float dy = coory - atominfo[atomid].y;  
    float dyz2 = (dy * dy) + atominfo[atomid].z;  
    float dx1 = coorx - atominfo[atomid].x;
```

Reuse partial distance  
components  $dy^2 + dz^2$

```
[...]
```

```
    float dx8 = dx7 + gridspacing_coalesce;  
    energyvalx1 += atominfo[atomid].w * rsqrtf(dx1*dx1 + dyz2);
```

```
[...]
```

```
    energyvalx8 += atominfo[atomid].w * rsqrtf(dx8*dx8 + dyz2);
```

```
}
```

```
energygrid[outaddr] += energyvalx1;
```

```
[...]
```

```
energygrid[outaddr+7*BLOCKSIZE_X] += energyvalx7;
```

Global memory ops  
occur only at the end  
of the kernel,  
decreases register use

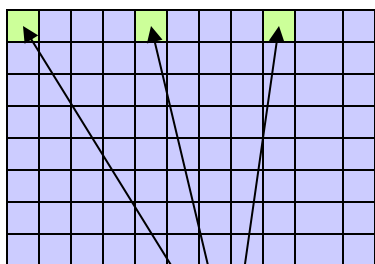
# DCS CUDA Block/Grid Decomposition

(unrolled, coalesced)

Grid of thread blocks:

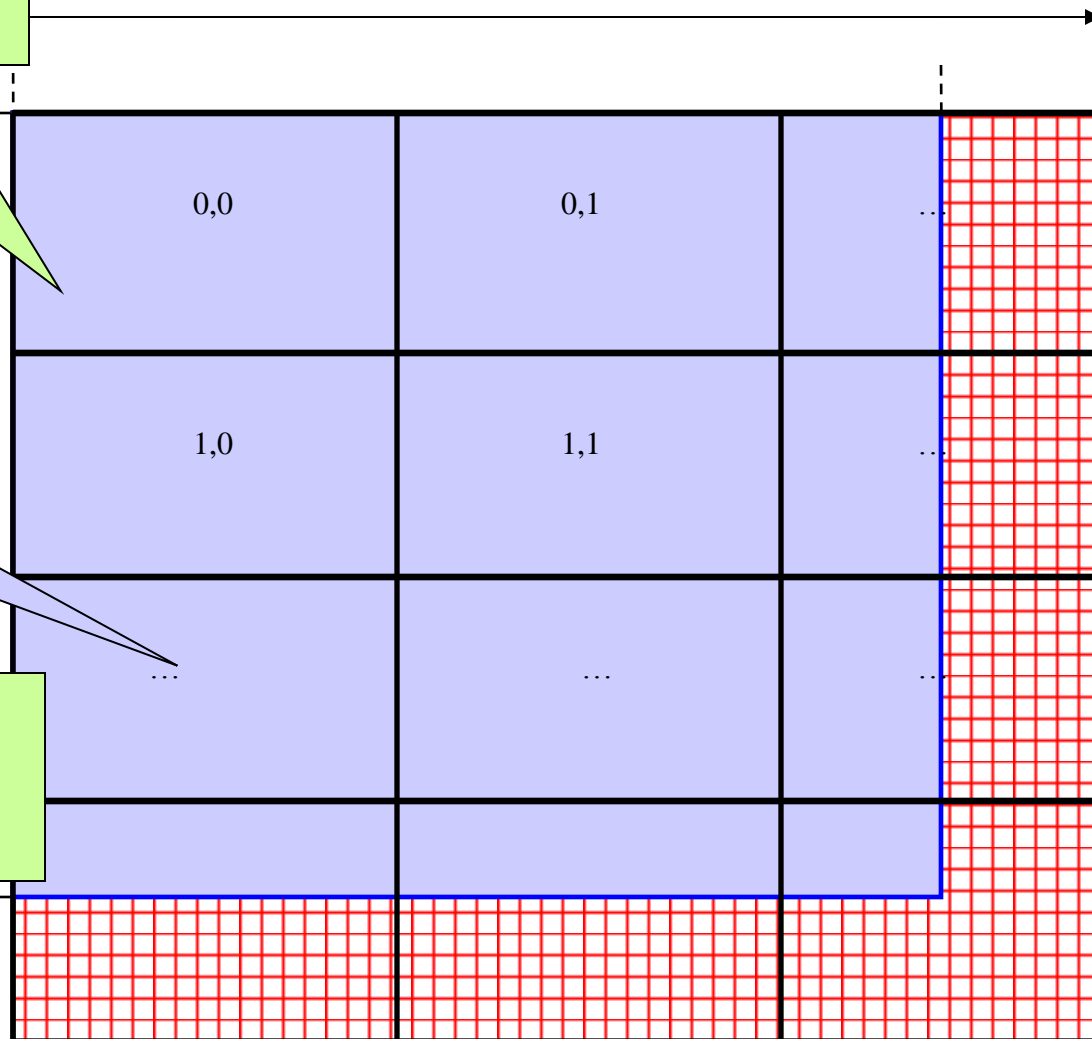
Unrolling increases  
computational tile size

Thread blocks:  
64-256 threads

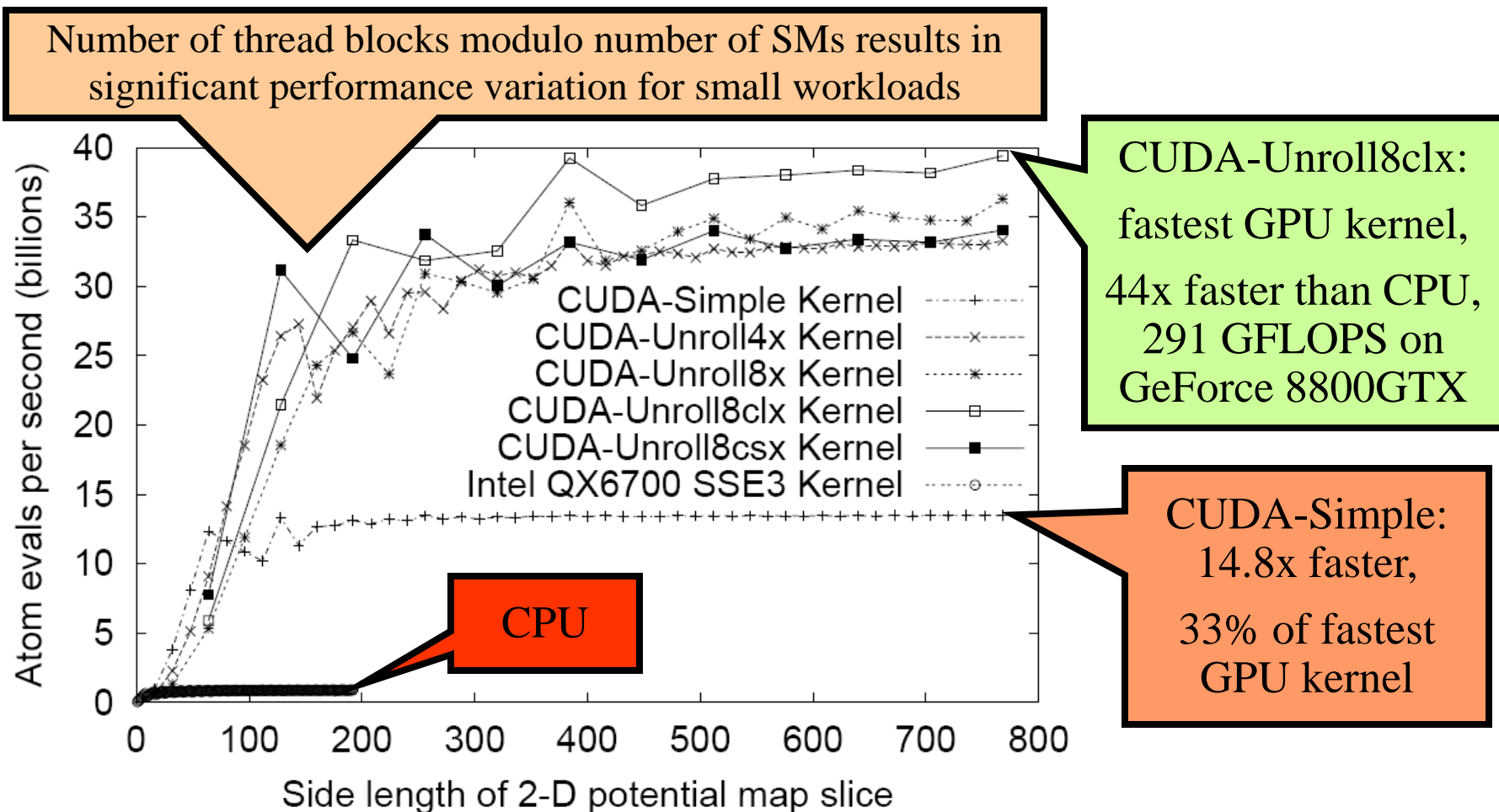


Threads compute  
up to 8 potentials,  
skipping by half-warps

Padding waste



# Direct Coulomb Summation Performance



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 Potential Map Calculation

- Both CPU and GPU versions of the code are easily parallelized by decomposing the 3-D potential map into slices, and computing them concurrently
- Potential maps often have 50-500 slices in the Z direction, so plenty of coarse grain parallelism is still available via the DCS algorithm

# 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 (GeForce 8800 GTX)

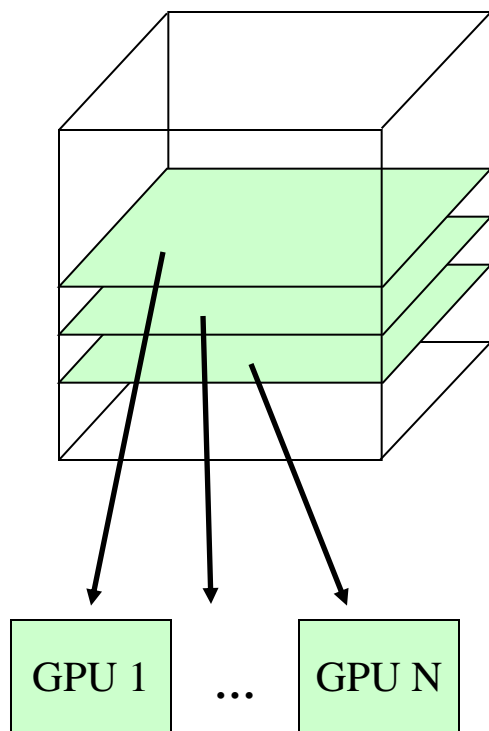
