

# VSCSE Summer School 2009

## Many-core Processors for Science and Engineering Applications

### Lecture 8: Application Case Study – Accelerating Molecular Dynamics Experimentation

Guest Lecture by John Stone

Theoretical and Computational Biophysics Group

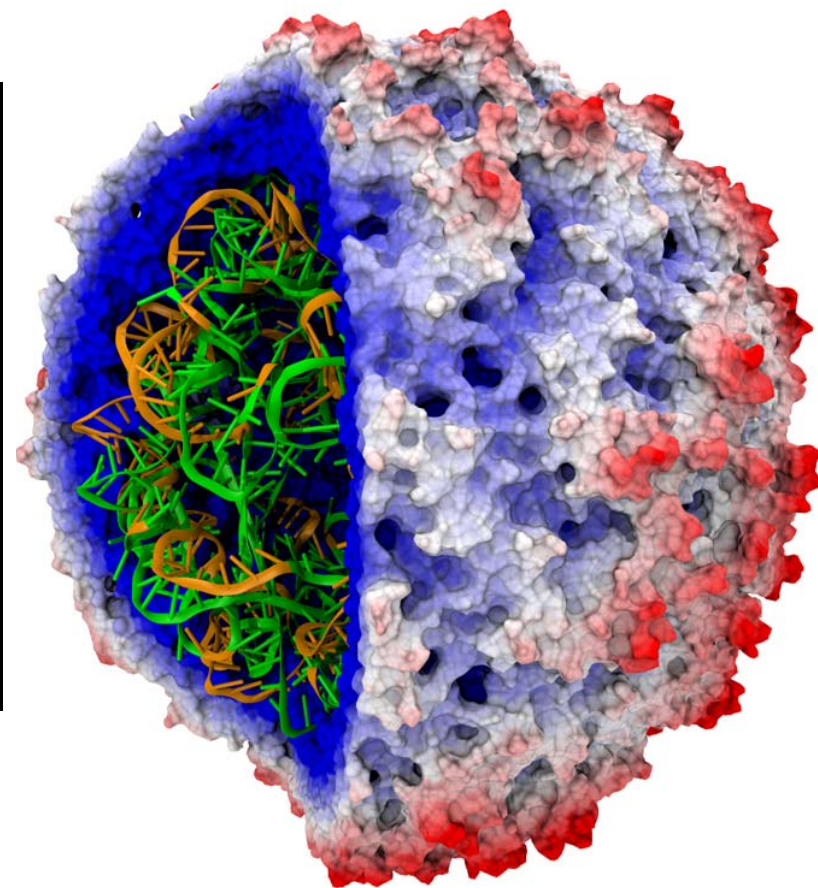
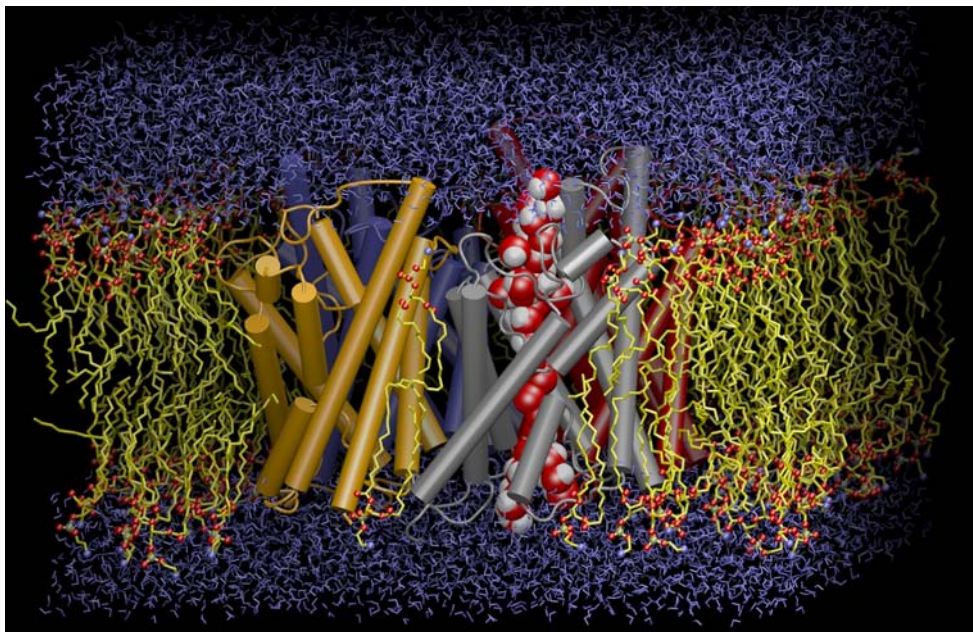
NIH Resource for Macromolecular Modeling and Bioinformatics

Beckman Institute for Advanced Science and Technology

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

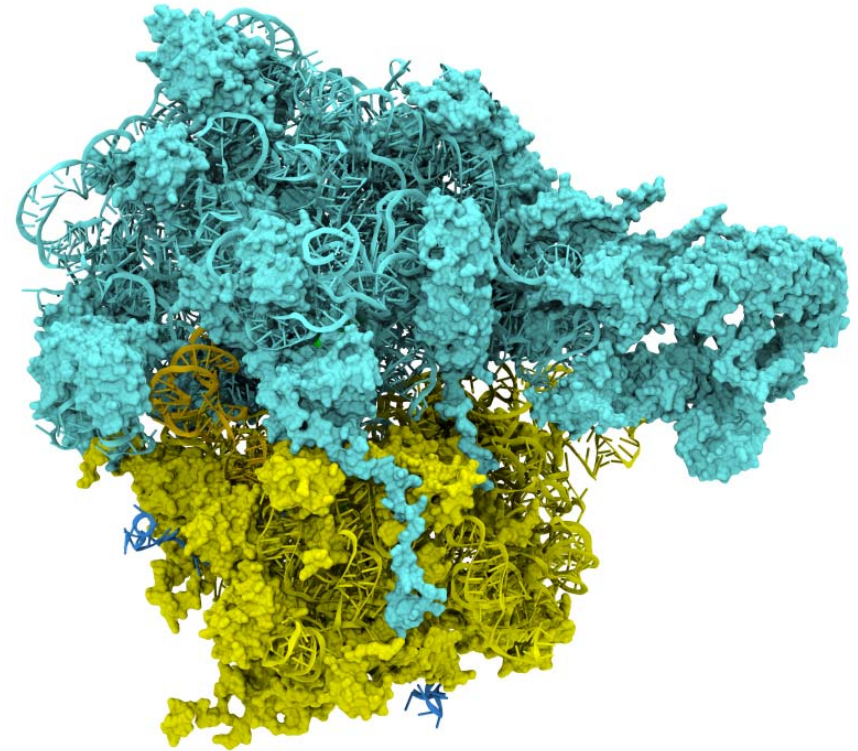
# VMD – “Visual Molecular Dynamics”

- Visualization and analysis of molecular dynamics simulations, sequence data, volumetric data, quantum chemistry simulations, particle systems, ...
- User extensible with scripting and plugins
- <http://www.ks.uiuc.edu/Research/vmd/>



# Integrating CUDA Kernels Into VMD

- VMD: molecular visualization and analysis
- State-of-the-art simulations require more viz/analysis power than ever before
- For some algorithms, CUDA can bring what was previously supercomputer class performance to an appropriately equipped desktop workstation



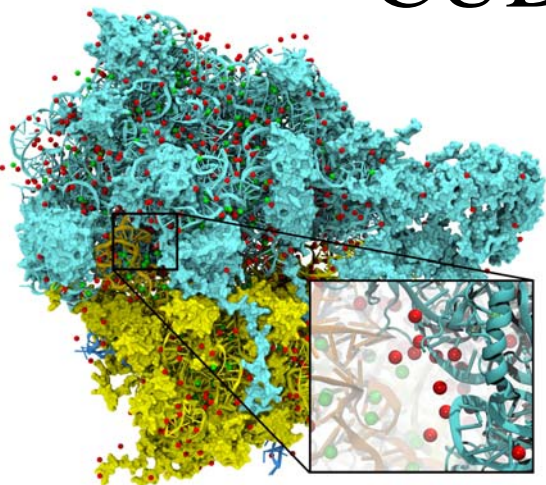
Ribosome: 260,790 atoms  
before adding solvent/ions

# Range of VMD Usage Scenarios

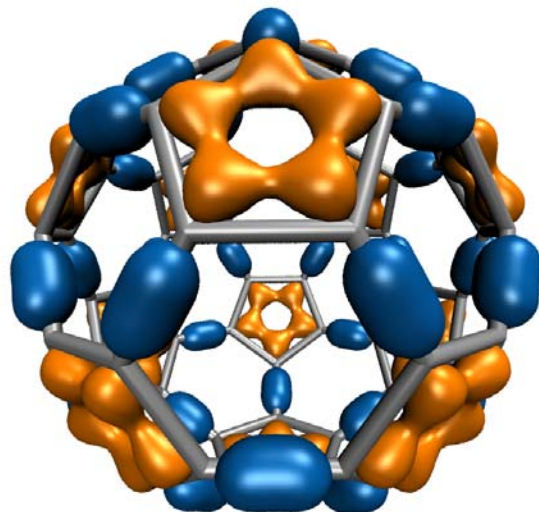
- Users run VMD on a diverse range of hardware: laptops, desktops, clusters, and supercomputers
- Typically used as a desktop science application, for interactive 3D molecular graphics and analysis
- Can also be run in pure text mode for numerically intensive analysis tasks, batch mode movie rendering, etc...
- GPU acceleration provides an opportunity to make some **slow, or batch** calculations capable of being run **interactively, or on-demand...**



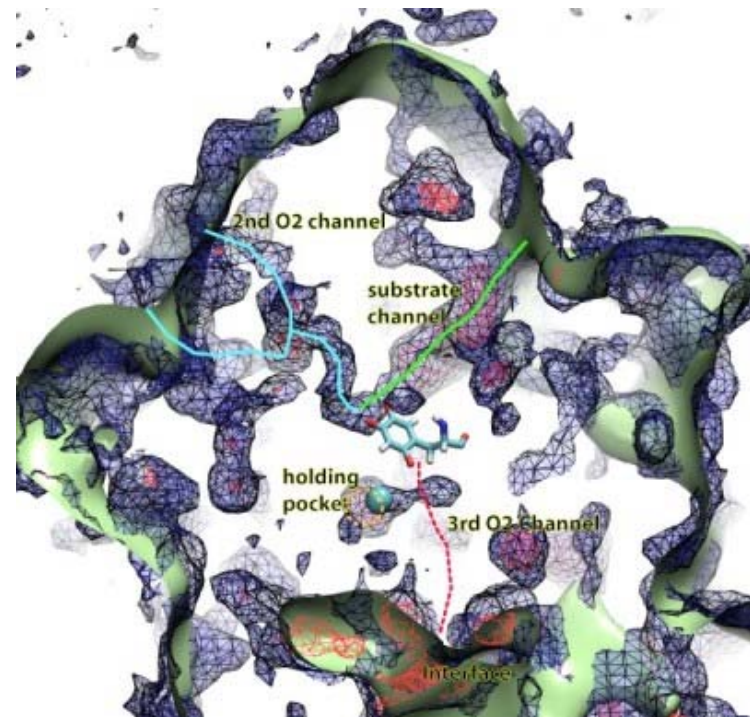
# CUDA Acceleration in VMD



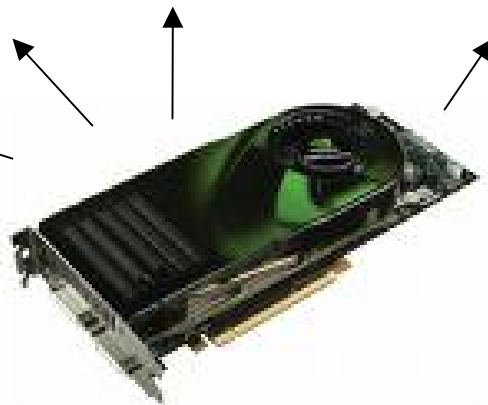
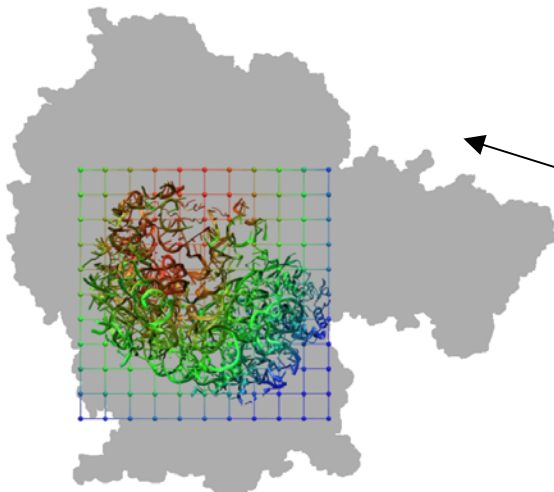
Electrostatic field calculation, ion placement:  
factor of 20x to 44x faster



Molecular orbital calculation and display:  
factor of 120x faster



Imaging of gas migration pathways in proteins with implicit ligand sampling:  
factor of 20x to 30x faster

