# CUDA Application Examples

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/**

GPGPU 2015: Advanced Methods for Computing with CUDA,

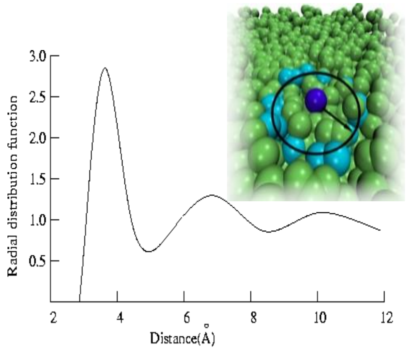
University of Cape Town, April 2015

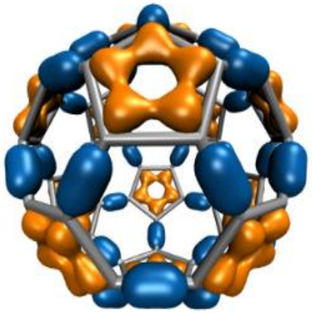
# What Speedups Can GPUs Achieve?

- Single-GPU speedups of **2.5x** to **8x** vs. one CPU socket are common
- Best speedups can reach **25x** or more, attained on codes dominated by floating point arithmetic, especially native GPU machine instructions, e.g. expf(), rsqrtf(), …
- **Amdahl's Law** can prevent legacy codes from achieving peak speedups with shallow GPU acceleration efforts

# CUDA GPU-Accelerated Trajectory Analysis and Visualization in VMD

| **VMD GPU-Accelerated Feature or Kernel** | **Typical speedup vs. multi-core CPU (e.g. 4-core CPU)** |
|---|---|
| **Molecular orbital display** | **30x** |
| **Radial distribution function** | **23x** |
| **Molecular surface display** | **15x** |
| **Electrostatic field calculation** | **11x** |
| **Ray tracing w/ shadows,<br>AO lighting** | **7x** |
| Ion placement | 6x |
| MDFF density map synthesis | 6x |
| Implicit ligand sampling | 6x |
| Root mean squared fluctuation | 6x |
| Radius of gyration | 5x |
| Close contact determination | 5x |
| Dipole moment calculation | 4x |

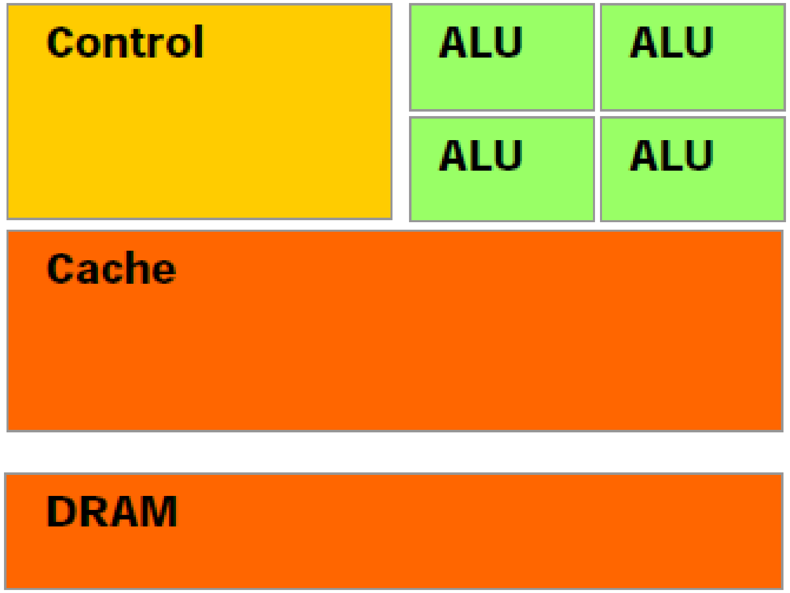


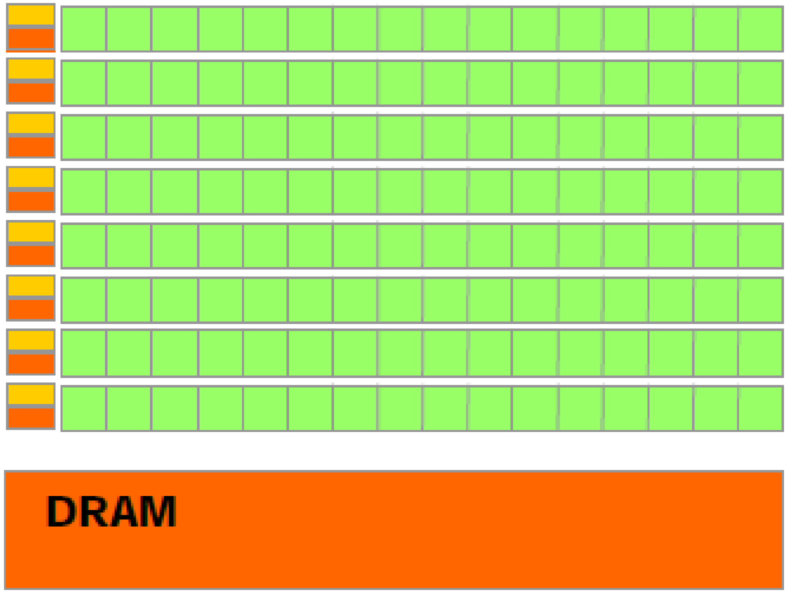


# Comparison of CPU and GPU Hardware Architecture

**CPU**: Cache heavy, focused on individual thread performance

**GPU**: ALU heavy, massively parallel, throughput oriented





# Multiple Debye-Hückel Electrostatics

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

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

# MDH Kernel (CUDA)

```
extern shared f loat 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 lg z = gz [ igrid ] ;  float v = 0.0 f ;
for ( int jatom = 0 ; jatom < natoms ; jatom+=lsize ) {
   syncthreads ( ) ;
   i f ( ( jatom + l i d ) < 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 ( ) ;
   i f ( ( 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++) {
         f loat dx = lgx − smem[ i                    ] ;
         f loat dy = lgy − smem[ i +       lsize ] ;
         f loat dz = lgz − smem[ i + 2 * lsize ] ;
         f loat 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

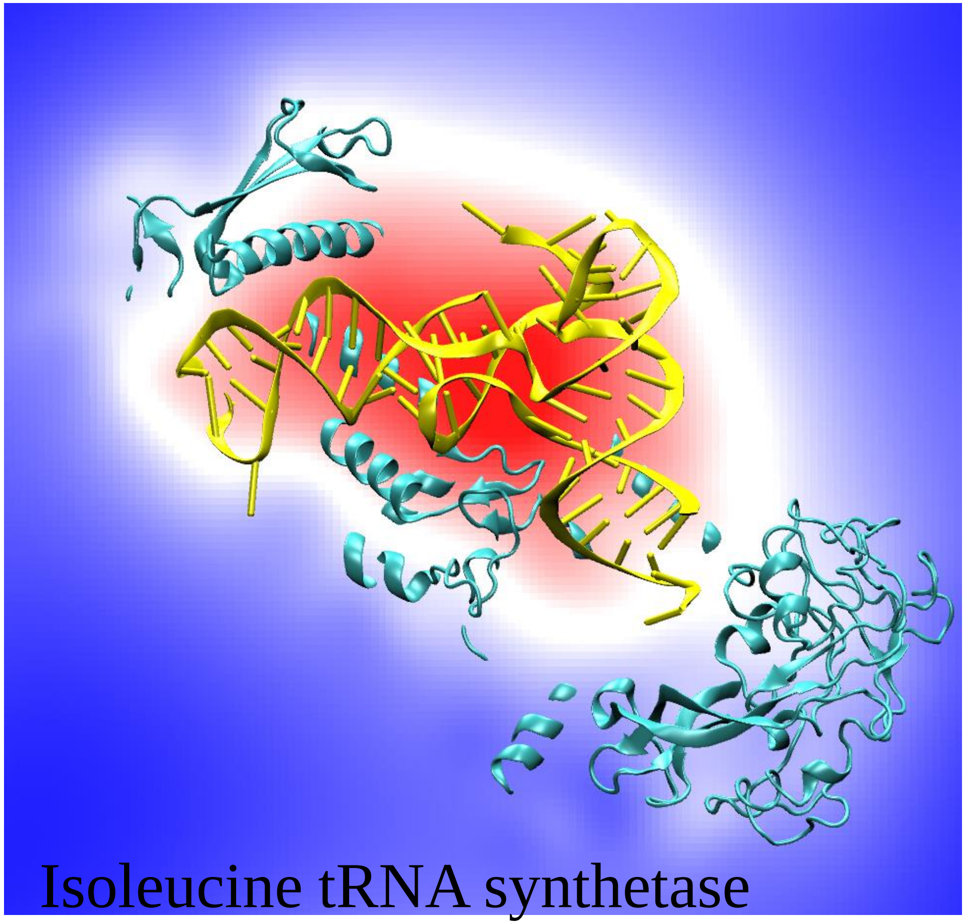
Loop over all all atoms in shared memory accumulating potential contributions into grid points

# 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

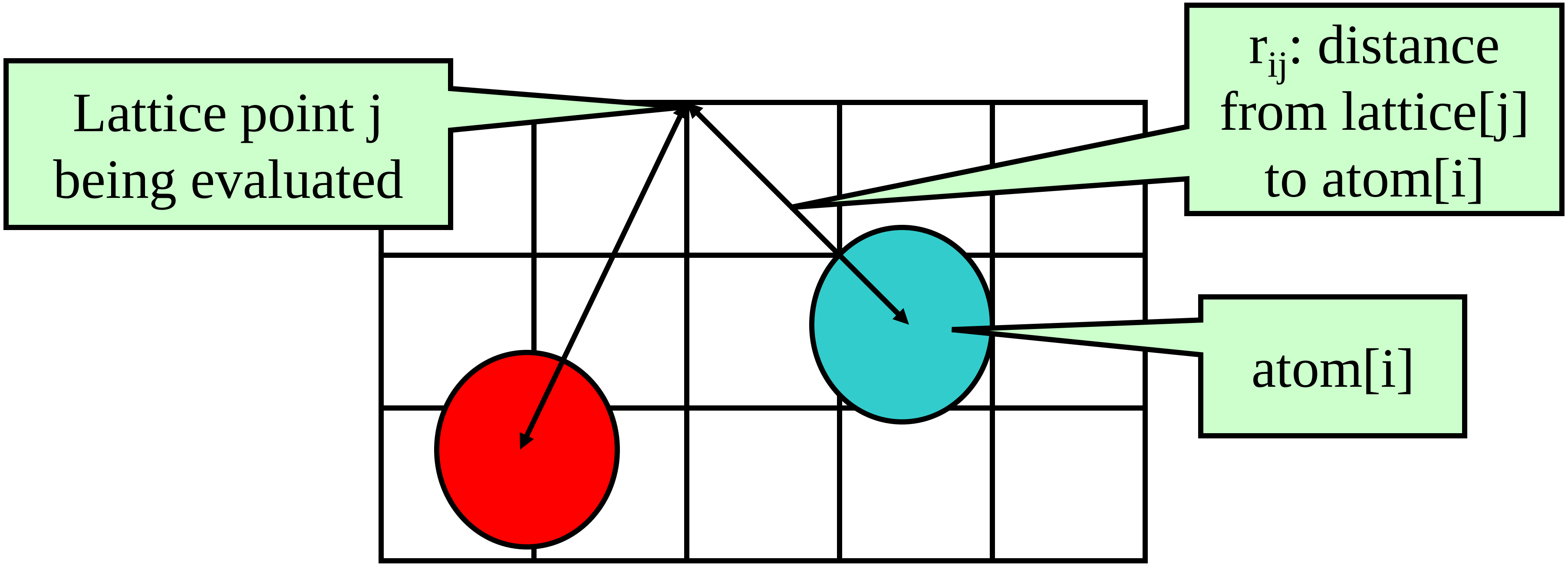


Isoleucine tRNA synthetase

# Direct Coulomb Summation (DCS) Algorithm Detail

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

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



Lattice point j being evaluated

$r_{ij}$: distance from lattice[j] to atom[i]

atom[i]

# 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;
   }
 }
}
```