- Effective memory bandwidth scales with the number of GPUs utilized
- PCIe bus bandwidth not a bottleneck for this algorithm
- 117 billion evals/sec
- 863 GFLOPS
- 131x speedup vs. CPU core
- Power: 700 watts during benchmark



Quad-core Intel QX6700

Three NVIDIA GeForce 8800GTX

# Multi-GPU Direct Coulomb Summation



NCSA GPU Cluster

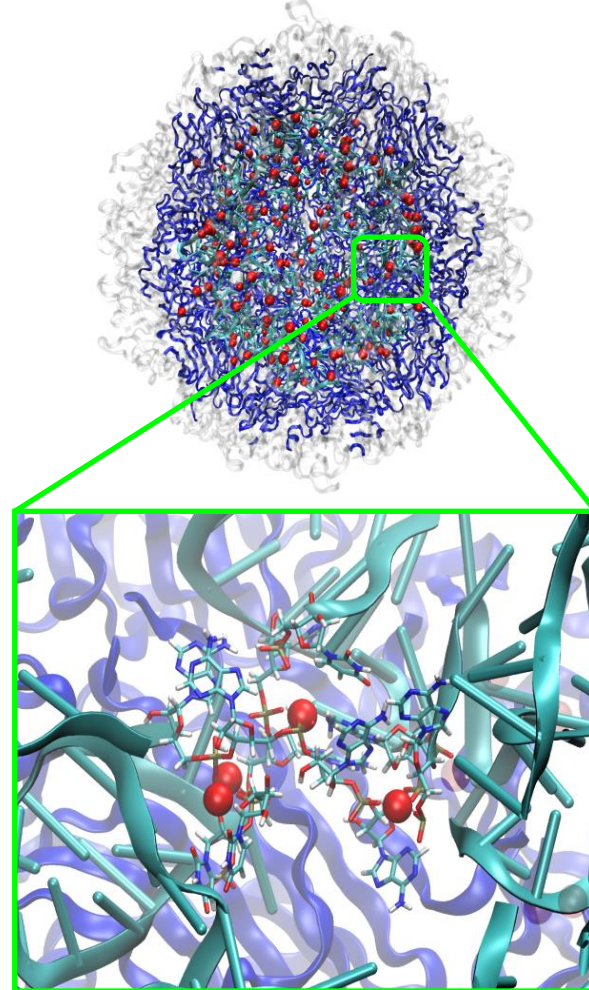
<http://www.ncsa.uiuc.edu/Projects/GPUcluster/>

	Evals/sec	TFLOPS	Speedup*
4-GPU (2 Quadroplex) Opteron node at NCSA	157 billion	1.16	176
4-GPU GTX 280 (GT200)	241 billion	1.78	271

\*Speedups relative to Intel QX6700 CPU core w/ SSE

# Multi-GPU DCS Performance: Initial Ion Placement Lattice Calculation

- Original virus DCS ion placement ran for 110 CPU-hours on SGI Altix Itanium2
- Same calculation now takes 1.35 GPU-hours
- 27 minutes (wall clock) if three GPUs are used concurrently
- CUDA Initial ion placement lattice calculation performance:
  - 82 times faster for virus (STMV) structure
  - 110 times faster for ribosome
- Three GPUs give performance equivalent to ~330 SGI Altix CPUs for the ribosome case



Satellite Tobacco Mosaic Virus (STMV)

Ion Placement

# Brief Shared Memory Example: Multiple Debye-Hückel Electrostatics

- Part of Poisson-Boltzmann solver in the popular APBS electrostatics solver package
- Method: compute electrostatic potentials at grid points on boundary faces of box containing molecule
- Screening function:

$$S(r) = \frac{e^{-\kappa(r-\sigma_j)}}{1 + \kappa\sigma_j}$$

# Shared memory: MDH Kernel (CUDA)