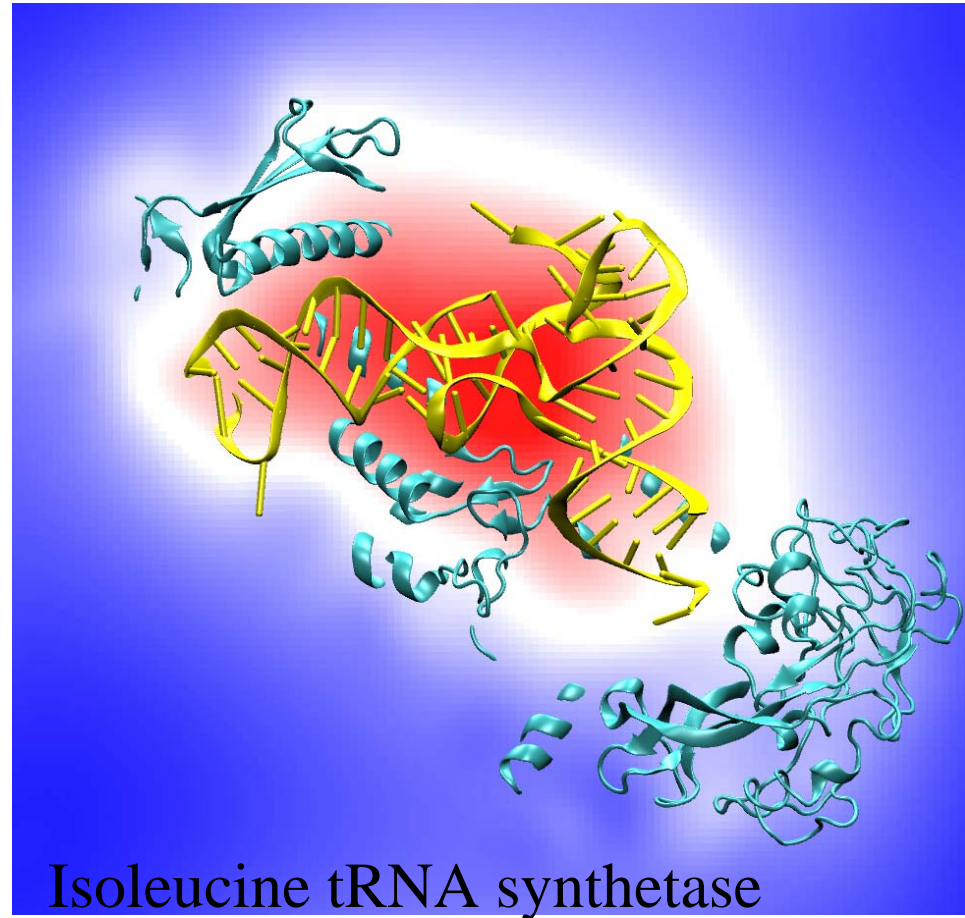


# Electrostatic Potential Maps

- 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



# Overview of Direct Coulomb Summation (DCS) Algorithm

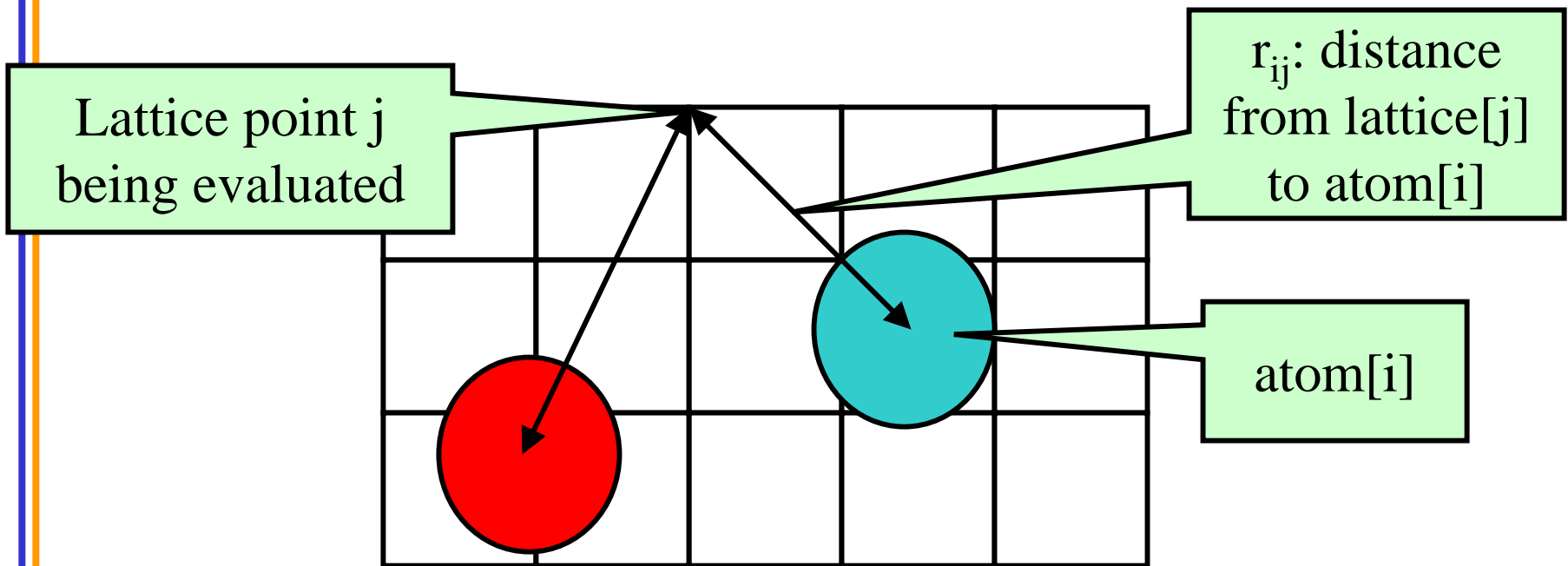
- One of several ways to compute the electrostatic potentials on a grid, ideally suited for the GPU
- Methods such as multilevel summation can achieve much higher performance at the cost of additional complexity
- Begin with DCS for computing electrostatic maps:
  - conceptually simple algorithm well 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 gridspace, float z, const float *atoms,
            int numatoms) {
    int i,j,n;
    int atomarrdim = numatoms * 4;
    for (j=0; j<grid.y; j++) {
        float y = gridspace * (float) j;
        for (i=0; i<grid.x; i++) {
            float x = gridspace * (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

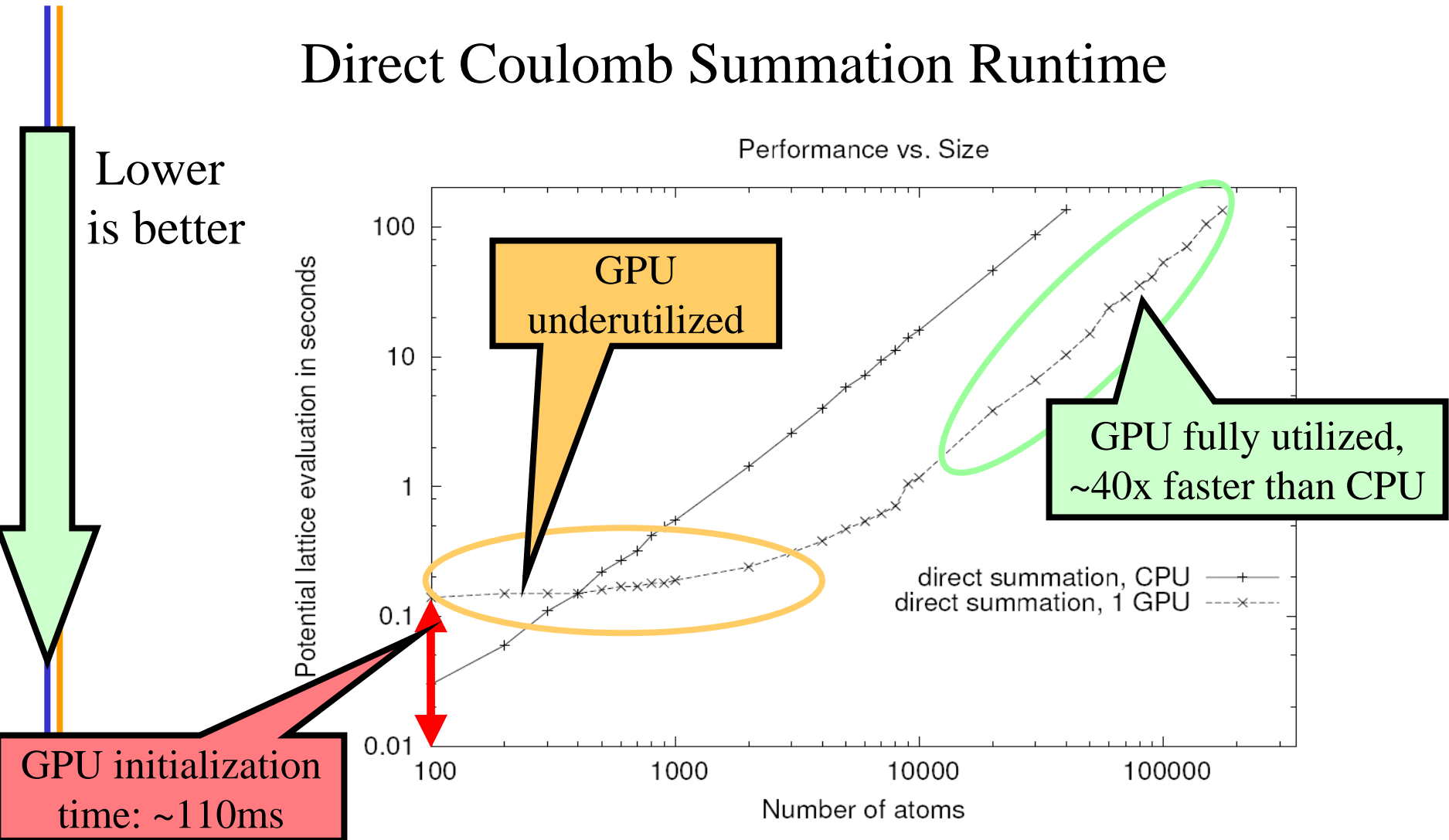
- Electrostatic maps used for ion placement require evaluation of ~20 potential lattice points per atom for a typical biological structure
- Atom list has the smallest memory footprint, best choice for the inner loop (both CPU and GPU)
- Lattice point coordinates are computed on-the-fly
- Atom coordinates 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

# 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 current GPUs
- Analyze the real-world use case for the problem and select the kernel for the problem sizes that will be heavily used



# 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.

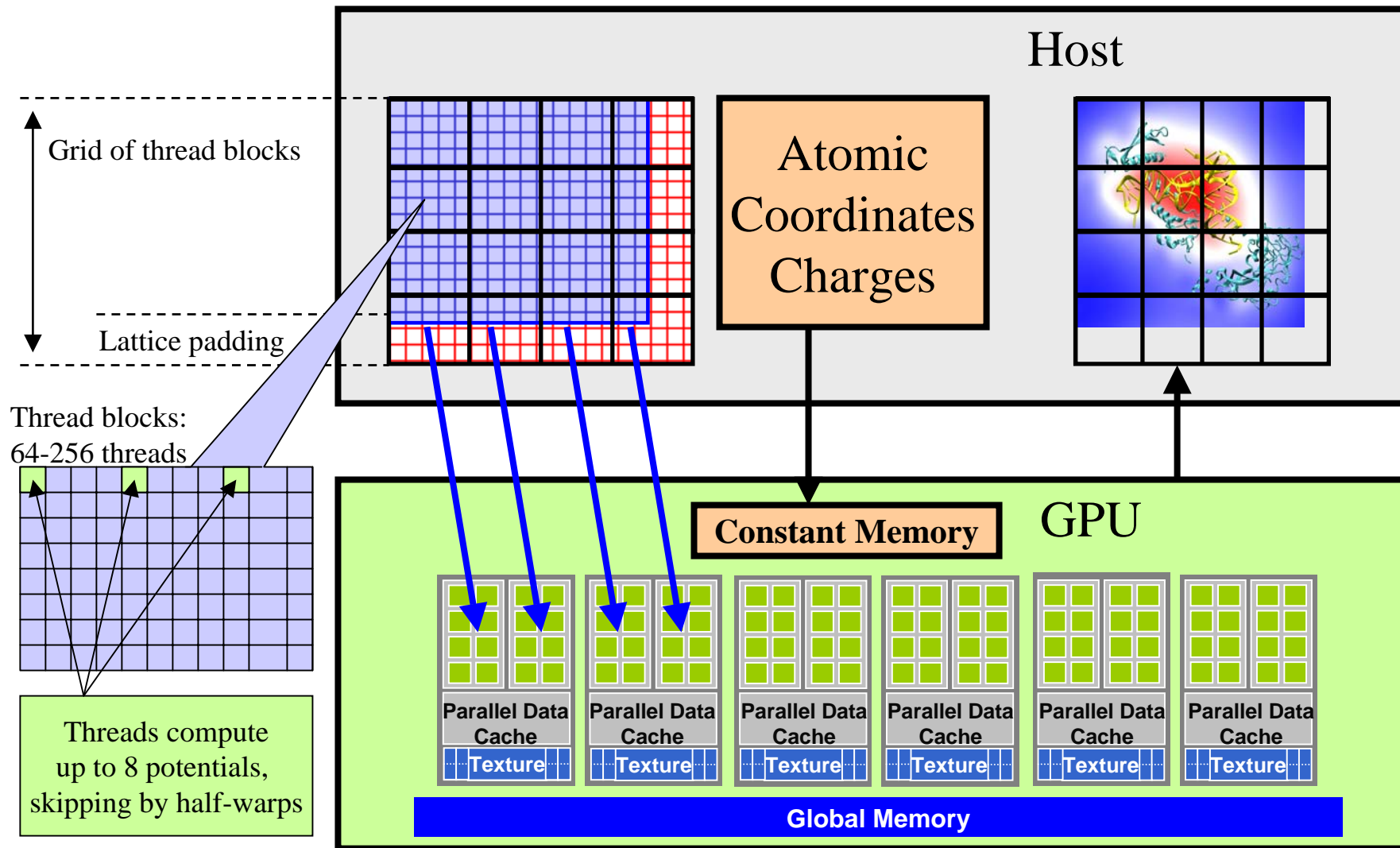
# DCS Observations for GPU Implementation