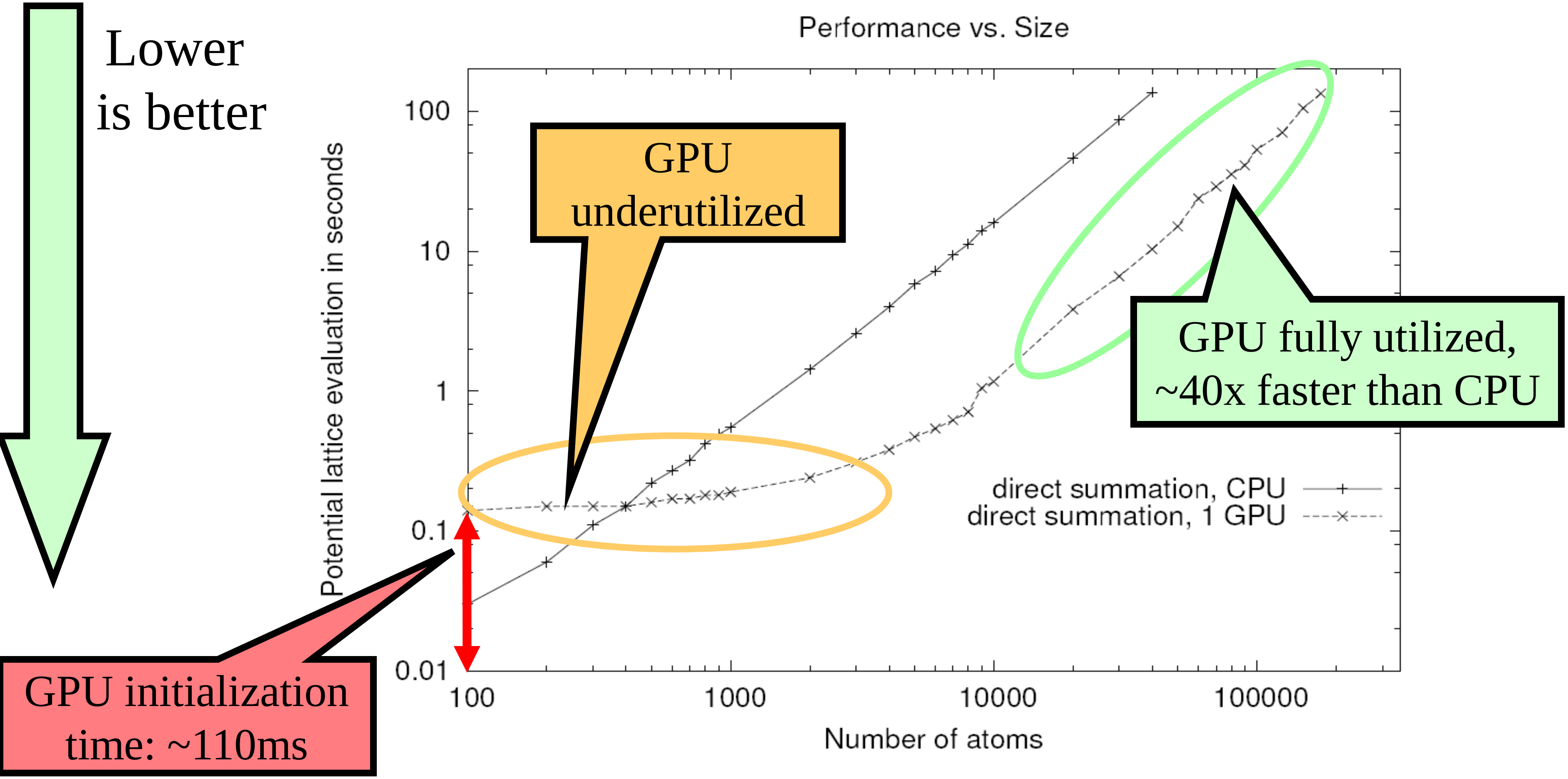
# DCS Algorithm Design Observations

- 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

# 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

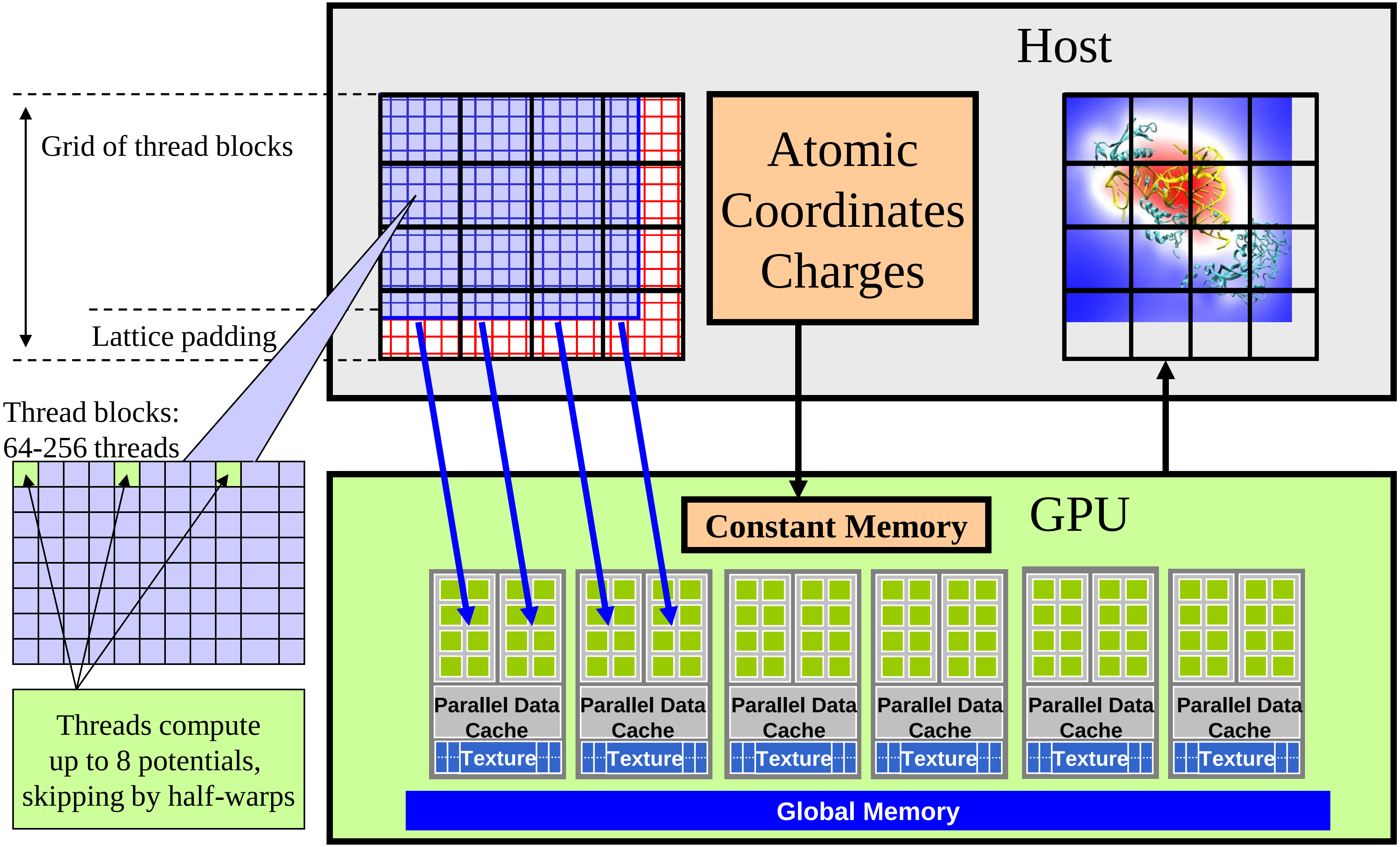


Accelerating molecular modeling applications with graphics processors.
J. Stone, J. Phillips, P. Freddolino, D. Hardy, L. Trabuco, K. Schulten.
*J. Comp. Chem.*, 28:2618-2640, 2007.