```
extern shared float smem [ ] ;
```

```
int igrid = (blockIdx .x  blockDim.x ) + threadIdx .x ;    int lsize = blockDim.x ;    int lid= threadIdx .x ;
```

```
float lgx = gx [ igrid ] ; float lgy = gy [ igrid ] ; float lgz = gz [ igrid ] ; float v = 0.0 f ;
```

```
for ( int jatom = 0 ; jatom < natoms ; jatom+=lsize ) {
```

```
    __syncthreads ( ) ;
```

```
    if ( ( jatom + lid ) < natoms ) {
```

```
        smem[ lid          ] = ax [ jatom + lid] ;
```

```
        smem[ lid +      lsize ] = ay [ jatom + lid] ;
```

```
        smem[ lid + 2 * lsize ] = az [ jatom + lid] ;
```

```
        smem[ lid + 3 * lsize ] = charge [ jatom + lid] ;
```

```
        smem[ lid + 4 * lsize ] = size [ jatom + lid] ;
```

```
    }
```

```
    __syncthreads ( ) ;
```

```
    if ( ( jatom+l s i z e ) > natoms ) l s i z e = natoms - jatom ;
```

```
    for ( int i=0; i<l s i z e ; i++) {
```

```
        float dx = lgx - smem[ i          ] ;
```

```
        float dy = lgy - smem[ i +      lsize ] ;
```

```
        float dz = lgz - smem[ i + 2 * lsize ] ;
```

```
        float dist = sqrtf ( dxdx + dydy + dzdz ) ;
```

```
        v += smem[i+3*lsize] * expf(-xkappa ( dist - smem[ i+4*lsize ] ) ) / (1.0 f + xkappa smem[ i+4*lsize ] ) * dist) ;
```