- 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

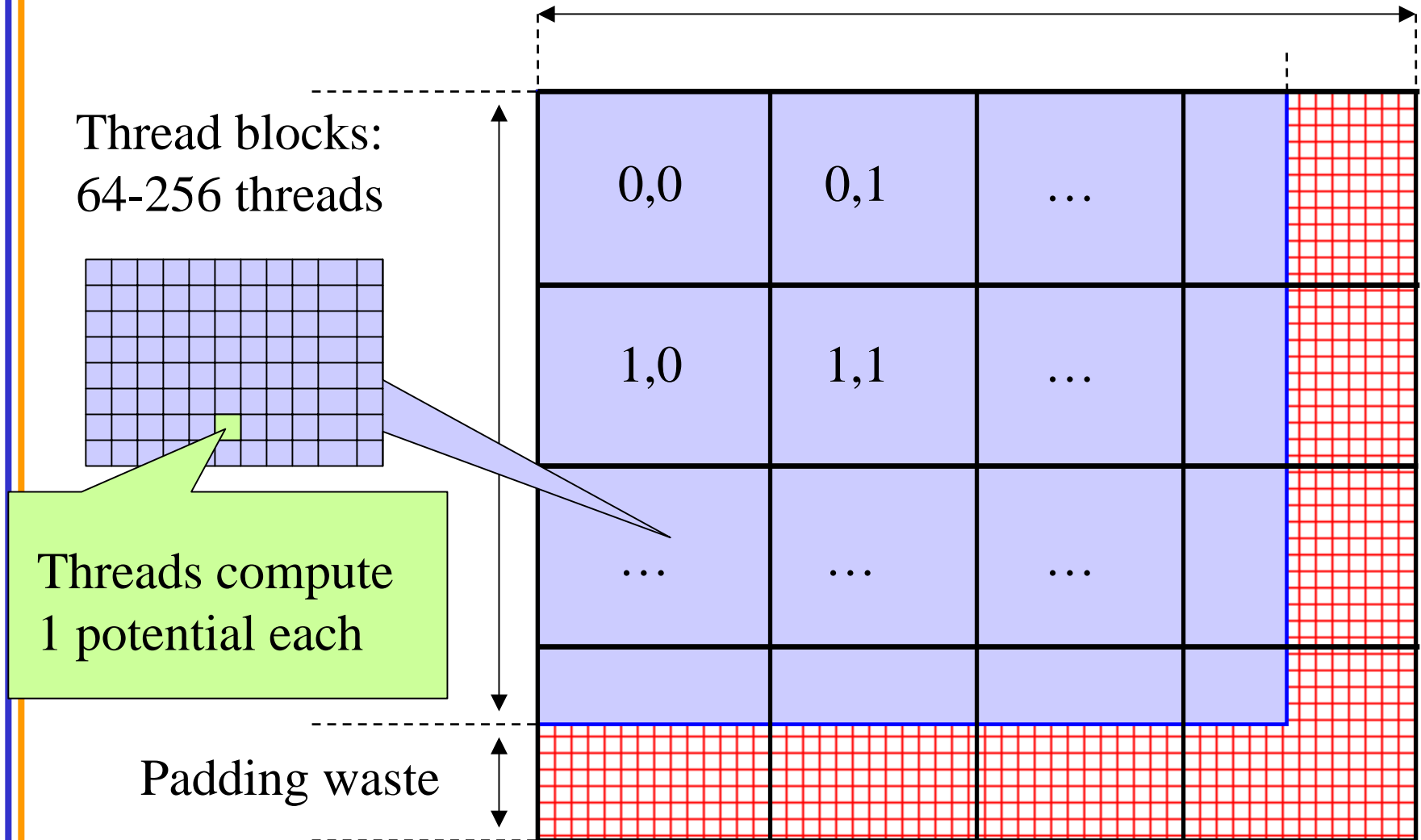




# DCS CUDA Block/Grid Decomposition

(non-unrolled)

Grid of thread blocks:



# 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 (~100ms) 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

- 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  $\sim 4000$  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 = gridspaceing * xindex;
float coory = gridspaceing * 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

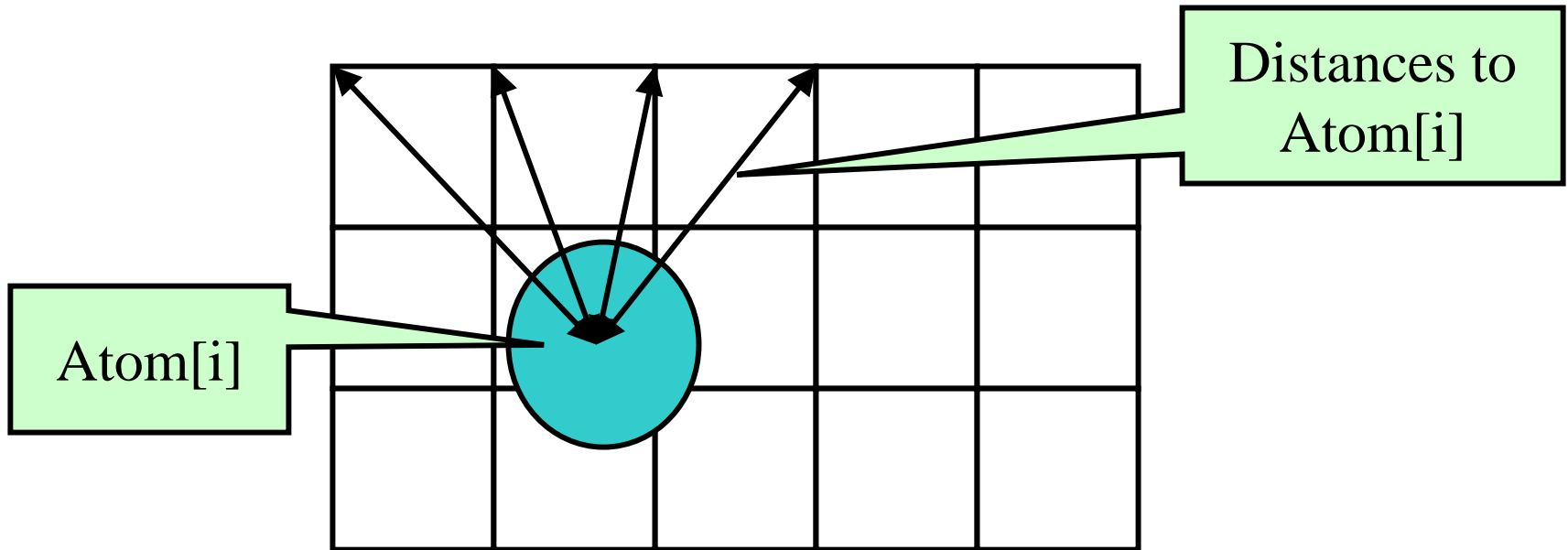
# DCS CUDA Algorithm: Unrolling Loops

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

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

$$\text{potential}[j+1] \ += \ \text{atom}[i].\text{charge} / r_{i(j+1)}$$

...





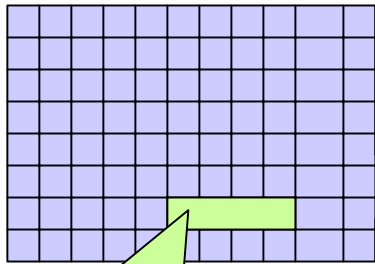
# DCS CUDA Block/Grid Decomposition

(unrolled)

Grid of thread blocks:

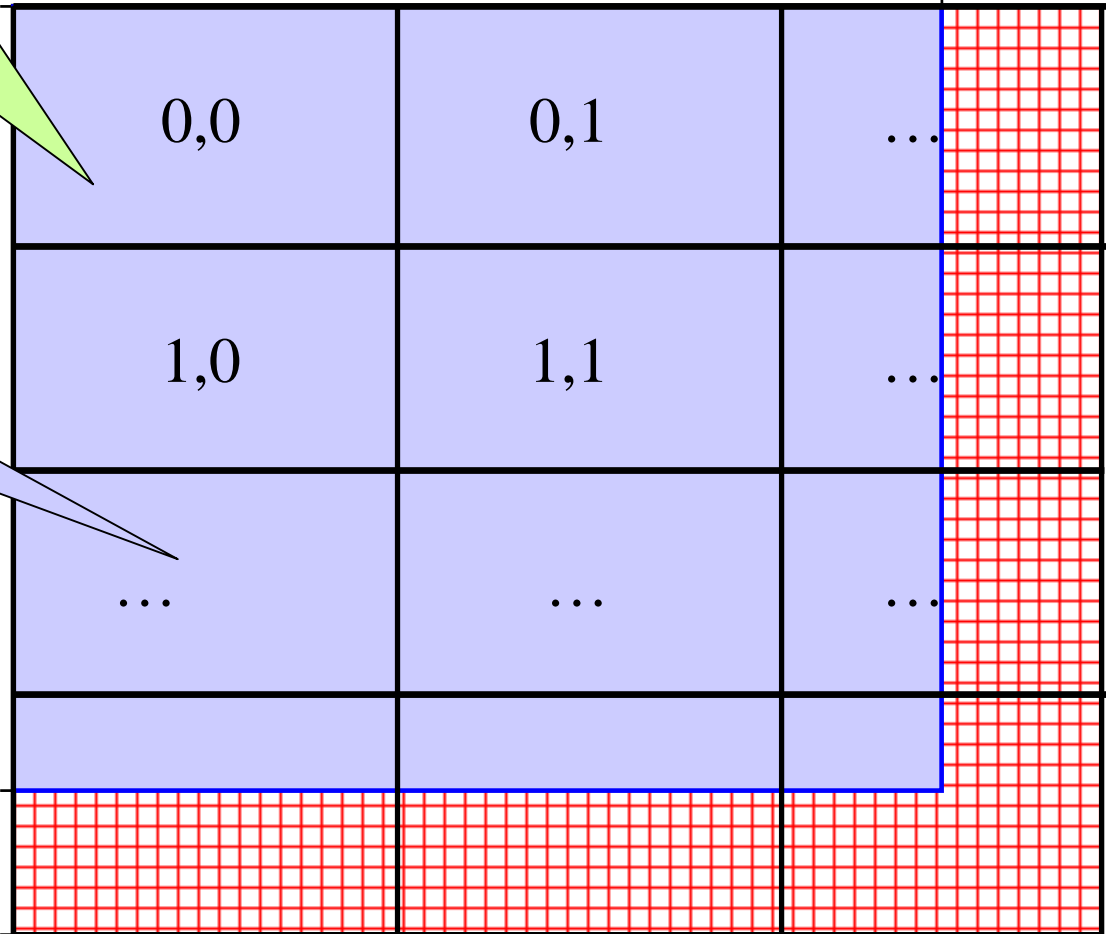
Unrolling increases computational tile size

Thread blocks:  
64-256 threads



Threads compute up to 8 potentials

Padding waste



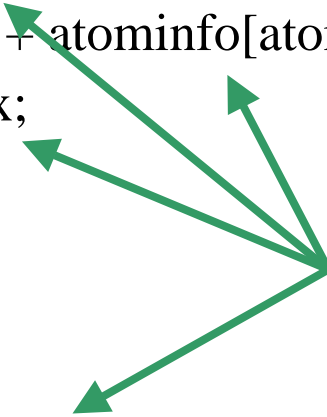
# DCS Version 2: Const+Precalc+Loop Unrolling

259 GFLOPS, 33.4 Billion Atom Evals/Sec

- 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 G80 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 of the summation, avoiding register consumption

# DCS Version 4: Inner Loop

```
...float coory = gridspacing * yindex;
float coorx = gridspacing * xindex;
float gridspacing_coalesce = gridspacing * BLOCKSIZEX;
int atomid;
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;
[...]
```

```
    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*BLOCKSIZEX] += energyvalx7;
```

Points spaced for  
memory coalescing

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

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

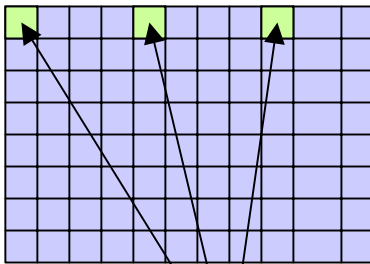
# DCS CUDA Block/Grid Decomposition

(unrolled, coalesced)

Grid of thread blocks:

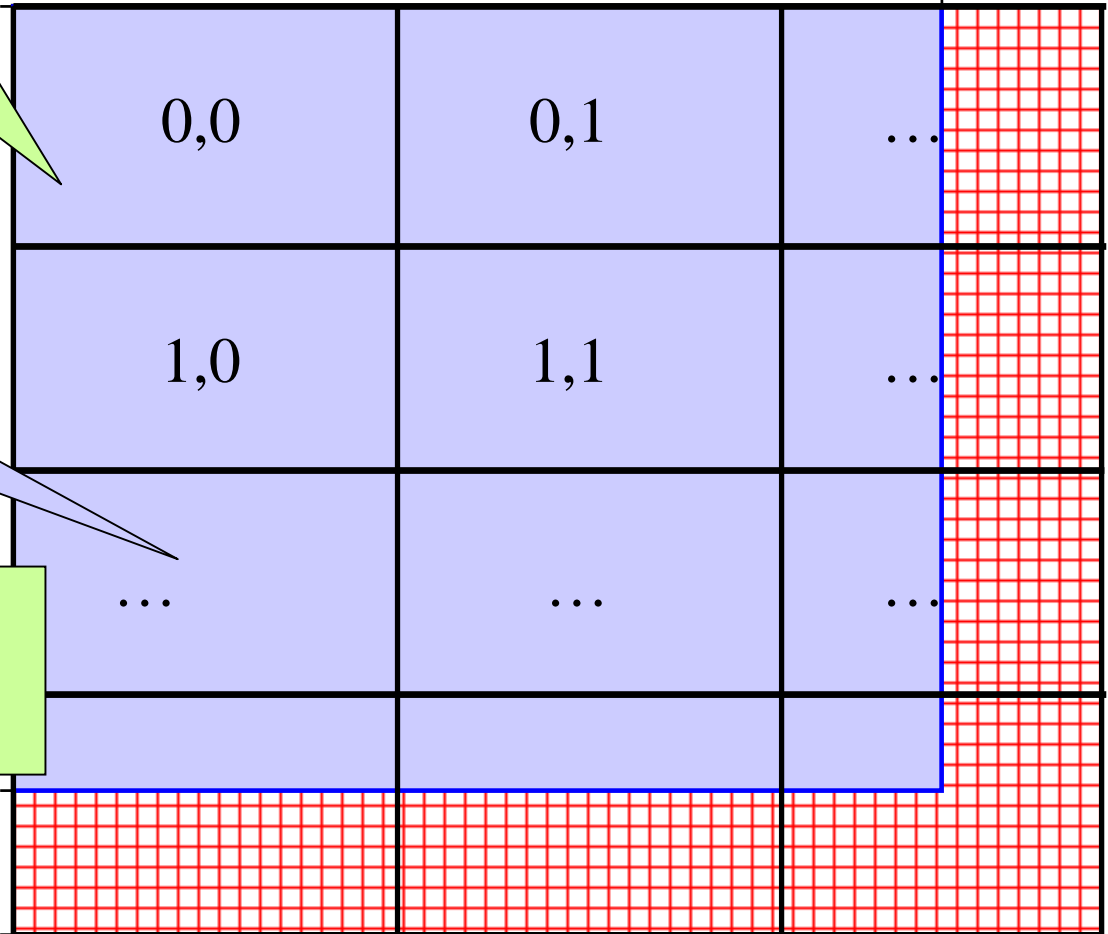
Unrolling increases computational tile size

Thread blocks:  
64-256 threads



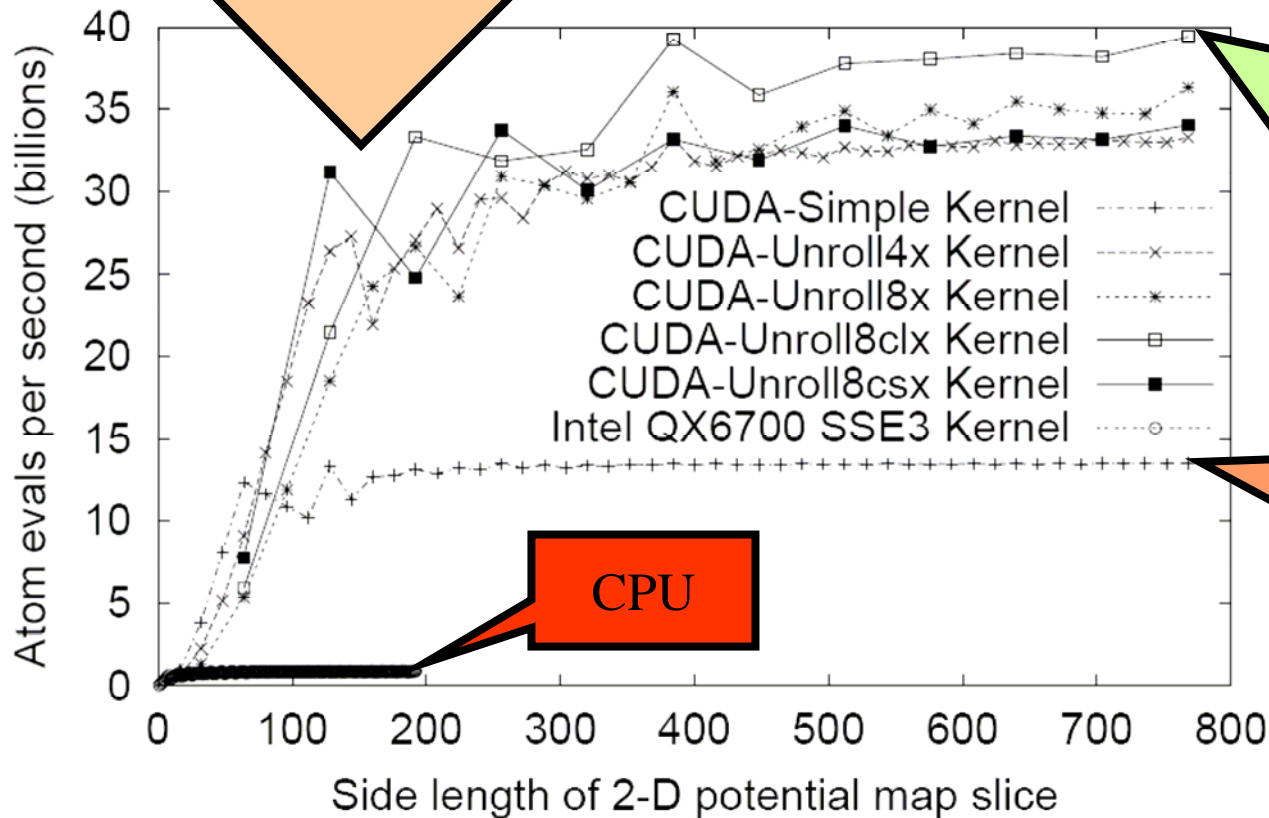
Threads compute up to 8 potentials, skipping by half-warps

Padding waste



# Direct Coulomb Summation Performance

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.



# 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
  - Handling error conditions within child threads is dependent on the thread library and, makes dealing with any CUDA errors somewhat tricky. Easiest way to deal with this is with a shared exception queue/stack for all of the worker threads.
- 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

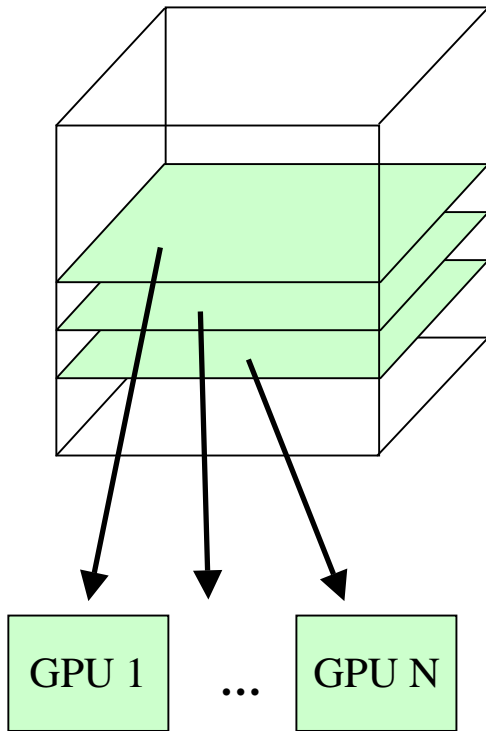
- Effective memory bandwidth scales with the number of GPUs utilized
- PCIe bus bandwidth not a bottleneck for this algorithm
- 117 billion evals/sec
- 863 GFLOPS
- 131x speedup vs. CPU core
- Power: 700 watts during benchmark



Quad-core Intel QX6700

Three NVIDIA GeForce 8800GTX

# Multi-GPU Direct Coulomb Summation



NCSA GPU Cluster

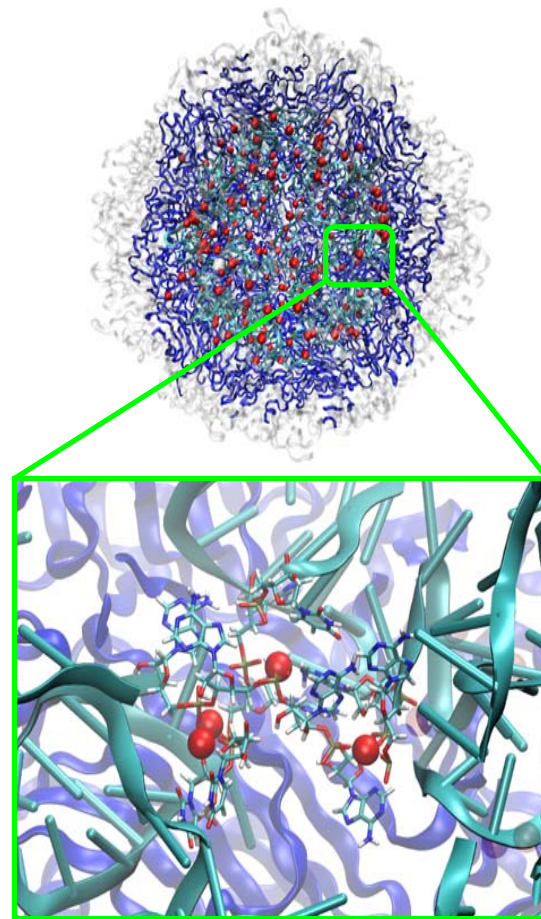
<http://www.ncsa.uiuc.edu/Projects/GPUcluster/>

	Evals/sec	TFLOPS	Speedup*
4-GPU (2 Quadroplex) Opteron node at NCSA	157 billion	1.16	<b>176</b>
4-GPU GTX 280 (GT200) In new NCSA Lincoln cluster	241 billion	1.78	<b>271</b>

\*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 Altix CPUs for the ribosome case



Satellite Tobacco Mosaic Virus (STMV)  
Ion Placement

# Multi-Level Summation Method for Coulomb Potential

## 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)



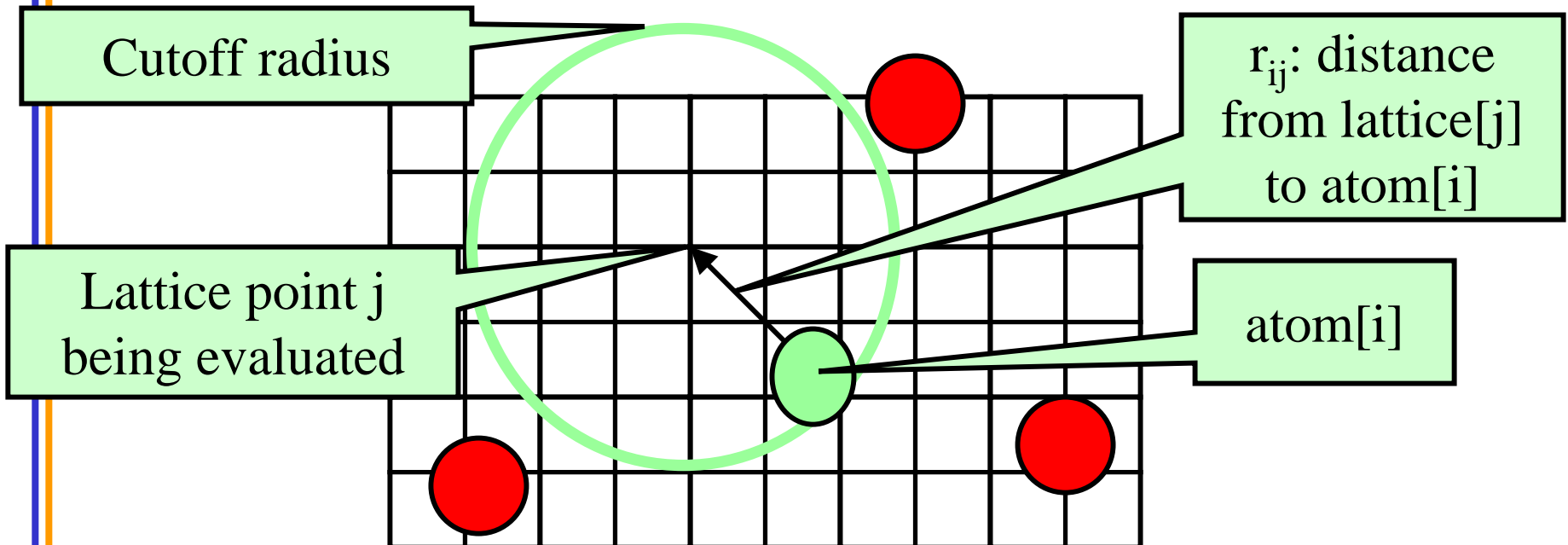
# Short-range Cutoff Summation

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

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

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

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



# Cutoff Summation on the GPU

Atoms are spatially hashed into fixed-size bins

CPU handles overflowed bins (GPU kernel can be very aggressive)