# DCS Observations for GPU Implementation

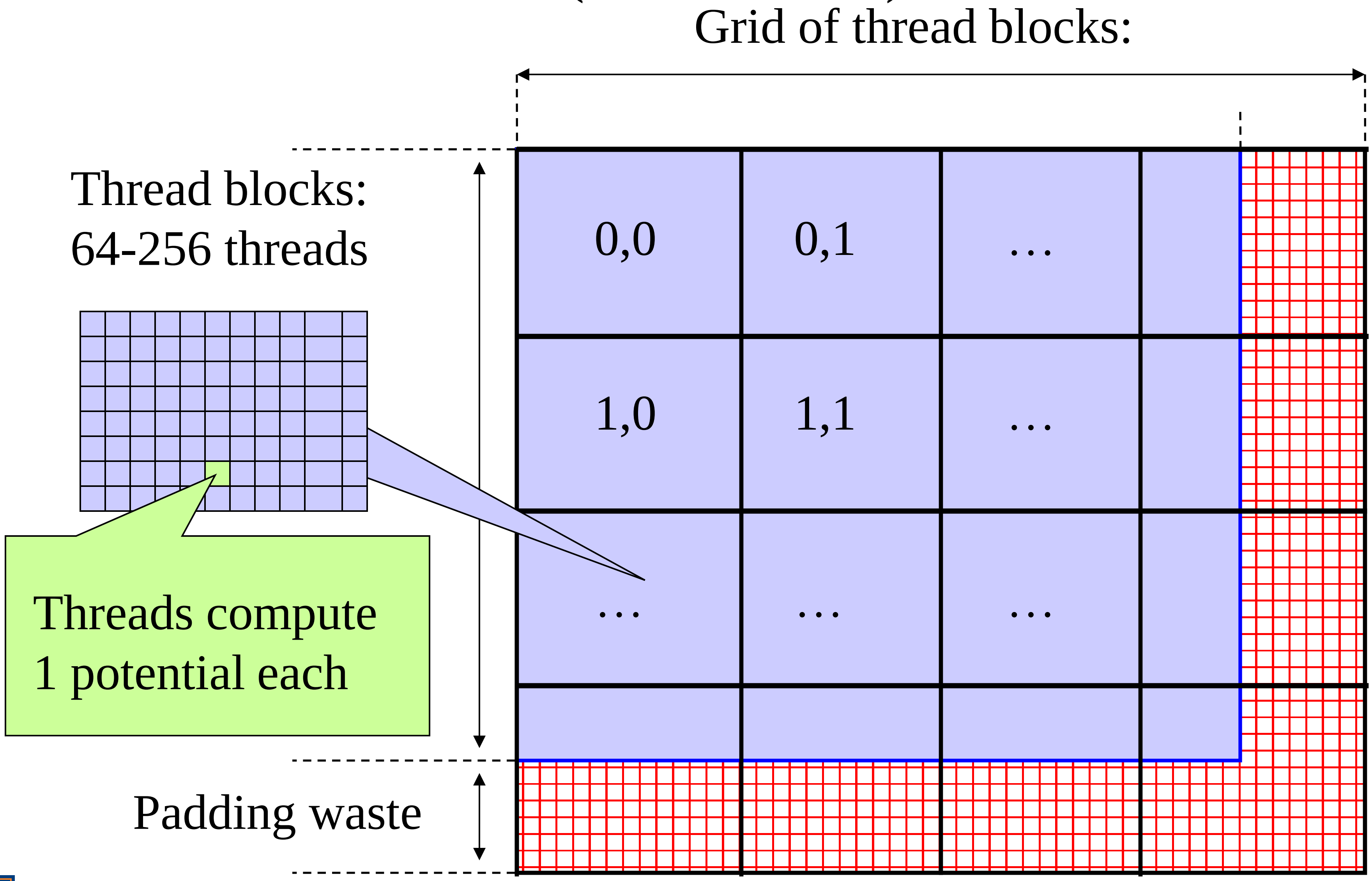
- Naive implementation has a low ratio of FP arithmetic operations to memory transactions (at least for a GPU…)
- The innermost loop will consume operands VERY quickly
- Since atoms are read-only, they are ideal candidates for texture memory or constant memory
- GPU implementations must access constant 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

# 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

# DCS Version 1: Const+Precalc 187 GFLOPS, 18.6 Billion Atom Evals/Sec (G80)

- 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, thread coarsening)