```
    }
```

```
}
```

```
val [ igrid ] = pre1 * v;
```

Collectively load atoms from global memory into shared memory

Loop over all all atoms in shared memory accumulating potential contributions into grid points



# 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)

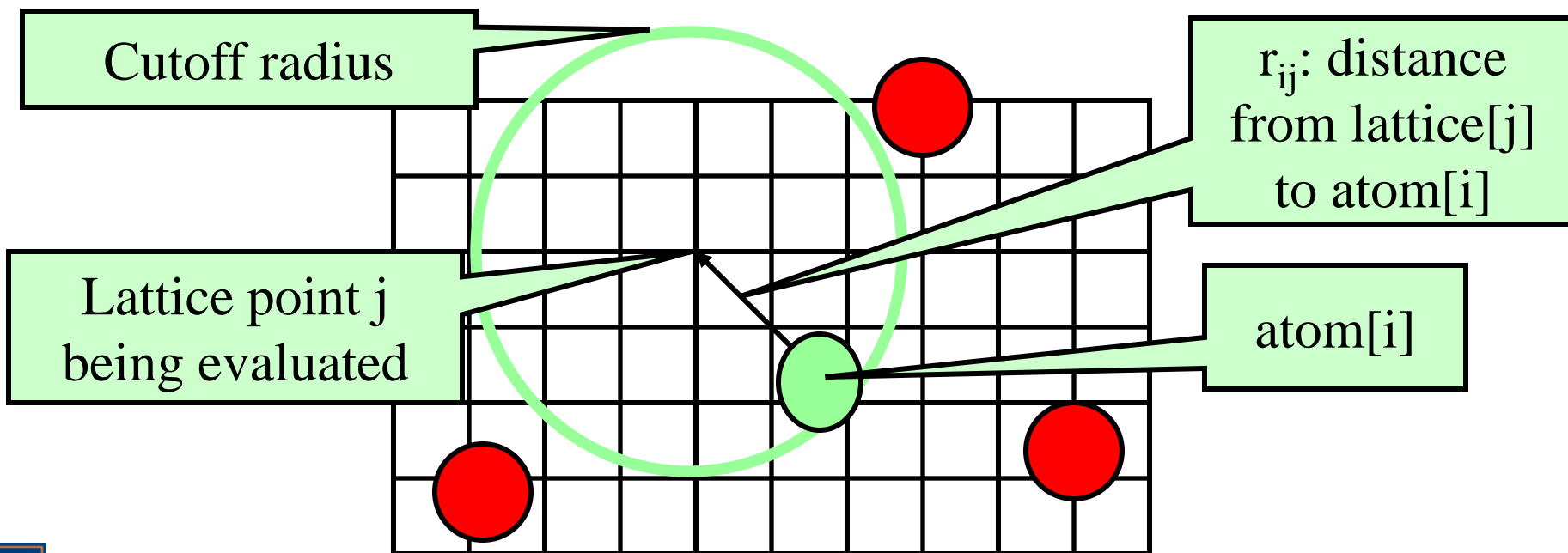
# Short-Range Cutoff Summation

- Each lattice point accumulates electrostatic potential contribution from atoms within cutoff distance:

if ( $r_{ij} < \text{cutoff}$ )

$\text{potential}[j] += (\text{charge}[i] / r_{ij}) * s(r_{ij})$

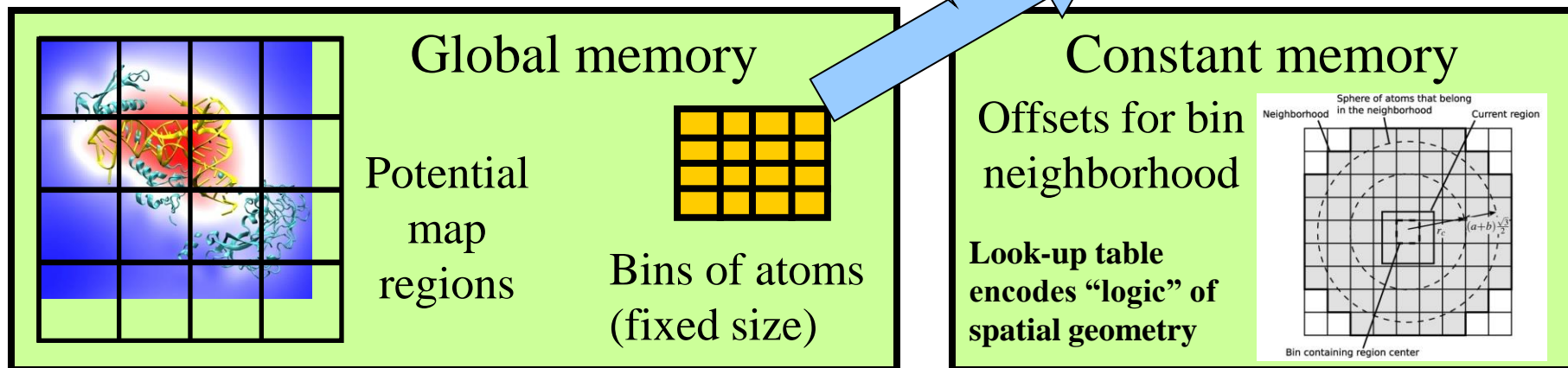
- Smoothing function  $s(r)$  is algorithm dependent



# CUDA Cutoff Electrostatic Potential Summation

- Atoms are spatially hashed into **fixed-size bins** (guarantees coalescing)
- CPU handles overflowed bins (GPU kernel can be very aggressive)
- GPU thread block calculates corresponding region of potential map,
- GPU bin/region neighbor checks are costly; solved with universal table look-up

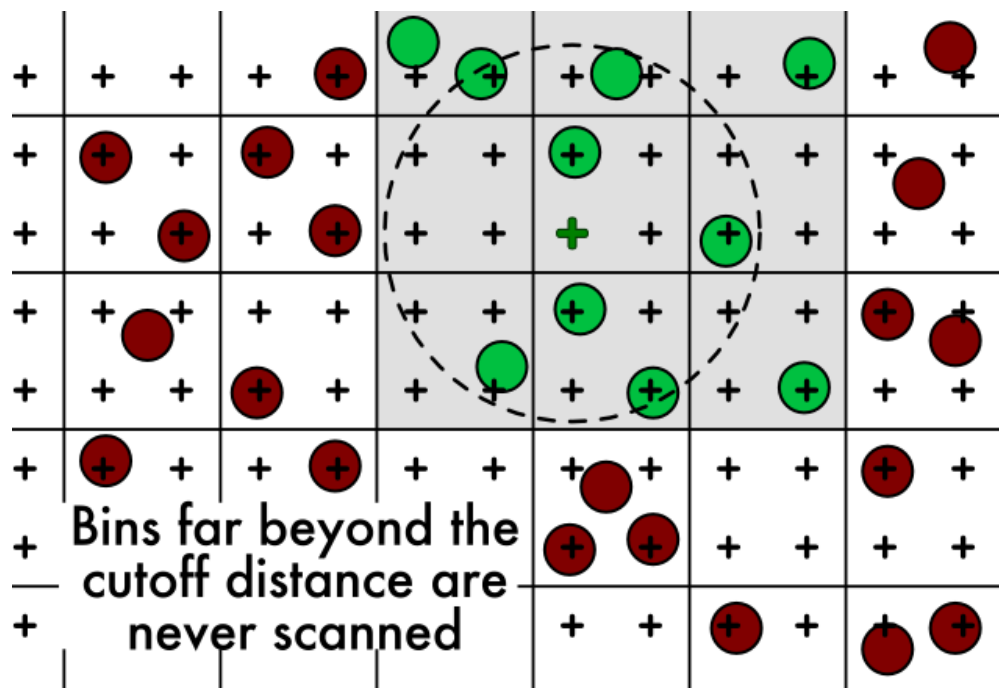
Each thread block cooperatively loads atom bins from surrounding neighborhood into shared memory for evaluation: **GATHER**





# Spatial Sorting of Atoms Into “Bins”

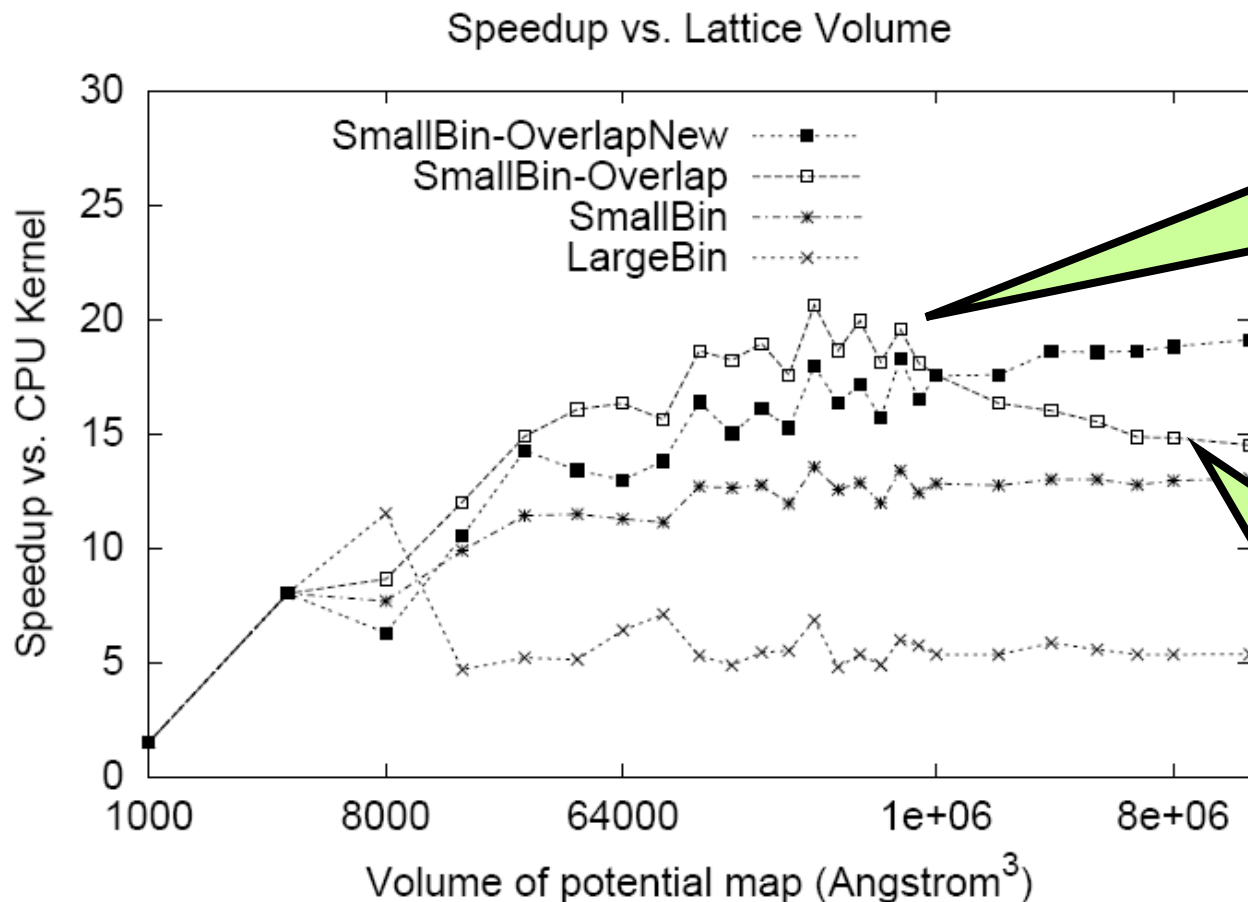
- Sort atoms into *bins* by their coordinates
- Each bin is sized to guarantee GPU memory coalescing
- Each bin holds up to 8 atoms, containing 4 FP values (3 coords, 1 charge)
- Each lattice point **gathers** potentials from atom bins within cutoff



# Using the CPU to Optimize 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
  - Use fixed size bin data structures, with “empty” slots skipped or producing zeroed out results
  - Handle exceptional or irregular work units on the CPU; GPU processes the bulk of the work concurrently
  - On average, the GPU is kept highly occupied, attaining a high fraction of peak performance

# Cutoff Summation Runtime



GPU cutoff with  
CPU overlap:

17x-21x faster than  