GPU thread block calculates corresponding region of potential map,

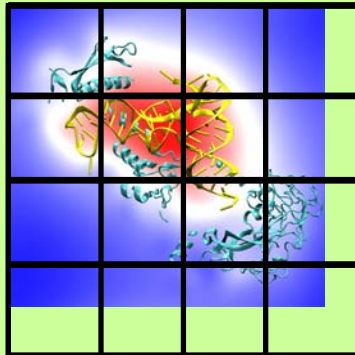
Bin/region neighbor checks costly; solved with universal table look-up

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

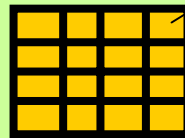
Shared memory

atom bin

Global memory



Potential map regions

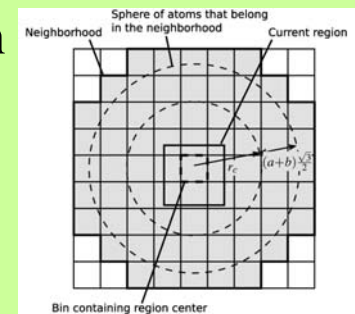


Bins of atoms

Constant memory

Offsets for bin neighborhood

Look-up table encodes "logic" of spatial geometry

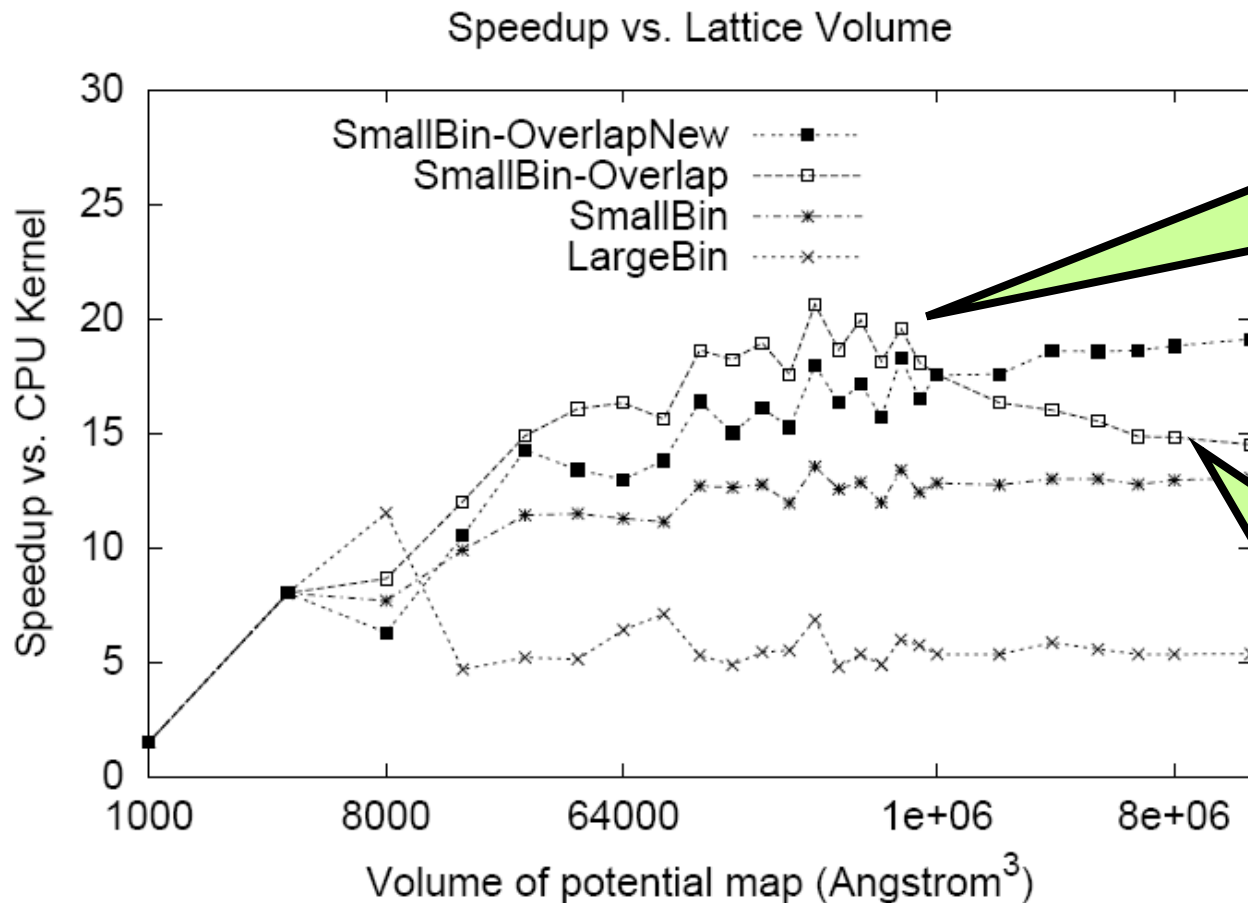




# Using the CPU to Improve GPU Performance

- 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
  - Use fixed size bin data structures, with “empty” slots skipped or producing zeroed out results
  - 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

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 (GT200 only) for potential accumulation resulted in only a ~10% performance penalty vs. pure single-precision arithmetic, while reducing the effects of floating point truncation

# Bonus Material!!

- This is a good point at which to address questions
- If time allows I will continue on covering the more advanced topics that follow, but it's important that we have addressed any questions about the fundamental material presented up to this point before continuing.

# Multilevel Summation

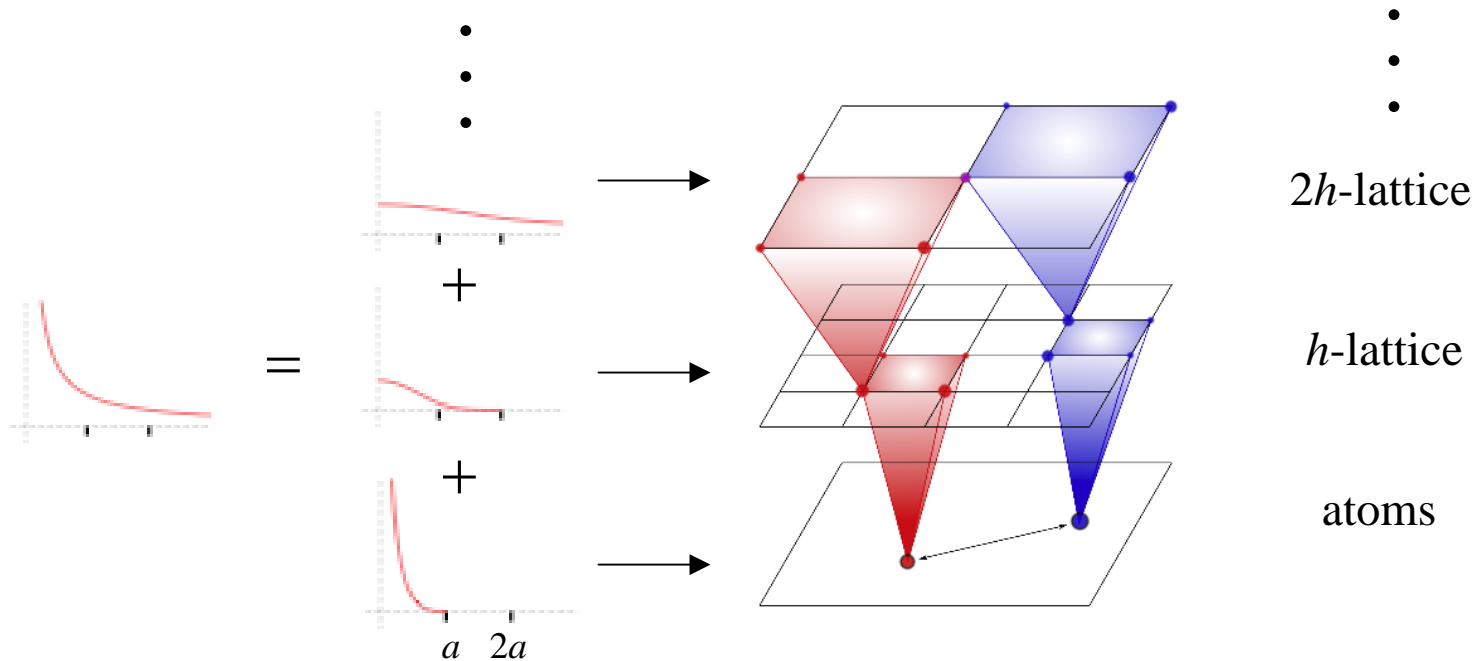
- Approximates full electrostatic potential
- Calculates sum of smoothed pairwise potentials interpolated from a hierarchy of lattices
- Advantages over PME and/or FMM:
  - Algorithm has linear time complexity
  - Permits non-periodic and periodic boundaries
  - Produces continuous forces for dynamics (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

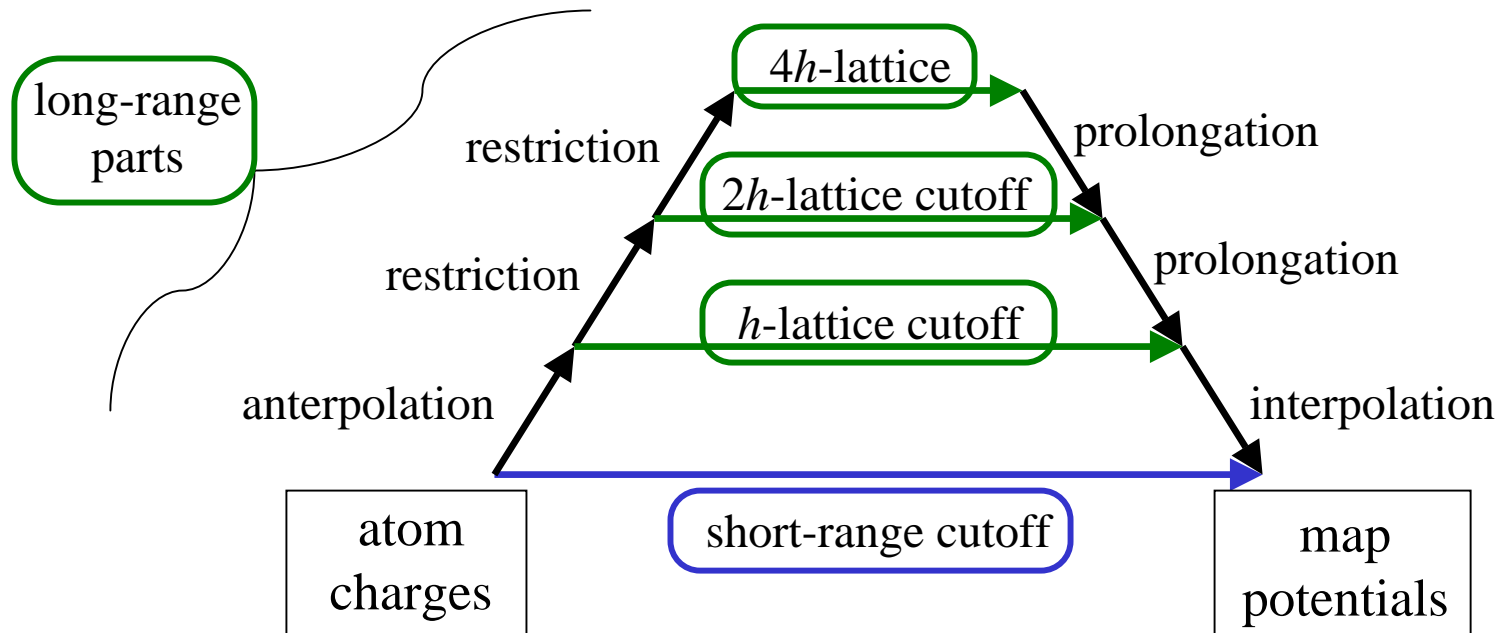
- Split the  $1/r$  potential into a short-range cutoff part plus smoothed parts that are successively more slowly varying. All but the top level potential are cut off.
- The smoothed potentials are interpolated from successively coarser lattices.
- The lattice spacing is doubled at each successive level.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



# Multilevel Summation Calculation

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

## Computational Steps

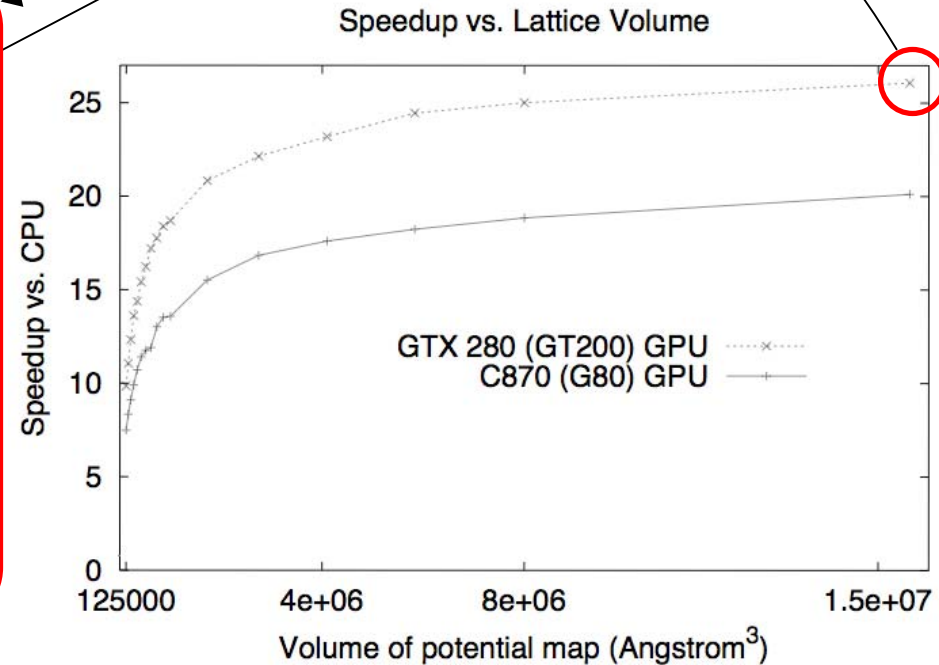


# Multilevel Summation on the GPU

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.