- 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

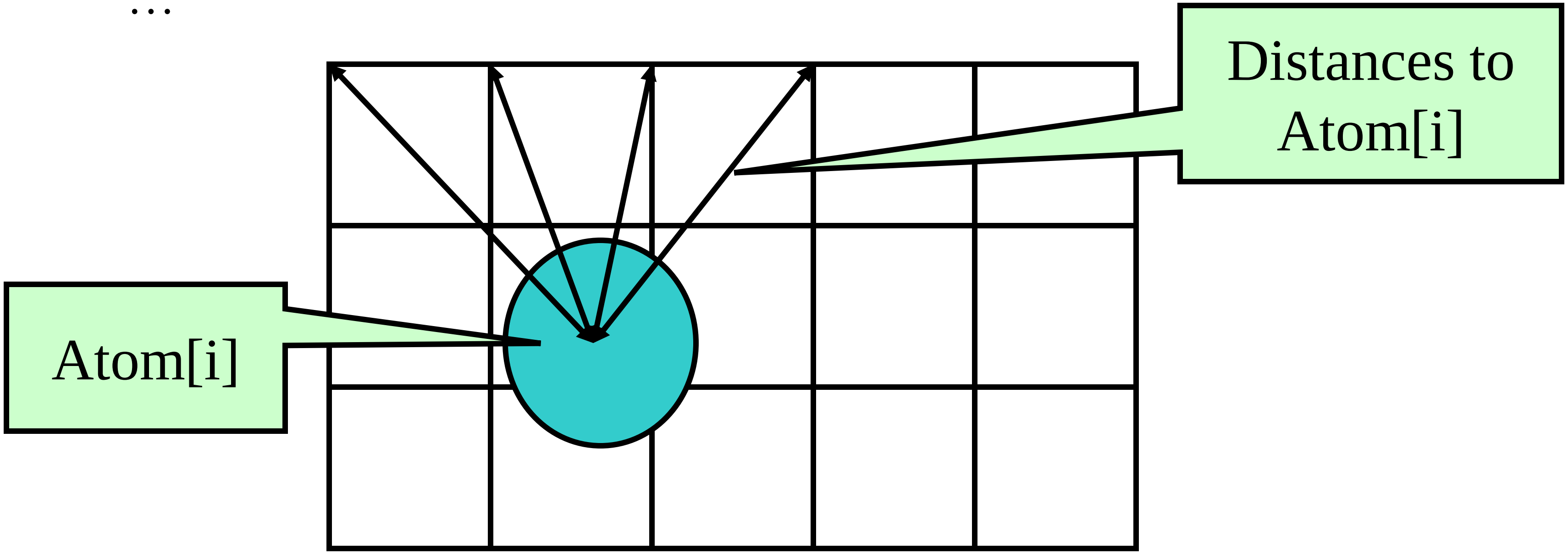
# DCS CUDA Algorithm: Unrolling Loops

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

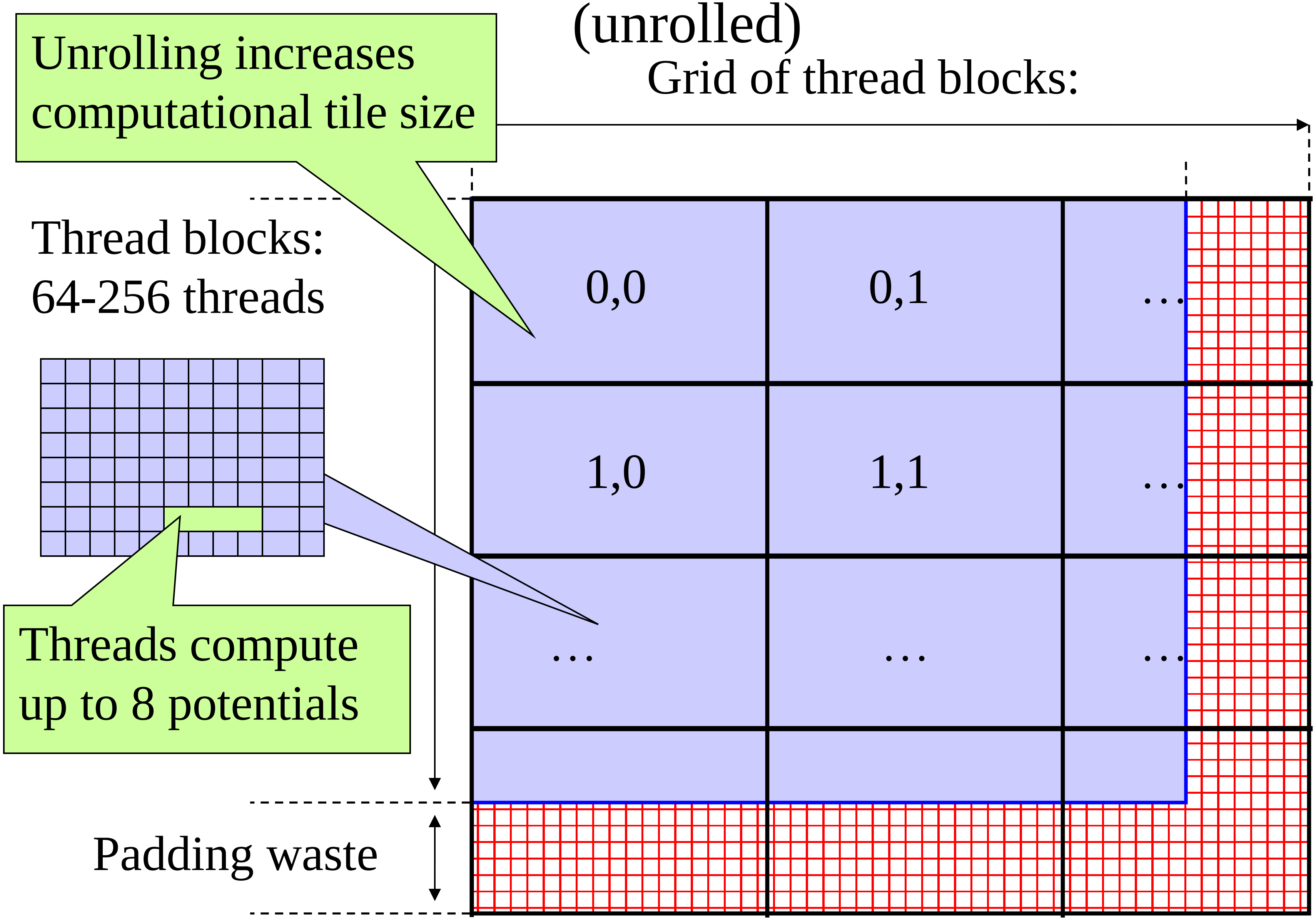
potential[j ] += atom[i].charge / $r_{ij}$

potential[j+1] += atom[i].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
0,0
0,1
…
1,0
1,1
…
…
…
…
Threads compute up to 8 potentials
Padding waste

# 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 4: Const+Loop Unrolling+Coalescing 291.5 GFLOPS, 39.5 Billion Atom Evals/Sec (G80)

- 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
- Source code is available by request

# 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;
```

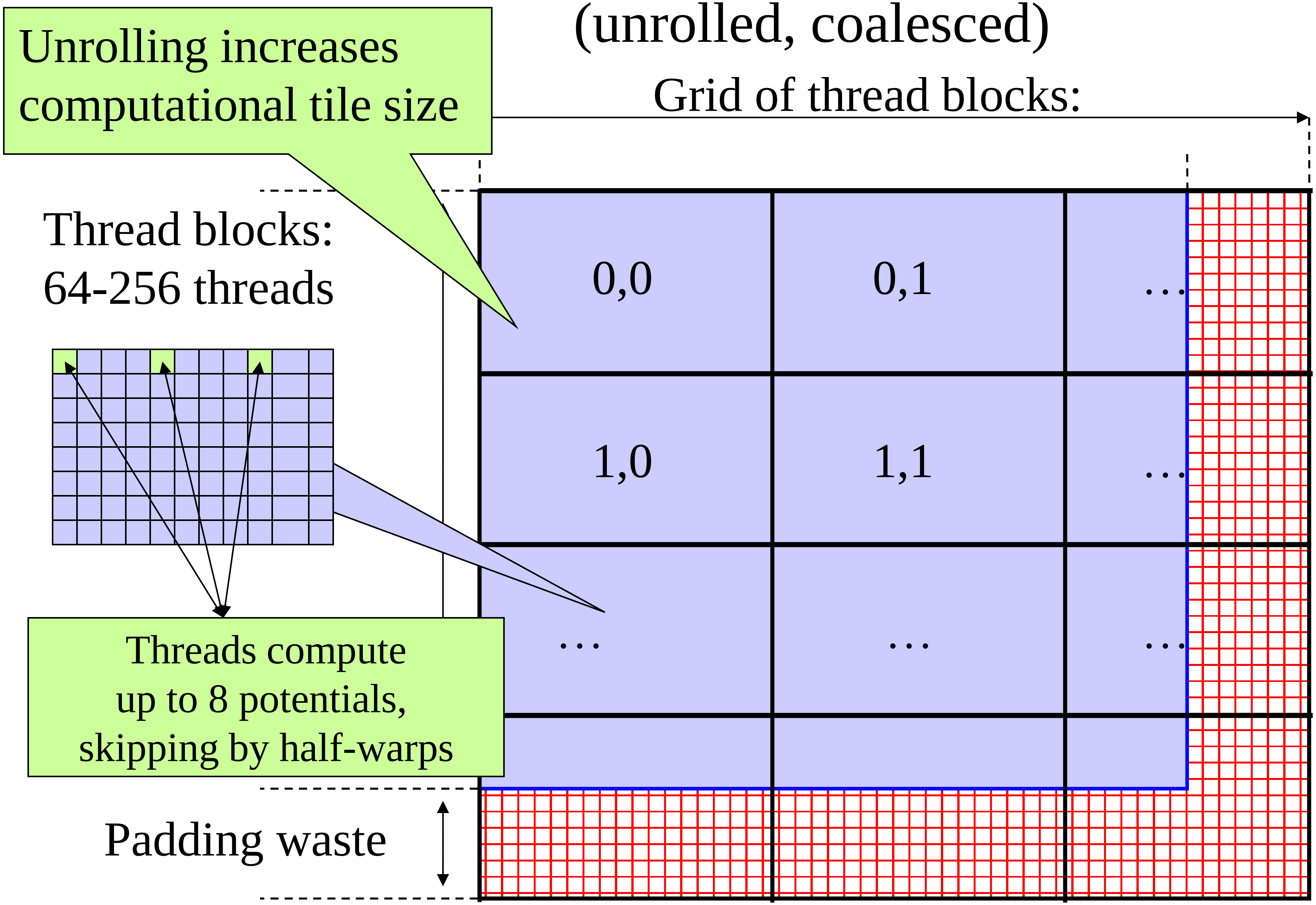
Points spaced for memory coalescing

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)



Unrolling increases computational tile size

Grid of thread blocks:

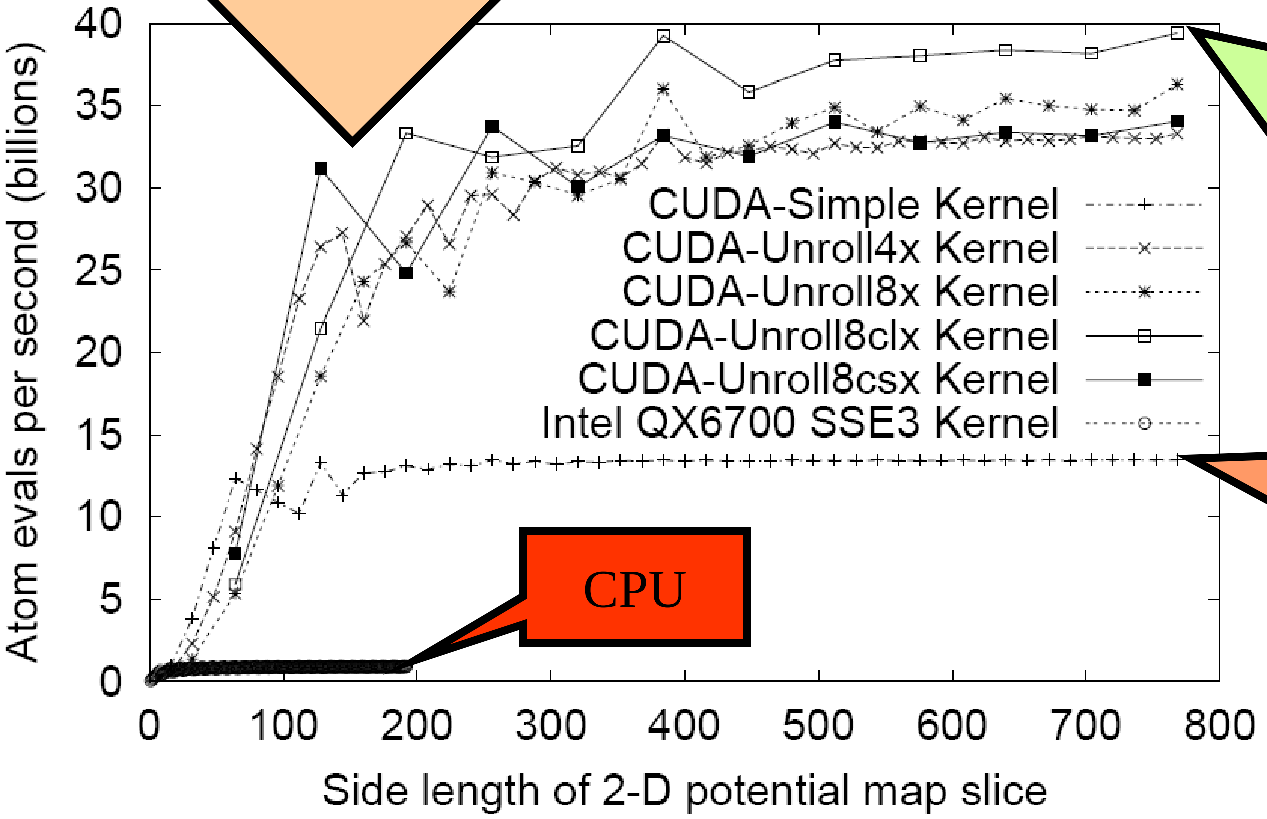
Thread blocks: 64-256 threads

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

CPU

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

# DCS Version 4 Inner Loop, Scalar OpenCL