CPU core

If asynchronous  
stream blocks due to  
queue filling,  
performance will  
degrade from  
peak...

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.

# Cutoff Summation Observations

- Use of CPU to handle overflowed bins is very effective, overlaps completely with GPU work
- One caveat when using streaming API is to avoid overfilling the stream queue with work, as doing so can trigger blocking behavior (greatly improved in current drivers)
- The use of compensated summation (all GPUs) or double-precision (SM  $\geq 1.3$  only) for potential accumulation resulted in only a  $\sim 10\%$  performance penalty vs. pure single-precision arithmetic, while reducing the effects of floating point truncation

# Multilevel Summation

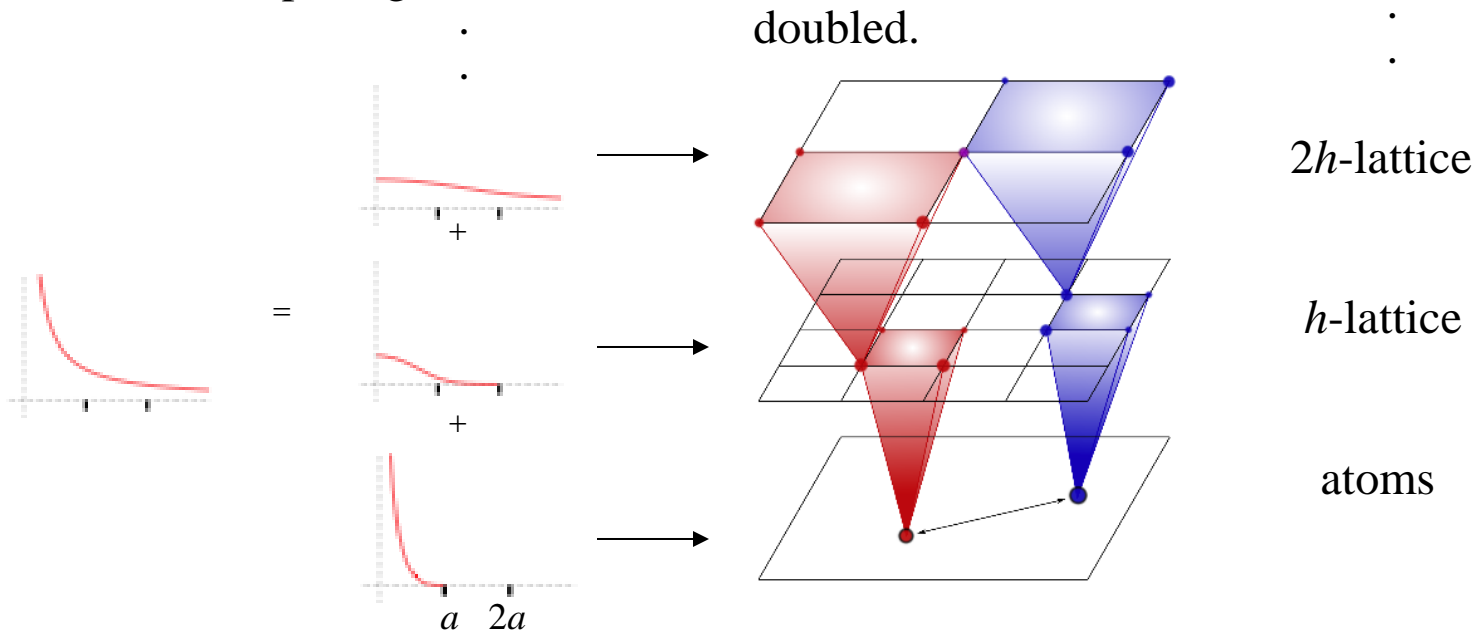
- Approximates full electrostatic potential
- Calculates sum of smoothed pairwise potentials interpolated from a hierarchy of lattices
- Advantages over PME and/or FMM:
  - Algorithm has linear time complexity
  - Permits non-periodic and periodic boundaries
  - Produces continuous forces for dynamics (advantage over FMM)
  - Avoids 3-D FFTs for better parallel scaling (advantage over PME)
  - Spatial separation allows use of multiple time steps
  - Can be extended to other pairwise interactions
- Skeel, Tezcan, Hardy, *J Comp Chem*, 2002 — Computing forces for molecular dynamics
- Hardy, Stone, Schulten, *J Paral Comp*, 2009 — GPU-acceleration of potential map calculation

# Multilevel Summation Main Ideas

## Split the $1/r$ potential

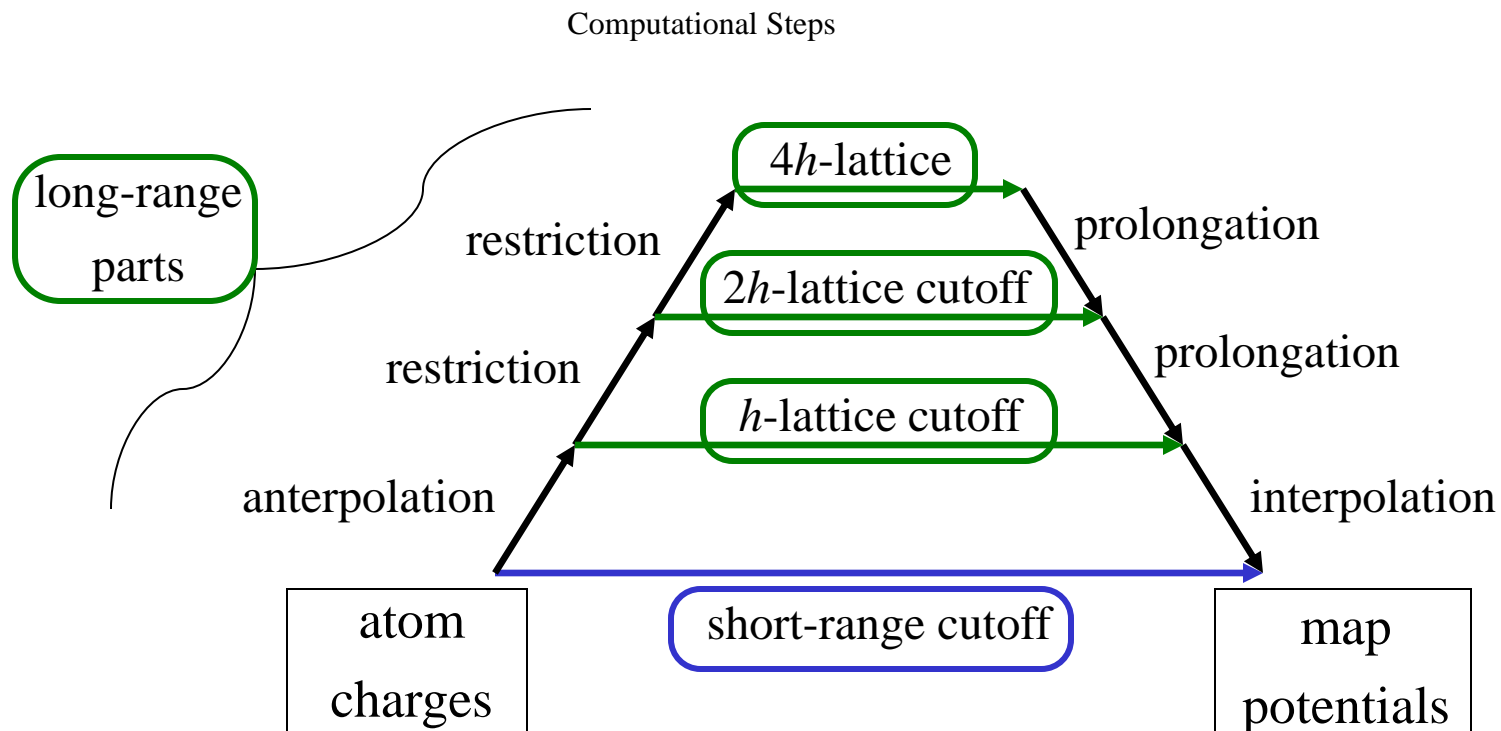
## Interpolate the smoothed potentials

- Split the  $1/r$  potential into a short-range cutoff part plus smoothed parts that are successively more slowly varying. All but the top level potential are cut off.
- The smoothed potentials are interpolated from successively coarser lattices.
- The lattice spacing is doubled at each successive level. The cutoff distance is also doubled.