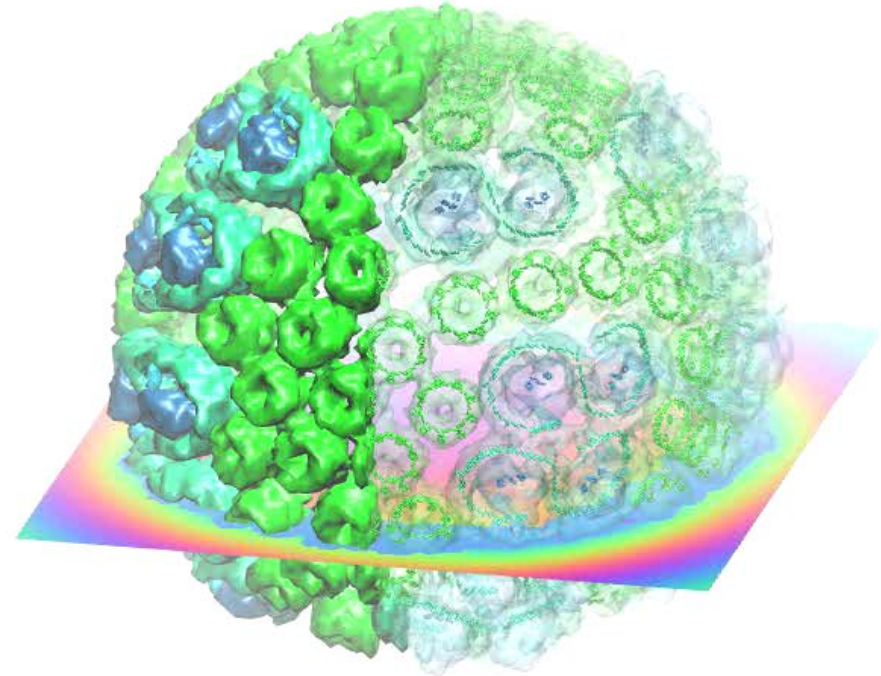
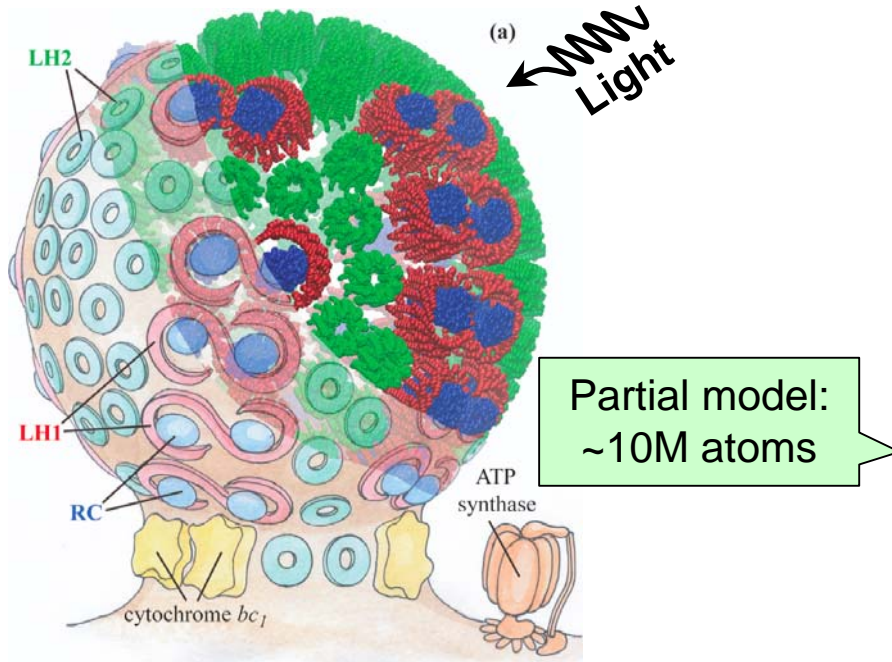
Computational steps	CPU (s)	w/ GPU (s)	Speedup
<b>Short-range cutoff</b>	480.07	14.87	32.3
Long-range anterpolation	0.18		
restriction	0.16		
<b>lattice cutoff</b>	49.47	1.36	36.4
prolongation	0.17		
interpolation	3.47		
Total	533.52	20.21	26.4





# Photobiology of Vision and Photosynthesis

## Investigations of the chromatophore, a photosynthetic organelle



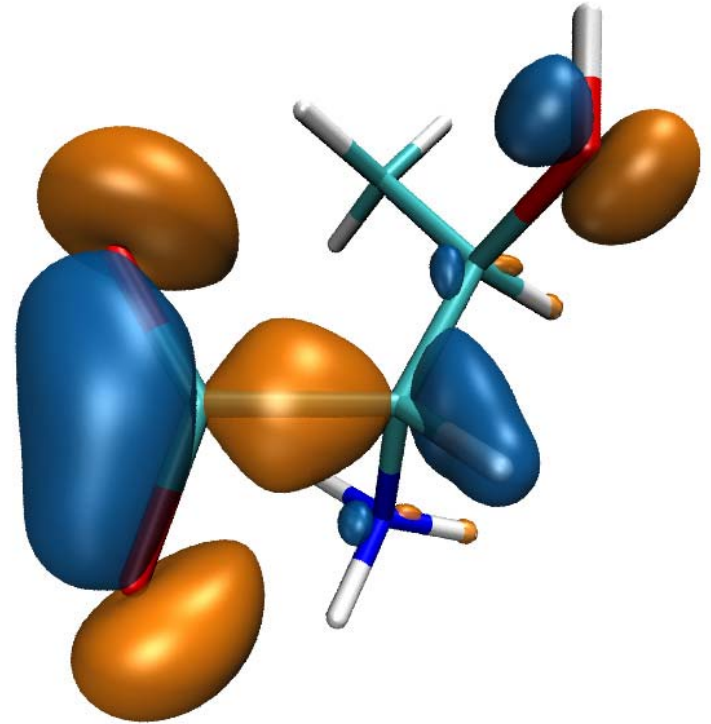
Electrostatics needed to build full structural model, place ions, study macroscopic properties

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

**Full chromatophore model will permit structural, chemical and kinetic investigations at a structural systems biology level**

# Molecular Orbitals

- Visualization of MOs aids in understanding the chemistry of molecular system
- MO spatial distribution is correlated with probability density for an electron(s)
- Algorithms for computing other interesting properties are similar, and can share code

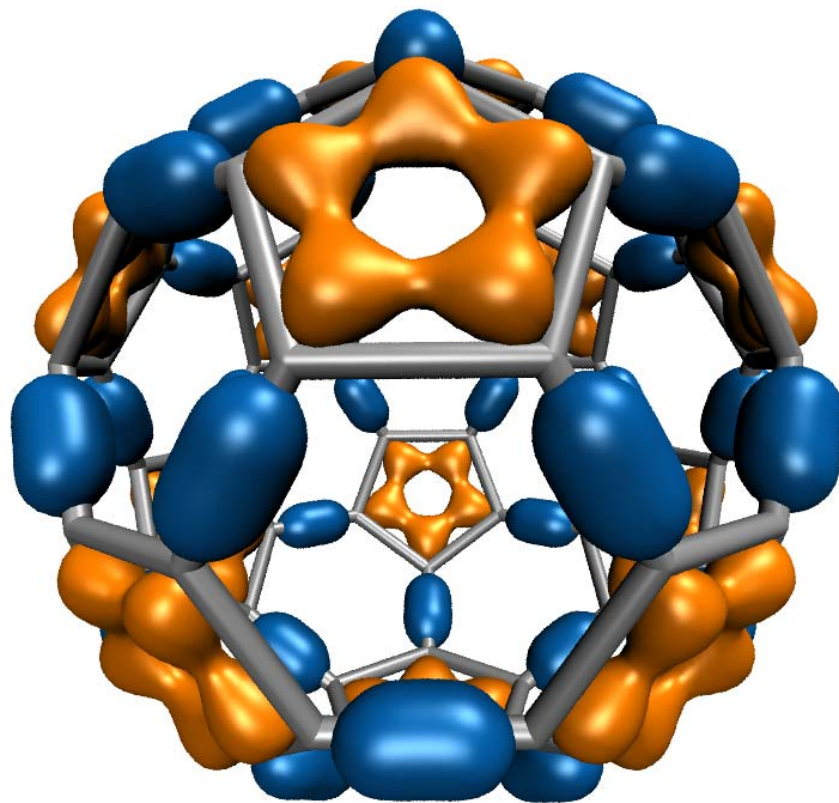


High Performance Computation and Interactive Display of Molecular Orbitals on GPUs and Multi-core CPUs.

J. Stone, J. Saam, D. Hardy, K. Vandivort, W. Hwu, K. Schulten,  
*2nd Workshop on General-Purpose Computation on Graphics  
Prrocessing Units (GPGPU-2), ACM International Conference  
Proceeding Series, volume 383, pp. 9-18, 2009.*

# Computing Molecular Orbitals

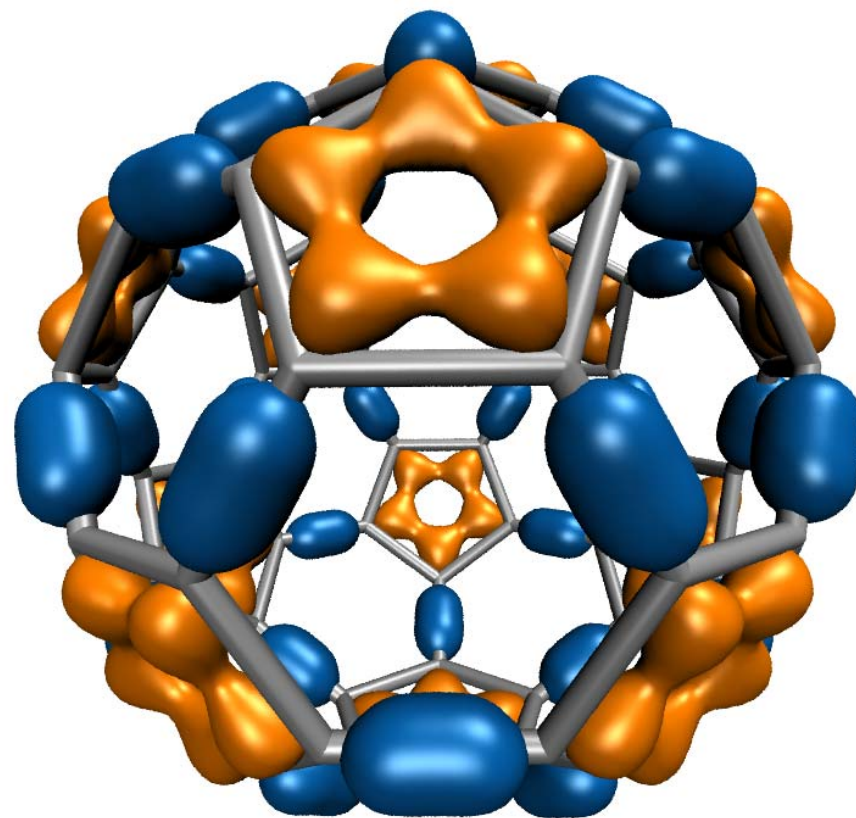
- Calculation of high resolution MO grids can require tens to hundreds of seconds in existing tools
- Existing tools cache MO grids as much as possible to avoid recomputation:
  - Doesn't eliminate the wait for initial calculation, hampers interactivity
  - Cached grids consume 100x-1000x more memory than MO coefficients



$C_{60}$

# Animating Molecular Orbitals

- Animation of (classical mechanics) molecular dynamics trajectories provides insight into simulation results
- To do the same for QM or QM/MM simulations one must compute MOs at **~10 FPS** or more
- **>100x** speedup (GPU) over existing tools now makes this possible!



$C_{60}$



# Molecular Orbital Computation and Display Process

## One-time initialization

**Initialize Pool of GPU Worker Threads**

Read QM simulation log file, trajectory

Preprocess MO coefficient data  
eliminate duplicates, sort by type, etc...