```
…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 dx2 = dx1 + gridspacing_coalesce;
    float dx3 = dx2 + gridspacing_coalesce;
    float dx4 = dx3 + gridspacing_coalesce;
    float charge = atominfo[atomid].w;
    energyvalx1 += charge * native_rsqrt(dx1*dx1 + dyz2);
    energyvalx2 += charge * native_rsqrt(dx2*dx2 + dyz2);
    energyvalx3 += charge * native_rsqrt(dx3*dx3 + dyz2);
    energyvalx4 += charge * native_rsqrt(dx4*dx4 + dyz2);
  }
```

**Well-written CUDA code can often be easily ported to OpenCL if C++ features and pointer arithmetic aren't used in kernels.**

# DCS Version 4 Inner Loop (CUDA)

(only 4-way unrolling for conciseness to compare OpenCL)

```
…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 dx2 = dx1 + gridspacing_coalesce;
   float dx3 = dx2 + gridspacing_coalesce;
   float dx4 = dx3 + gridspacing_coalesce;
   float charge = atominfo[atomid].w;
   energyvalx1 += charge * rsqrtf(dx1*dx1 + dyz2);
   energyvalx2 += charge * rsqrtf(dx2*dx2 + dyz2);
   energyvalx3 += charge * rsqrtf(dx3*dx3 + dyz2);
   energyvalx4 += charge * rsqrtf(dx4*dx4 + dyz2);
  }
```

# DCS Version 4 Inner Loop, Vectorized OpenCL