# Multilevel Summation Calculation

$$\text{map potential} = \text{exact short-range interactions} + \text{interpolated long-range interactions}$$

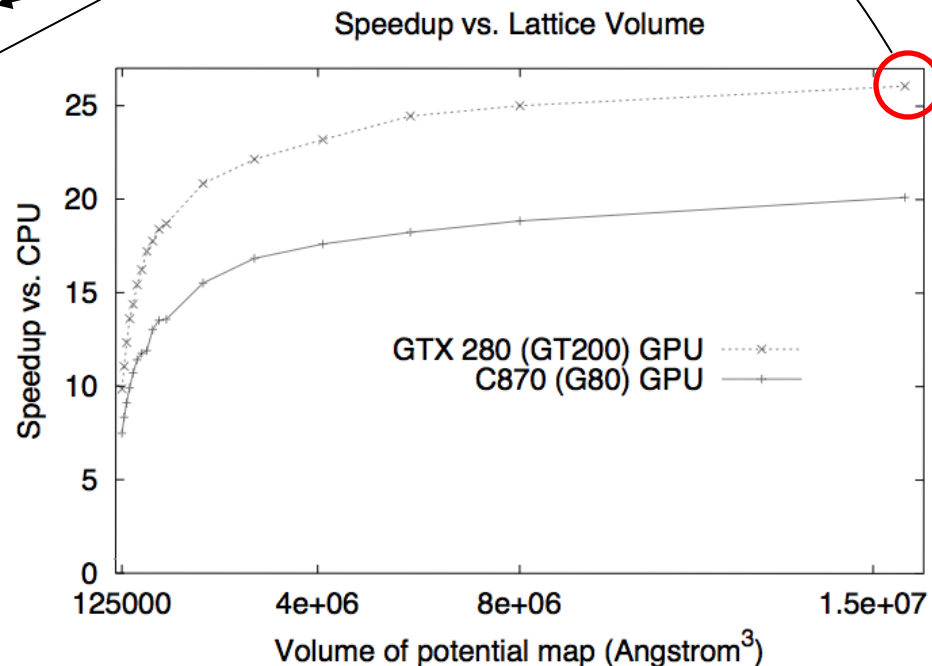


# Multilevel Summation on the GPU

Accelerate **short-range cutoff** and **lattice cutoff** parts

Performance profile for 0.5 Å map of potential for 1.5 M atoms.  
Hardware platform is Intel QX6700 CPU and NVIDIA GTX 280.

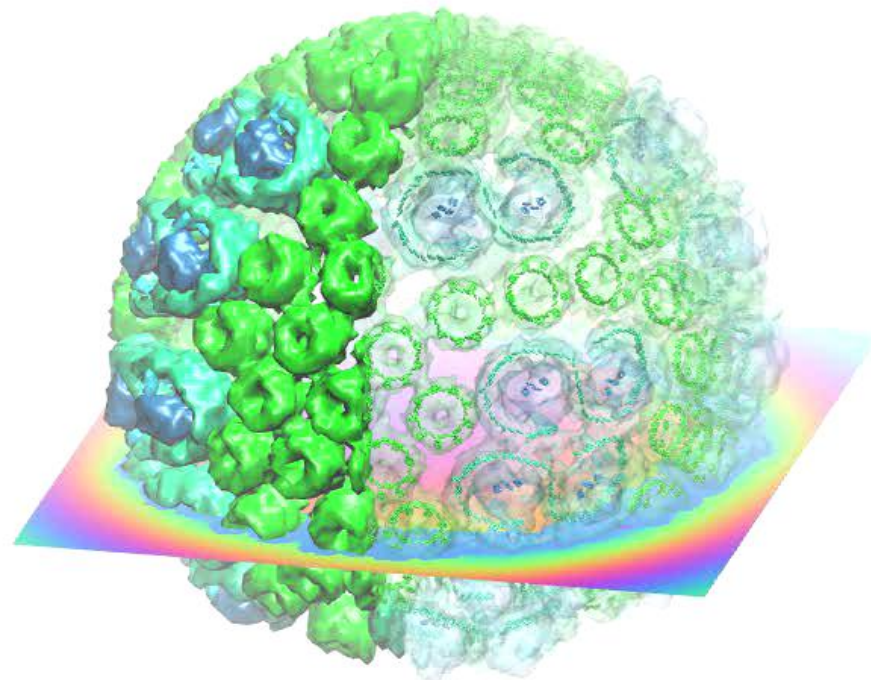
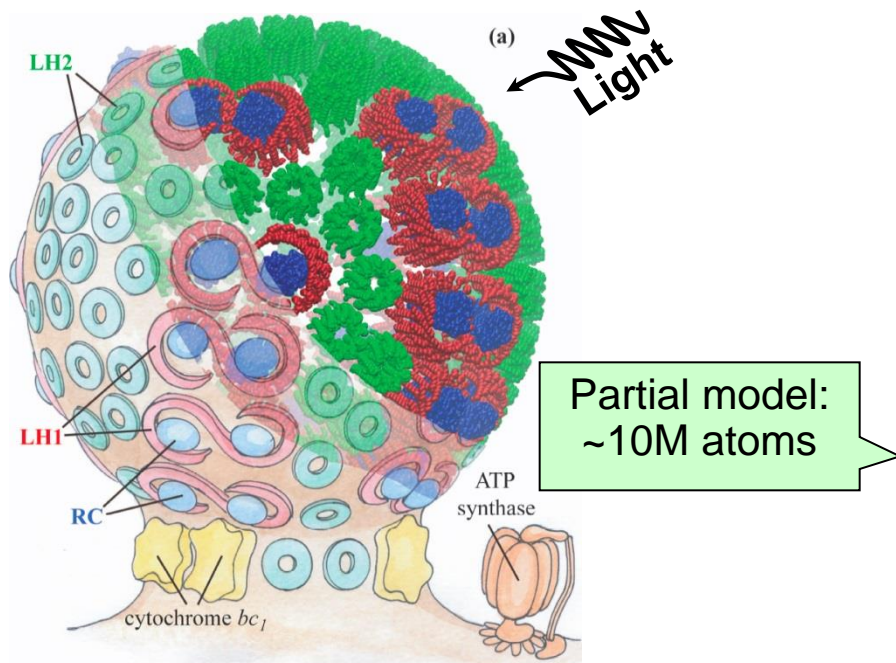
Computational steps	CPU (s)	w/ GPU (s)	Speedup
Short-range cutoff	480.07	14.87	32.3
Long-range antepolation	0.18		
restriction	0.16		
lattice cutoff	49.47	1.36	36.4
prolongation	0.17		
interpolation	3.47		
Total	533.52	20.21	26.4





# Photobiology of Vision and Photosynthesis

Investigations of the chromatophore, a photosynthetic organelle



Electrostatics needed to build full structural model, place ions, study macroscopic properties

Electrostatic field of chromatophore model  
from multilevel summation method:  
computed with 3 GPUs (G80) in ~90 seconds,  
46x faster than single CPU

# 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 programming
- 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

# Multi-core CPUs, Accelerators and Production Software

- A few of my rants about the ongoing state of parallel programming, accelerators...
- Currently, a programmer writes multiple codes for the same kernel, e.g. pthreads, MPI, CUDA, SSE, straight C, ...
- Error, exception handling in a multi-kernel environment can be quite tricky, particularly if buried within child threads, with various other operations in-flight already

# Multi-core CPUs, Accelerators and Production Software (2)

- Current APIs are composable, but can get quite messy:
  - Combinatorial expansion of multiple APIs and techniques leads to significant code bloat
  - Pure library-based interfaces are particularly unwieldy due to code required for packing/unpacking function parameters