**For each trj frame, for each MO shown**

For current frame and MO index,  
retrieve MO wavefunction coefficients

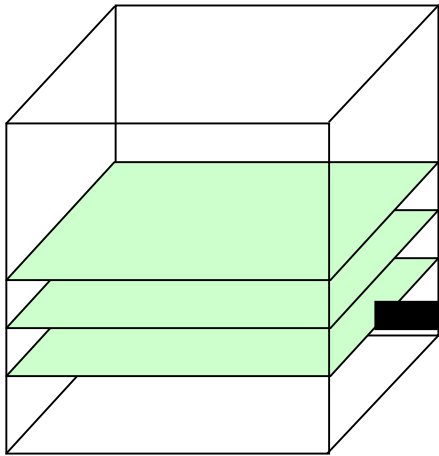
**Compute 3-D grid of MO wavefunction amplitudes**  
Most performance-demanding step, run on **GPU...**

Extract isosurface mesh from 3-D MO grid

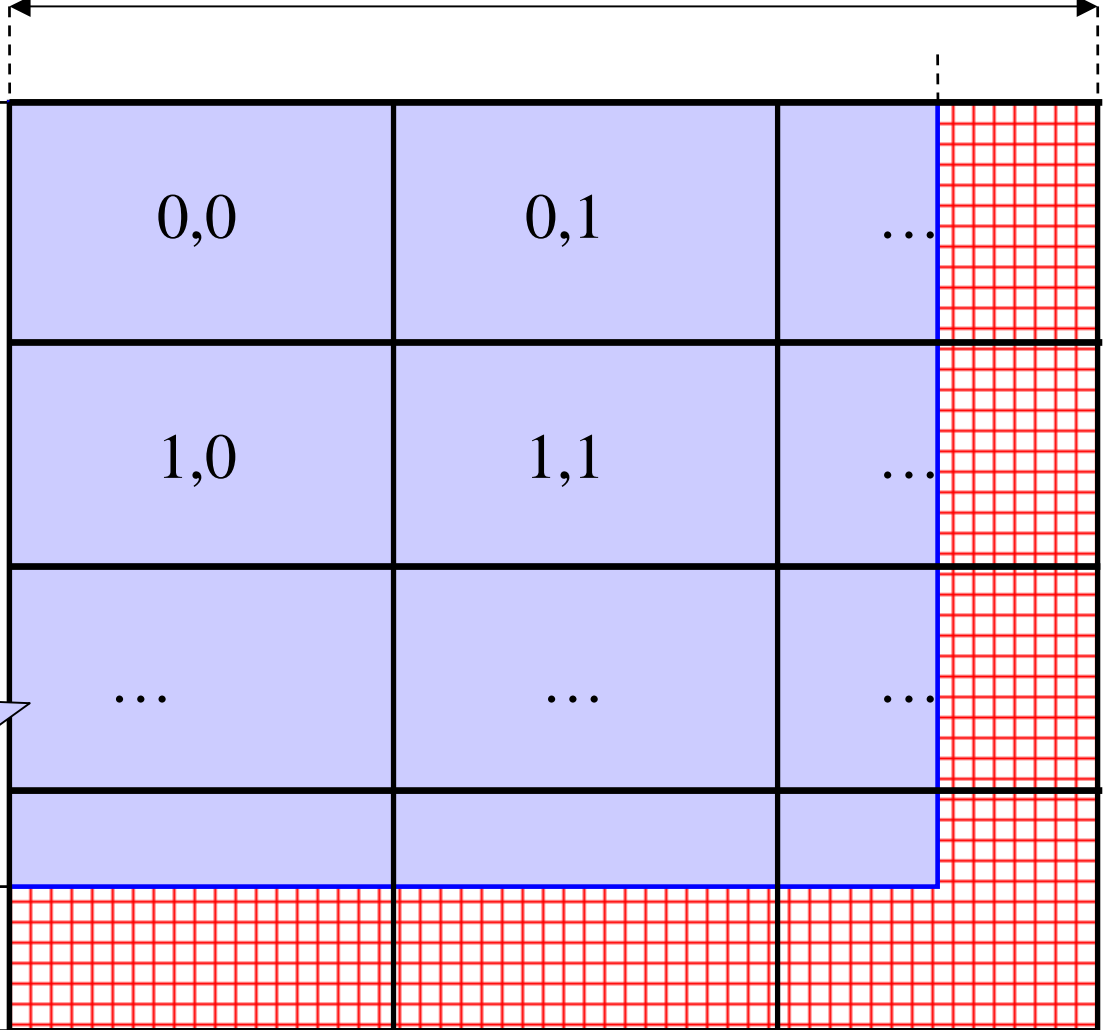
Apply user coloring/texturing  
and render the resulting surface

# CUDA Block/Grid Decomposition

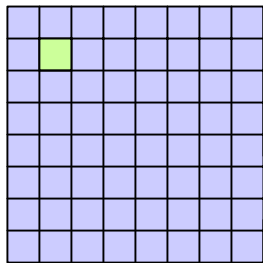
MO 3-D lattice decomposes into 2-D slices (CUDA grids)



Grid of thread blocks:



Small 8x8 thread blocks afford large per-thread register count, shared mem. Threads compute one MO lattice point each.



Padding optimizes glob. mem perf, guaranteeing coalescing



# MO Kernel for One Grid Point (Naive C)

Loop over atoms

...

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

```
  int prim_counter = atom_basis[at];
```

```
  calc_distances_to_atom(&atompos[at], &xdist, &ydist, &zdist, &dist2, &xdiv);
```

```
  for (contracted_gto=0.0f, shell=0; shell < num_shells_per_atom[at]; shell++) {
```

Loop over shells

```
    int shell_type = shell_symmetry[shell_counter];
```

```
    for (prim=0; prim < num_prim_per_shell[shell_counter]; prim++) {
```

```
      float exponent = basis_array[prim_counter];
```

```
      float contract_coeff = basis_array[prim_counter + 1];
```

```
      contracted_gto += contract_coeff * expf(-exponent*dist2);
```

```
      prim_counter += 2;
```

```
    }
```

Loop over primitives:  
largest component of  
runtime, due to expf()

```
    for (tmpshell=0.0f, j=0, zdp=1.0f; j<=shell_type; j++, zdp*=zdist) {
```

```
      int imax = shell_type - j;
```

```
      for (i=0, ydp=1.0f, xdp=pow(xdist, imax); i<=imax; i++, ydp*=ydist, xdp*=xdiv)
```

```
        tmpshell += wave_f[ifunc++] * xdp * ydp * zdp;
```

```
    }
```

Loop over angular  
momenta

(unrolled in real code)

```
    value += tmpshell * contracted_gto;
```

```
    shell_counter++;
```

```
  }
```

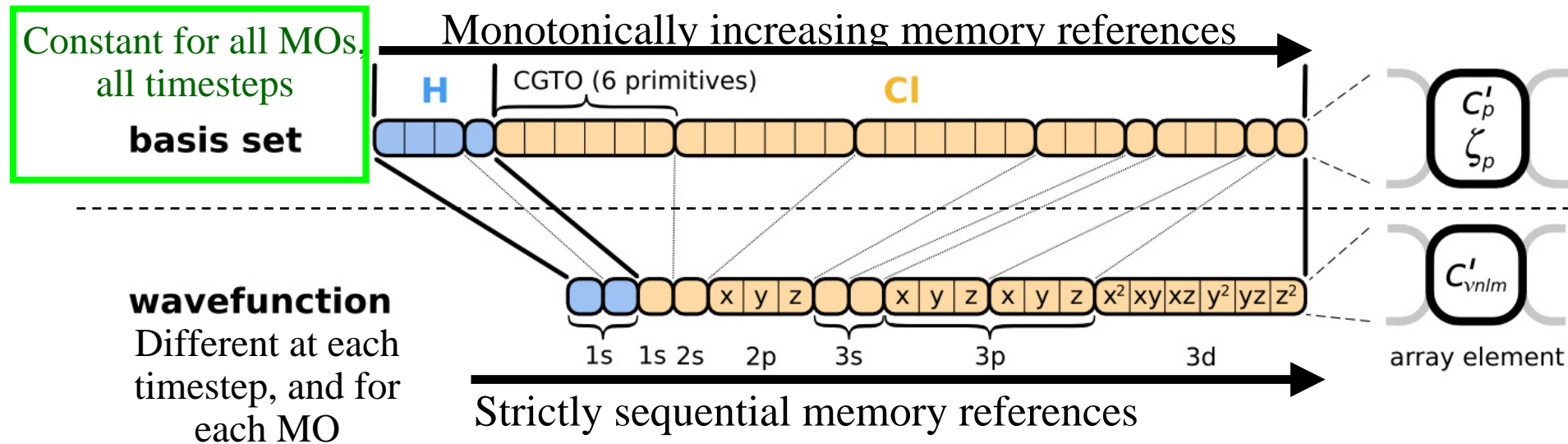
```
} .....
```

# Preprocessing of Atoms, Basis Set, and Wavefunction Coefficients

- Must make effective use of high bandwidth, low-latency GPU on-chip memory, or CPU cache:
  - Overall storage requirement reduced by eliminating duplicate basis set coefficients
  - Sorting atoms by element type allows re-use of basis set coefficients for subsequent atoms of identical type
- Padding, alignment of arrays guarantees coalesced GPU global memory accesses, CPU SSE loads



# GPU Traversal of Atom Type, Basis Set, Shell Type, and Wavefunction Coefficients

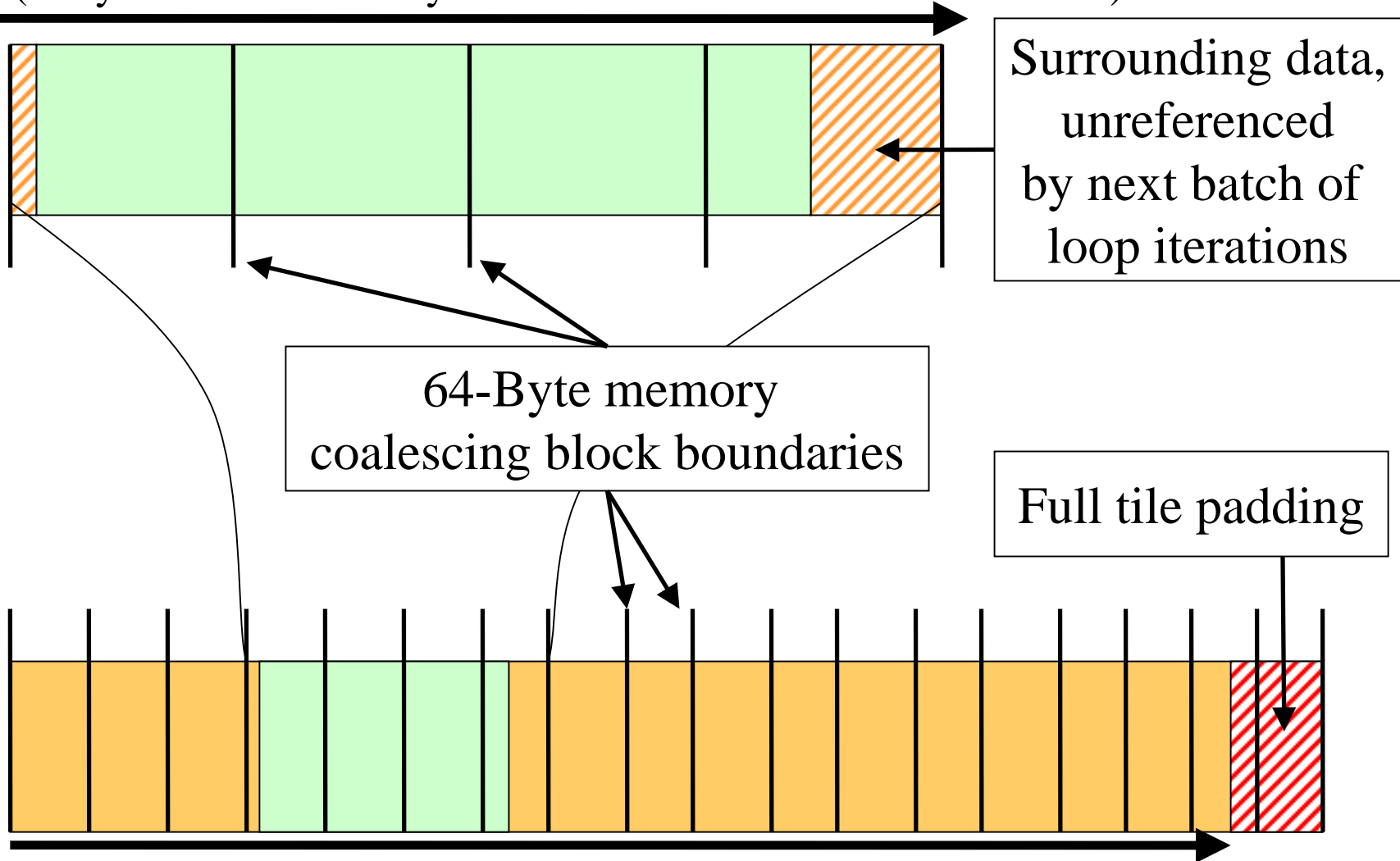


- Loop iterations always access same or consecutive array elements for all threads in a thread block:
  - Yields good constant memory cache performance
  - Increases shared memory tile reuse

# Use of GPU On-chip Memory

- If total data less than 64 kB, use only const mem:
  - Broadcasts data to all threads, no global memory accesses!
- For large data, shared memory used as a program-managed cache, coefficients loaded on-demand:
  - Tile data in shared mem is broadcast to 64 threads in a block
  - Nested loops traverse multiple coefficient arrays of varying length, complicates things significantly...
  - Key to performance is to locate tile loading checks outside of the two performance-critical inner loops
  - Tiles sized large enough to service entire inner loop runs
  - Only 27% slower than hardware caching provided by constant memory (GT200)

Array tile loaded in GPU shared memory. Tile size is a power-of-two, multiple of coalescing size, and allows simple indexing in inner loops (array indices are merely offset for reference within loaded tile).