```
 float4 gridspacing_u4 = { 0.f, 1.f, 2.f, 3.f };
 gridspacing_u4 *= gridspacing_coalesce;
 float4 energyvalx=0.0f;
…
 for (atomid=0; atomid<numatoms; atomid++) {
    float dy = coory - atominfo[atomid].y;
    float dyz2 = (dy * dy) + atominfo[atomid].z;
    float4 dx = gridspacing_u4 + (coorx – atominfo[atomid].x);
    float charge = atominfo[atomid].w;
    energyvalx1 += charge * native_rsqrt(dx1*dx1 + dyz2);
 }
```

**CPUs, AMD GPUs, and Cell often perform better with vectorized kernels.
Use of vector types may increase register pressure; sometimes a delicate balance…**

# Infinite vs. Cutoff Potentials

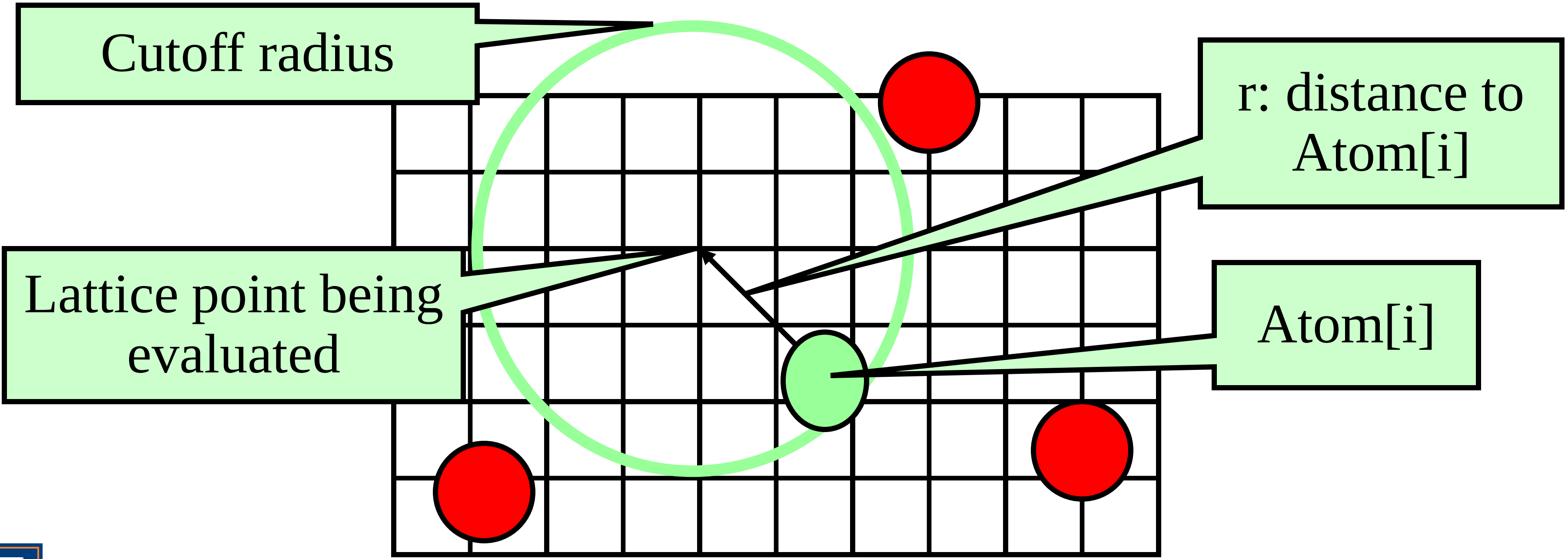
- Infinite range potential:
  - All atoms contribute to all lattice points
  - Summation algorithm has quadratic complexity
- Cutoff (range-limited) potential:
  - Atoms contribute within cutoff distance to lattice points
  - Summation algorithm has linear time complexity
  - Has many applications in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

# Cutoff Summation

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

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

- Smoothing function s(r) is algorithm dependent



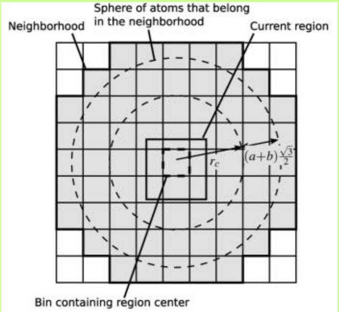
# Cutoff Summation on the GPU

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

CPU handles overflowed bins

**Constant memory**

Bin-Region neighborlist

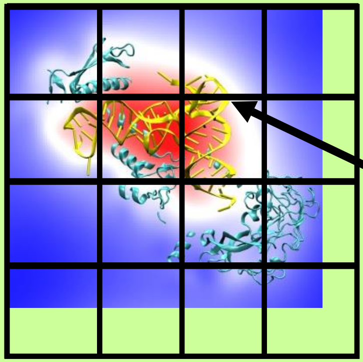

Neighborhood
Sphere of atoms that belong in the neighborhood
Current region
$r_c$
$(a+b)\frac{\sqrt{3}}{2}$
Bin containing region center

Atoms

Global memory

**Bins of 8 atoms**

Potential map regions



Process atom bins for current potential map region

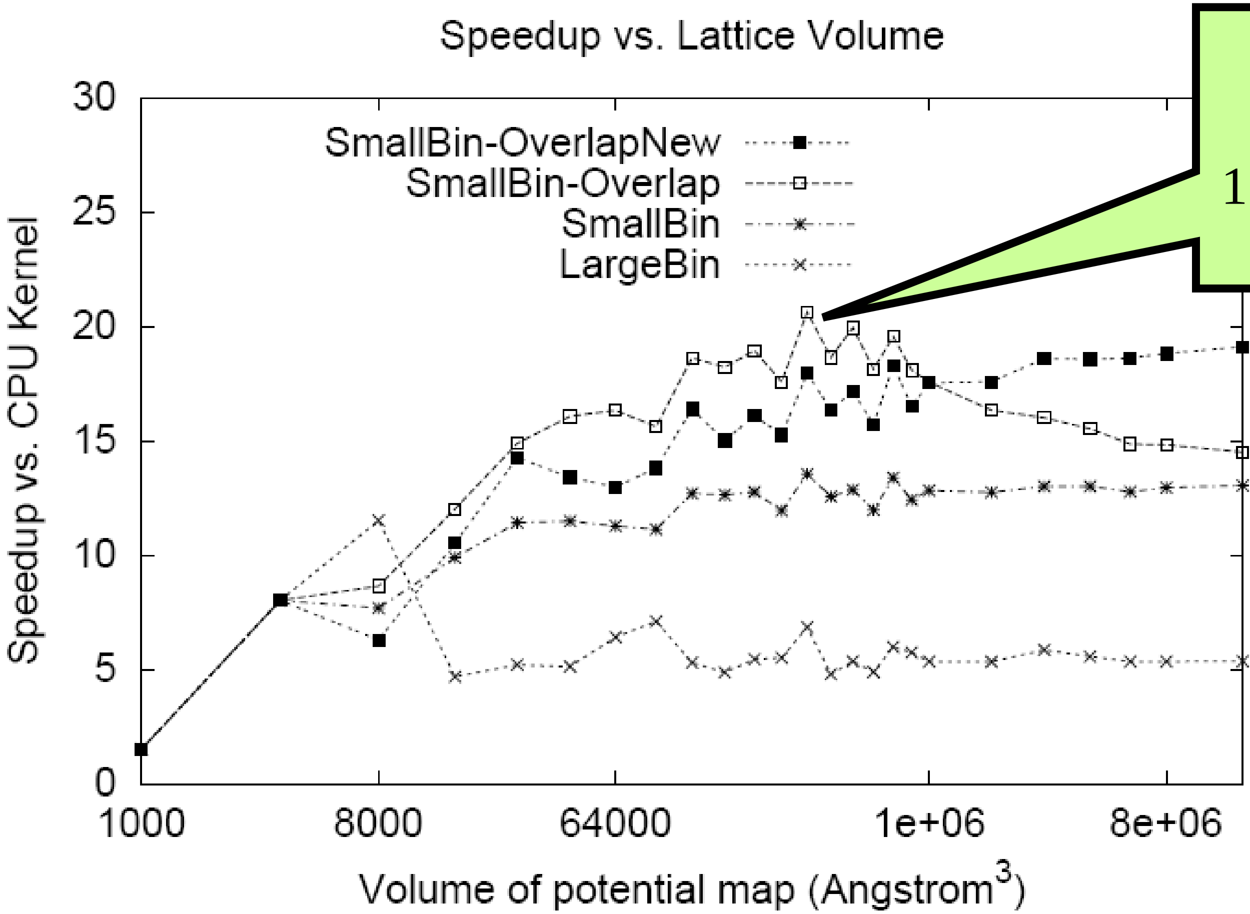
Shared memory

Atom bin

# Using the CPU to Improve GPU Performance

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

# Cutoff Summation Runtime



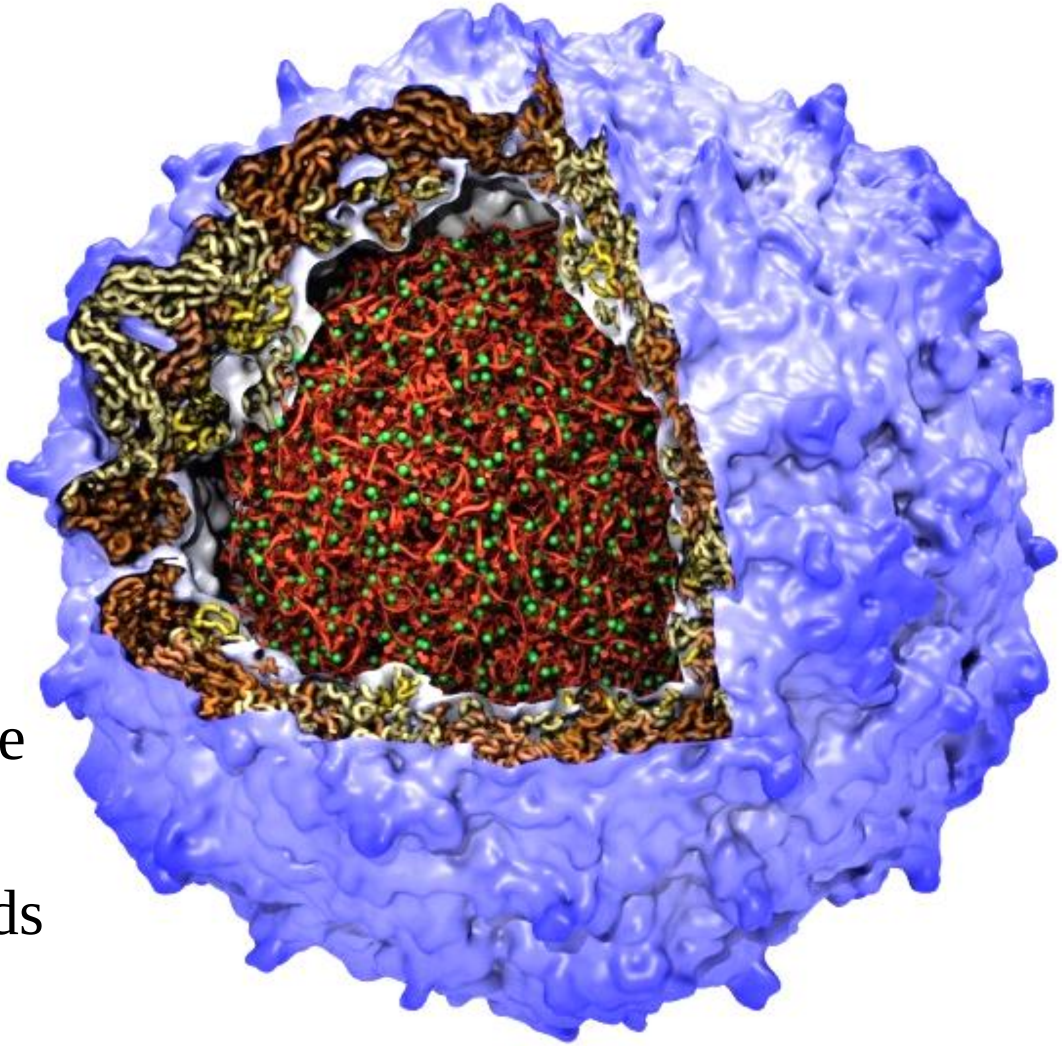
GPU cutoff with CPU overlap:

17x-21x faster than CPU core

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

# Molecular Surface Visualization

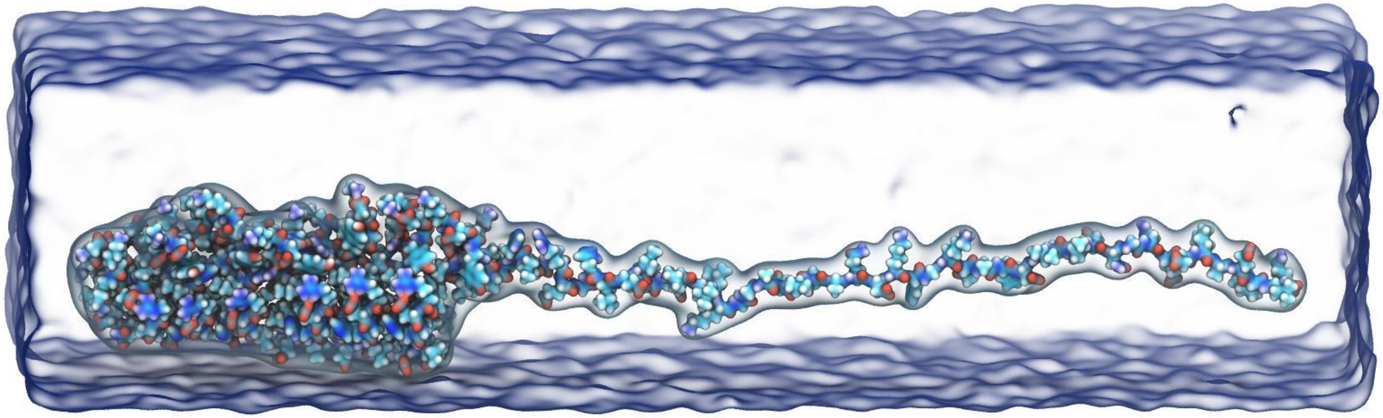
- Large biomolecular complexes are difficult to interpret with atomic detail graphical representations
- Even secondary structure representations become cluttered