  - Simple pthreads code can quickly bloat to hundreds of lines of code if there are many thread-specific memory allocations, parameters, etc to deal with
- Current systems do very little with NUMA topology info, CPU affinity, etc

# Acknowledgements

- Theoretical and Computational Biophysics Group, University of Illinois at Urbana-Champaign
- NCSA Blue Waters Team
- NCSA Innovative Systems Lab
- NVIDIA CUDA Center of Excellence, University of Illinois at Urbana-Champaign
- The CUDA team at NVIDIA
- NIH support: P41-RR005969

# GPU Computing Publications

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

- **Lattice Microbes: High-performance stochastic simulation method for the reaction-diffusion master equation.**  
E. Roberts, J. E. Stone, and Z. Luthey-Schulten.  
*J. Computational Chemistry* 34 (3), 245-255, 2013.
- **Fast Visualization of Gaussian Density Surfaces for Molecular Dynamics and Particle System Trajectories.** M. Krone, J. E. Stone, T. Ertl, and K. Schulten. *EuroVis Short Papers*, pp. 67-71, 2012.
- **Immersive Out-of-Core Visualization of Large-Size and Long-Timescale Molecular Dynamics Trajectories.** J. Stone, K. Vandivort, and K. Schulten. G. Bebis et al. (Eds.): *7th International Symposium on Visual Computing (ISVC 2011)*, LNCS 6939, pp. 1-12, 2011.
- **Fast Analysis of Molecular Dynamics Trajectories with Graphics Processing Units – Radial Distribution Functions.** B. Levine, J. Stone, and A. Kohlmeyer. *J. Comp. Physics*, 230(9):3556-3569, 2011.

# GPU Computing Publications

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

- **Quantifying the Impact of GPUs on Performance and Energy Efficiency in HPC Clusters.** J. Enos, C. Steffen, J. Fullop, M. Showerman, G. Shi, K. Esler, V. Kindratenko, J. Stone, J Phillips. *International Conference on Green Computing*, pp. 317-324, 2010.
- **GPU-accelerated molecular modeling coming of age.** J. Stone, D. Hardy, I. Ufimtsev, K. Schulten. *J. Molecular Graphics and Modeling*, 29:116-125, 2010.