Coefficient array in GPU global memory

# VMD MO Performance Results for $C_{60}$

Sun Ultra 24: Intel Q6600, NVIDIA GTX 280

Kernel	Cores/GPUs	Runtime (s)	Speedup
CPU ICC-SSE	1	46.58	1.00
CPU ICC-SSE	4	11.74	3.97
CPU ICC-SSE-approx**	4	3.76	12.4
CUDA-tiled-shared	1	0.46	100.
CUDA-const-cache	1	0.37	126.
<b>CUDA-const-cache-JIT*</b>	<b>1</b>	<b>0.27</b>	<b>173.</b> <b>(JIT 40% faster)</b>

$C_{60}$  basis set 6-31Gd. We used an unusually-high resolution MO grid for accurate timings. A more typical calculation has  $1/8^{\text{th}}$  the grid points.

\* Runtime-generated JIT kernel compiled using batch mode CUDA tools

\*\*Reduced-accuracy approximation of  $\exp()$ ,  
cannot be used for zero-valued MO isosurfaces

# Performance Evaluation:

Molekel, MacMolPlt, and VMD

Sun Ultra 24: Intel Q6600, NVIDIA GTX 280

	<b>C<sub>60</sub>-A</b>	<b>C<sub>60</sub>-B</b>	<b>Thr-A</b>	<b>Thr-B</b>	<b>Kr-A</b>	<b>Kr-B</b>
Atoms	60	60	17	17	1	1
Basis funcs (unique)	<b>300 (5)</b>	<b>900 (15)</b>	<b>49 (16)</b>	<b>170 (59)</b>	<b>19 (19)</b>	<b>84 (84)</b>

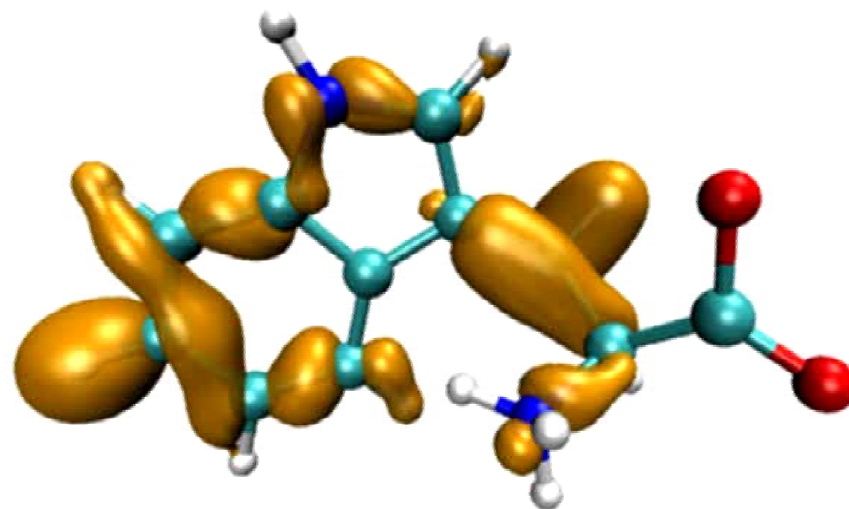
Kernel	Cores GPUs	Speedup vs. Molekel on 1 CPU core					
Molekel	1*	1.0	1.0	1.0	1.0	1.0	1.0
MacMolPlt	4	2.4	2.6	2.1	2.4	4.3	4.5
VMD GCC-cephes	4	3.2	4.0	3.0	3.5	4.3	6.5
VMD ICC-SSE-cephes	4	16.8	17.2	13.9	12.6	17.3	21.5
VMD ICC-SSE-approx**	4	59.3	53.4	50.4	49.2	54.8	69.8
VMD CUDA-const-cache	1	552.3	533.5	355.9	421.3	193.1	571.6

# VMD Orbital Dynamics Proof of Concept

One GPU can compute and animate this movie on-the-fly!

CUDA const-cache kernel,  
Sun Ultra 24, GeForce GTX 285

GPU MO grid calc.	<b>0.016 s</b>
CPU surface gen, volume gradient, and GPU rendering	<b>0.033 s</b>
<b>Total runtime</b>	<b>0.049 s</b>
<b>Frame rate</b>	<b>20 FPS</b>

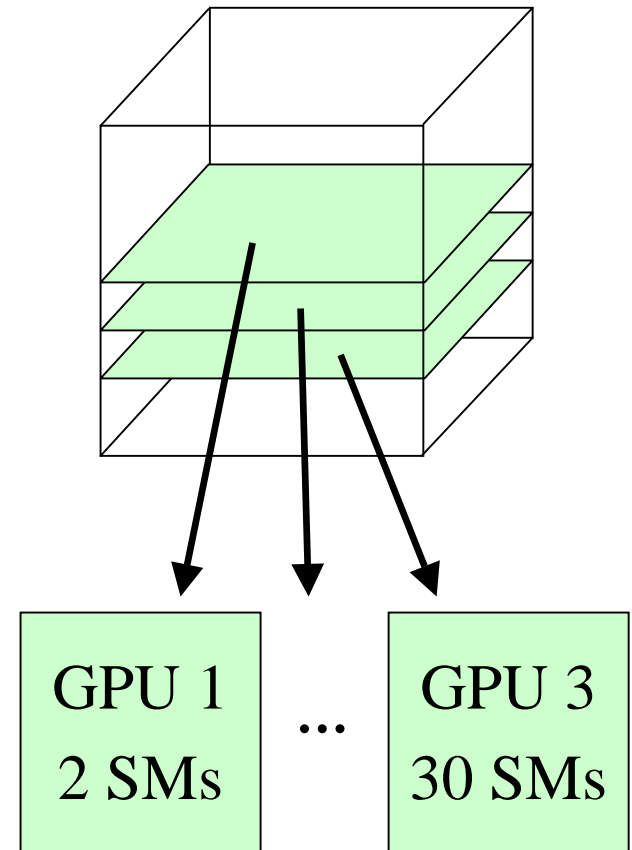


threonine

With GPU speedups over **100x**, previously insignificant CPU surface gen, gradient calc, and rendering are now **66%** of runtime. Need GPU-accelerated surface gen next...

# Multi-GPU Load Balance

- Many early CUDA codes assumed all GPUs were identical
- All new NVIDIA cards support CUDA, so a typical machine may have a diversity of GPUs of varying capability
- Static decomposition works poorly for non-uniform workload, or diverse GPUs, e.g. 2 SM, 16 SM, 30 SM



# VMD Multi-GPU Molecular Orbital Performance Results for C<sub>60</sub>

Kernel	Cores/GPUs	Runtime (s)	Speedup	Parallel Efficiency
CPU-ICC-SSE	1	46.580	1.00	100%
CPU-ICC-SSE	4	11.740	3.97	99%
CUDA-const-cache	1	0.417	112	100%
CUDA-const-cache	2	0.220	212	94%
CUDA-const-cache	3	0.151	308	92%
CUDA-const-cache	4	0.113	412	92%

Intel Q6600 CPU, 4x Tesla C1060 GPUs,

Uses persistent thread pool to avoid GPU init overhead,  
dynamic scheduler distributes work to GPUs



# MO Kernel Structure, Opportunity for JIT...

Data-driven, but representative loop trip counts in (...)

Loop over atoms (1 to ~200) {

Loop over electron shells for this atom type (1 to ~6) {

Loop over primitive functions for this shell type (1 to ~6) {

Unpredictable (at compile-time, since data-driven ) but  
small loop trip counts result in significant loop overhead.

**Dynamic kernel generation and JIT compilation can  
unroll entirely, resulting in 40% speed boost**

Loop over angular momenta for this shell type (1 to ~15) {}

}

}

# Molecular Orbital Computation and Display Process

## Dynamic Kernel Generation, Just-In-Time (JIT) COmpilation

**One-time  
initialization**

**Initialize Pool of GPU  
Worker Threads**

Read QM simulation log file, trajectory

Preprocess MO coefficient data  
eliminate duplicates, sort by type, etc...

**Generate/compile basis set-specific CUDA kernel**

For current frame and MO index,  
retrieve MO wavefunction coefficients

**Compute 3-D grid of MO wavefunction amplitudes  
using basis set-specific CUDA kernel**

Extract isosurface mesh from 3-D MO grid

Render the resulting surface

**For each trj frame, for  
each MO shown**

```

.....
// loop over the shells belonging to this atom (or basis function)
for (shell=0; shell < maxshell; shell++) {
    float contracted_gto = 0.0f;

    // Loop over the Gaussian primitives of this contracted
    // basis function to build the atomic orbital
    int maxprim = const_num_prim_per_shell[shell_counter];
    int shell_type = const_shell_symmetry[shell_counter];
    for (prim=0; prim < maxprim; prim++) {
        float exponent = const_basis_array[prim_counter];
        float contract_coeff = const_basis_array[prim_counter + 1];
        contracted_gto += contract_coeff * exp2f(-exponent*dist2);
        prim_counter += 2;
    }

    /* multiply with the appropriate wavefunction coefficient */
    float tmpshell=0;
    switch (shell_type) {
        case S_SHELL:
            value += const_wave_f[ifunc++] * contracted_gto;
            break;

[.....]
        case D_SHELL:
            tmpshell += const_wave_f[ifunc++] * xdist2;
            tmpshell += const_wave_f[ifunc++] * ydist2;
            tmpshell += const_wave_f[ifunc++] * zdist2;
            tmpshell += const_wave_f[ifunc++] * xdist * ydist;
            tmpshell += const_wave_f[ifunc++] * xdist * zdist;
            tmpshell += const_wave_f[ifunc++] * ydist * zdist;
            value += tmpshell * contracted_gto;
            break;

```

## General loop-based CUDA kernel



## Dynamically-generated CUDA kernel (JIT)



```

.....
contracted_gto = 1.832937 * expf(-7.868272*dist2);
contracted_gto += 1.405380 * expf(-1.881289*dist2);
contracted_gto += 0.701383 * expf(-0.544249*dist2);
// P_SHELL
tmpshell = const_wave_f[ifunc++] * xdist;
tmpshell += const_wave_f[ifunc++] * ydist;
tmpshell += const_wave_f[ifunc++] * zdist;
value += tmpshell * contracted_gto;

contracted_gto = 0.187618 * expf(-0.168714*dist2);
// S_SHELL
value += const_wave_f[ifunc++] * contracted_gto;

contracted_gto = 0.217969 * expf(-0.168714*dist2);
// P_SHELL
tmpshell = const_wave_f[ifunc++] * xdist;
tmpshell += const_wave_f[ifunc++] * ydist;
tmpshell += const_wave_f[ifunc++] * zdist;
value += tmpshell * contracted_gto;

contracted_gto = 3.858403 * expf(-0.800000*dist2);
// D_SHELL
tmpshell = const_wave_f[ifunc++] * xdist2;
tmpshell += const_wave_f[ifunc++] * ydist2;
tmpshell += const_wave_f[ifunc++] * zdist2;
tmpshell += const_wave_f[ifunc++] * xdist * ydist;
tmpshell += const_wave_f[ifunc++] * xdist * zdist;
tmpshell += const_wave_f[ifunc++] * ydist * zdist;
value += tmpshell * contracted_gto;

```

# Lessons Learned

- GPU algorithms need fine-grained parallelism and sufficient work to fully utilize the hardware
- Much of per-thread GPU algorithm optimization revolves around efficient use of multiple memory systems and latency hiding
- Concurrency can often be traded for per-thread performance, in combination with increased use of registers or shared memory
- Fine-grained GPU work decompositions often compose well with the comparatively coarse-grained decompositions used for multicore or distributed memory programming

# 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
- Targeted use of double-precision floating point arithmetic, or compensated summation can reduce the effects of floating point truncation at low cost to performance

# Summary

- GPUs are not a magic bullet, but they can perform amazingly well when used effectively
- There are many good strategies for extracting high performance from individual subsystems on the GPU
- It is wise to begin with a well designed application and a thorough understanding of its performance characteristics on the CPU before beginning work on the GPU
- By making effective use of multiple GPU subsystems at once, tremendous performance levels can potentially be attained

# Acknowledgements

- Additional Information and References:
  - <http://www.ks.uiuc.edu/Research/gpu/>
  - <http://www.ks.uiuc.edu/Research/vmd/>
- Questions, source code requests:
  - John Stone: [johns@ks.uiuc.edu](mailto:johns@ks.uiuc.edu)
- Acknowledgements:
  - D. Hardy, J. Saam, J. Phillips, P. Freddolino, L. Trabuco, J. Cohen, K. Schulten (UIUC TCB Group)
  - Prof. Wen-mei Hwu, Christopher Rodrigues (UIUC IMPACT Group)
  - David Kirk and the CUDA team at NVIDIA
  - NIH support: P41-RR05969
  - UIUC NVIDIA Center of Excellence

# Publications

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

- Long time-scale simulations of in vivo diffusion using GPU hardware. E. Roberts, J. Stone, L. Sepulveda, W. Hwu, Z. Luthey-Schulten. In *Eighth IEEE International Workshop on High Performance Computational Biology*, 2009. In press.
- 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.
- Multilevel summation of electrostatic potentials using graphics processing units. D. Hardy, J. Stone, K. Schulten. *J. Parallel Computing*, 35:164-177, 2009.
- 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.