- **OpenCL: A Parallel Programming Standard for Heterogeneous Computing.** J. Stone, D. Gohara, G. Shi. *Computing in Science and Engineering*, 12(3):66-73, 2010.
- **An Asymmetric Distributed Shared Memory Model for Heterogeneous Computing Systems.** I. Gelado, J. Stone, J. Cabezas, S. Patel, N. Navarro, W. Hwu. *ASPLOS '10: Proceedings of the 15<sup>th</sup> International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 347-358, 2010.



# GPU Computing Publications

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

- **GPU Clusters for High Performance Computing.** V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, W. Hwu. *Workshop on Parallel Programming on Accelerator Clusters (PPAC)*, In Proceedings IEEE Cluster 2009, pp. 1-8, Aug. 2009.
- **Long time-scale simulations of in vivo diffusion using GPU hardware.** E. Roberts, J. Stone, L. Sepulveda, W. Hwu, Z. Luthey-Schulten. In *IPDPS'09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Computing*, pp. 1-8, 2009.
- **High Performance Computation and Interactive Display of Molecular Orbitals on GPUs and Multi-core CPUs.** J. Stone, J. Saam, D. Hardy, K. Vandivort, W. Hwu, K. Schulten, *2nd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-2)*, *ACM International Conference Proceeding Series*, volume 383, pp. 9-18, 2009.
- **Probing Biomolecular Machines with Graphics Processors.** J. Phillips, J. Stone. *Communications of the ACM*, 52(10):34-41, 2009.
- **Multilevel summation of electrostatic potentials using graphics processing units.** D. Hardy, J. Stone, K. Schulten. *J. Parallel Computing*, 35:164-177, 2009.

# GPU Computing 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*, IEEE Press, 2008.
- **GPU acceleration of cutoff pair potentials for molecular modeling applications.** C. Rodrigues, D. Hardy, J. Stone, K. Schulten, and 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.