- Surface representations are easier to use when greater abstraction is desired, but are computationally costly
- Most surface display methods incapable of animating dynamics of large structures w/ millions of particles



**Poliovirus**

# VMD “QuickSurf” Representation

- Displays continuum of structural detail:
  - All-atom models
  - Coarse-grained models
  - Cellular scale models
  - Multi-scale models: All-atom + CG,  Brownian + Whole Cell
  - Smoothly variable between full detail, and reduced resolution representations of very large complexes

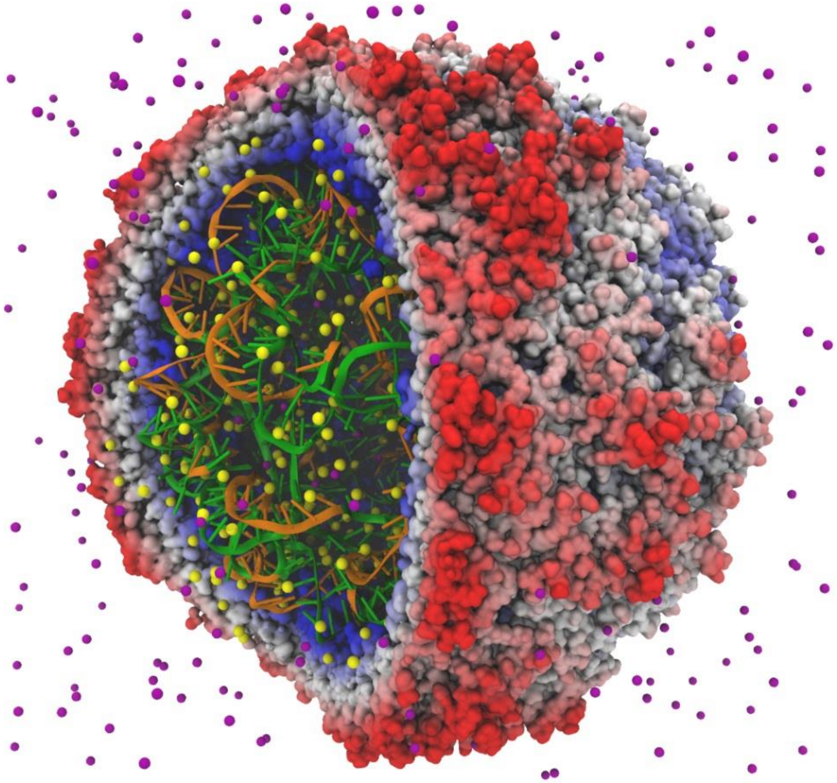


**Fast Visualization of Gaussian Density Surfaces for Molecular Dynamics and Particle System Trajectories.**

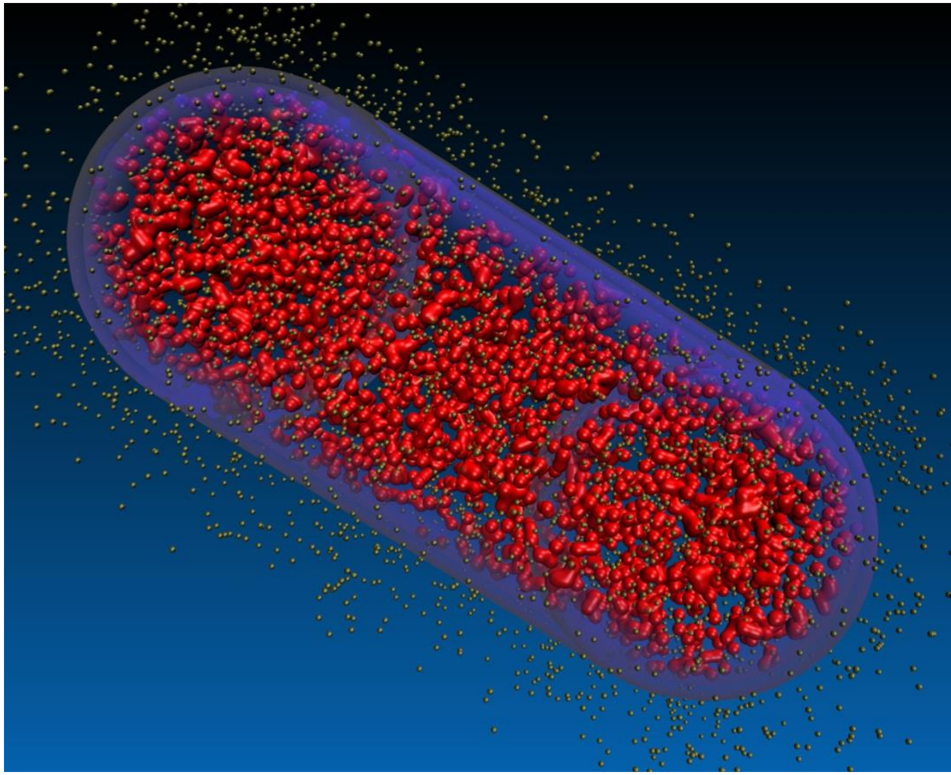
M. Krone, J. E. Stone, T. Ertl, K. Schulten. *EuroVis Short Papers*, pp. 67-71, 2012

# VMD "QuickSurf" Representation

- Uses multi-core CPUs and GPU acceleration to enable **smooth real-time animation** of MD trajectories
- Linear-time algorithm, scales to millions of particles, as limited by memory capacity

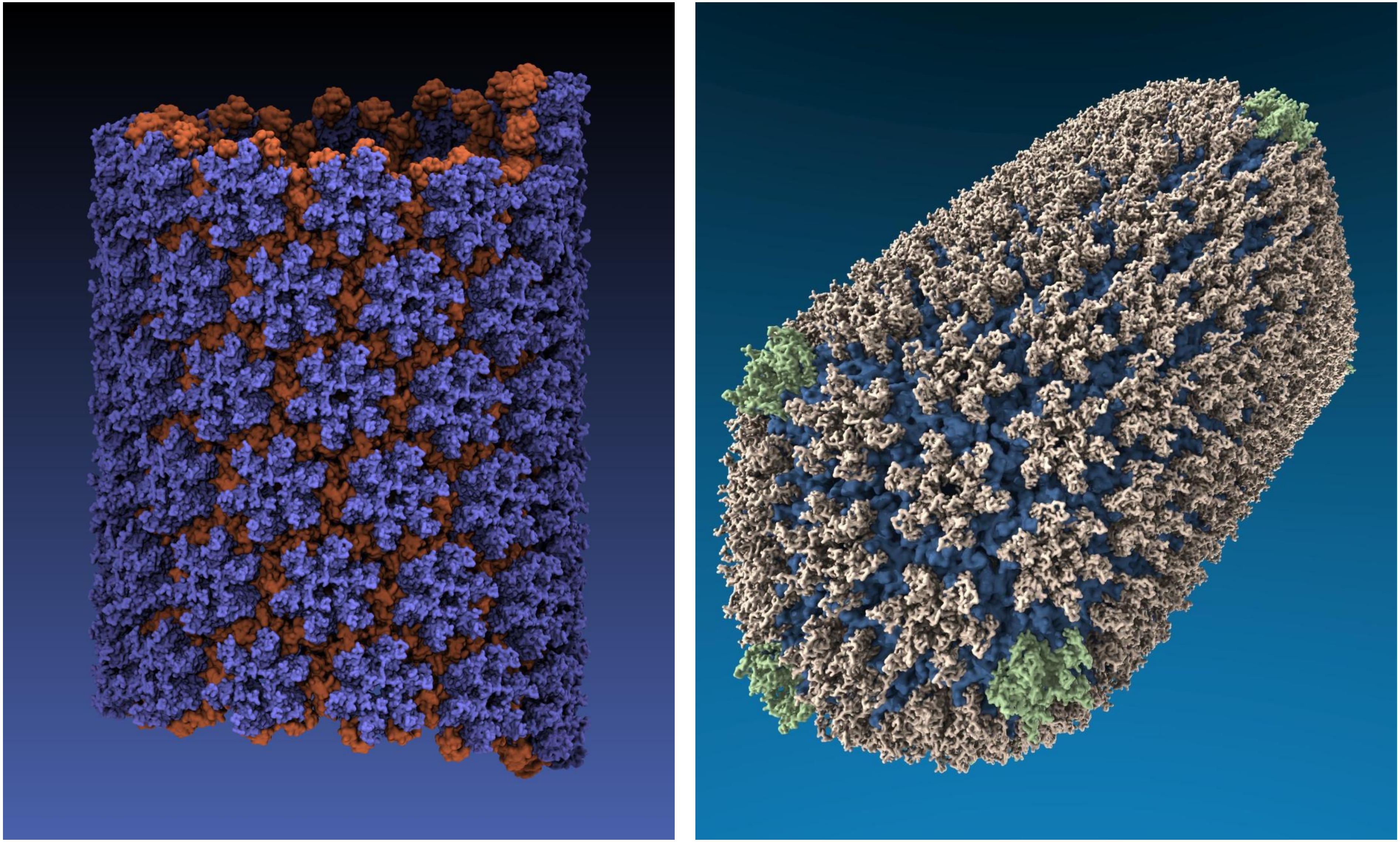


**Satellite Tobacco Mosaic Virus**



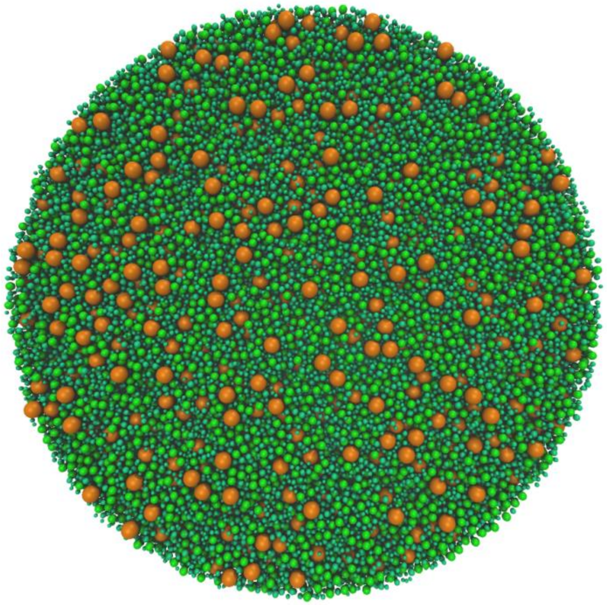
**Lattice Cell Simulations**

# VMD “QuickSurf” Representation

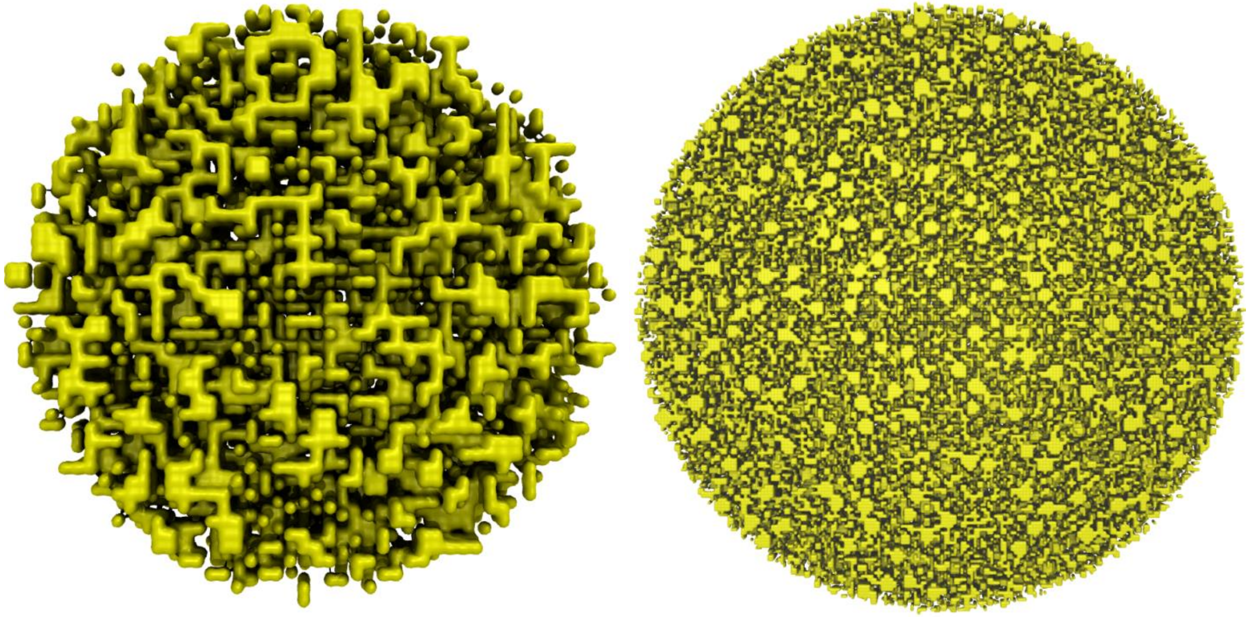


**All-atom HIV capsid simulations**

# QuickSurf Representation of Lattice Cell Models



**Continuous particle based model – often 70 to 300 million particles**



**Discretized lattice models derived from continuous model shown in VMD QuickSurf representation**

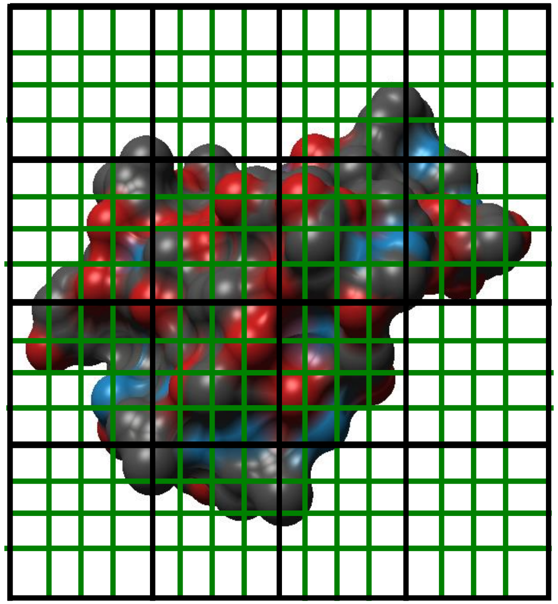
**Lattice Microbes: High-performance stochastic simulation method for the reaction-diffusion master equation**
E. Roberts, J. E. Stone, and Z. Luthey-Schulten.
J. Computational Chemistry 34 (3), 245-255, 2013.

# QuickSurf Algorithm Overview

- Build spatial acceleration data structures, optimize data for GPU
- Compute 3-D density map, 3-D volumetric texture map:

$$\rho(\vec{r}; \vec{r}_1, \vec{r}_2, \ldots, \vec{r}_N) = \sum_{i=1}^{N} e^{\frac{-|\vec{r}-\vec{r}_i|^2}{2\alpha^2}}$$

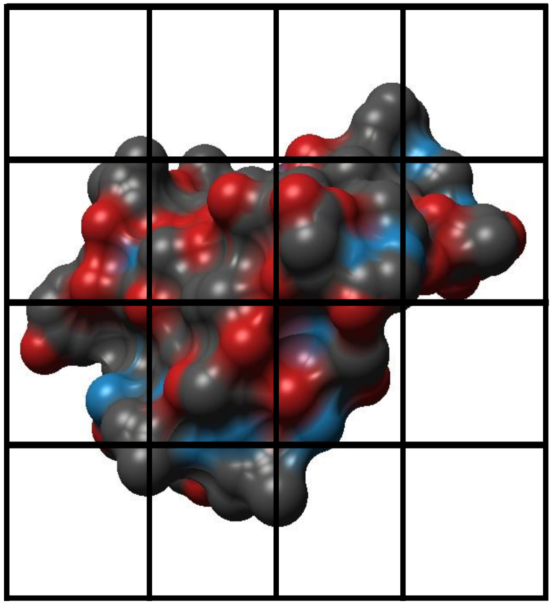
- Extract isosurface for a user-defined density value



**3-D density map lattice, spatial acceleration grid, and extracted surface**

# QuickSurf Particle Sorting, Bead Generation, Spatial Hashing

- Particles sorted into spatial acceleration grid:
  - Selected atoms or residue “beads” converted lattice coordinate system
  - Each particle/bead assigned cell index, sorted w/NVIDIA Thrust template library
- Complication:
  - Thrust allocates GPU mem. on-demand, no recourse if insufficient memory, have to re-gen QuickSurf data structures if caught by surprise!
- Workaround:
  - Pre-allocate guesstimate workspace for Thrust
  - Free the Thrust workspace right before use
  - Newest Thrust allows user-defined allocator code…



**Coarse resolution spatial acceleration grid**

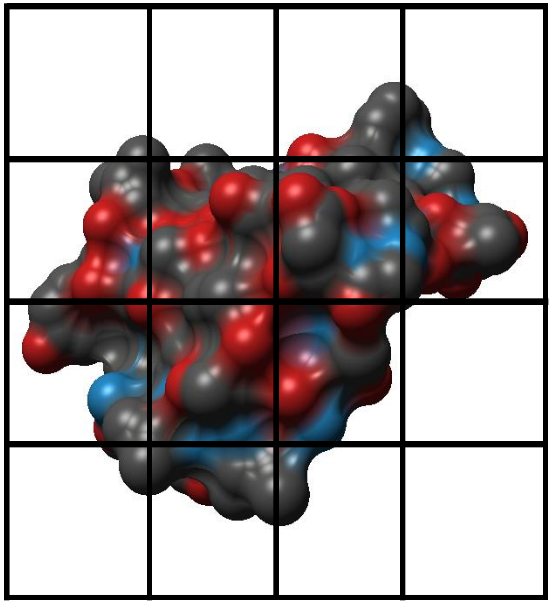
# Spatial Hashing Algorithm Steps/Kernels

1) Compute bin index for each atom, store to memory w/ atom index

2) **Sort** list of bin and atom index tuples (1) by bin index **(thrust kernel)**

3) Count atoms in each bin (2) using a **parallel prefix sum, aka *scan***, compute the destination index for each atom, store per-bin starting index and atom count **(thrust kernel)**

4) Write atoms to the output indices computed in (3), and we have completed the data structure

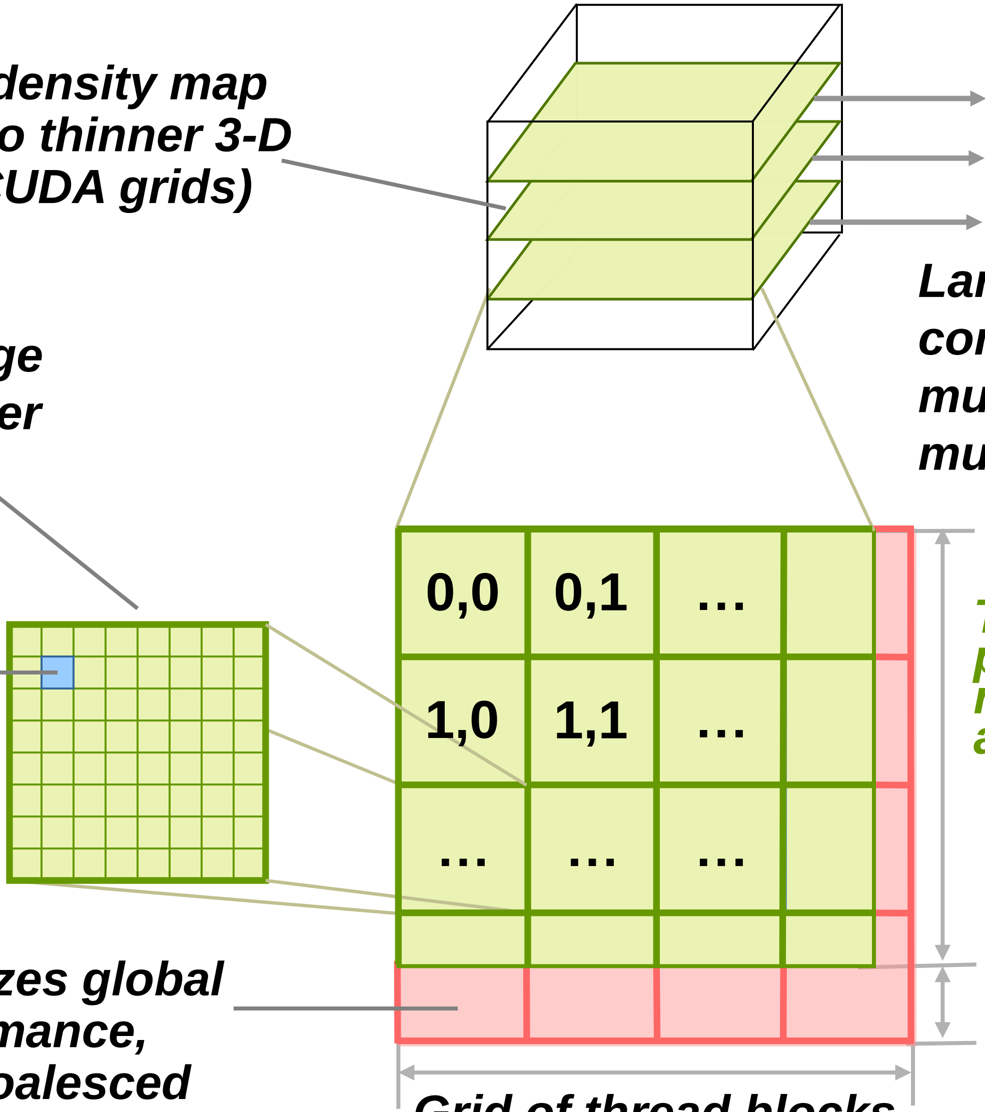


**QuickSurf uniform grid spatial subdivision data structure**

# QuickSurf and Limited GPU Global Memory

- High resolution molecular surfaces require a fine lattice spacing
- Memory use grows cubically with decreased lattice spacing
- Not typically possible to compute a surface in a single pass, so we loop over sub-volume "chunks" until done…
- Chunks pre-allocated and sized to GPU global mem capacity to prevent unexpected memory allocation failure while animating…
- Complication:
  - Thrust allocates GPU mem. on-demand, no recourse if insufficient memory, have to re-gen QuickSurf data structures if caught by surprise!
- Workaround:
  - Pre-allocate guesstimate workspace for Thrust
  - Free the Thrust workspace right before use
  - Newest Thrust allows user-defined allocator code…

# QuickSurf Density Parallel Decomposition



***QuickSurf 3-D density map decomposes into thinner 3-D slabs/slices (CUDA grids)***

***Large volume computed in multiple passes, or multiple GPUs***

***Small 8x8 thread blocks afford large per-thread register count, shared memory***

***Each thread computes one or more density map lattice points***

***Threads producing results that are used***

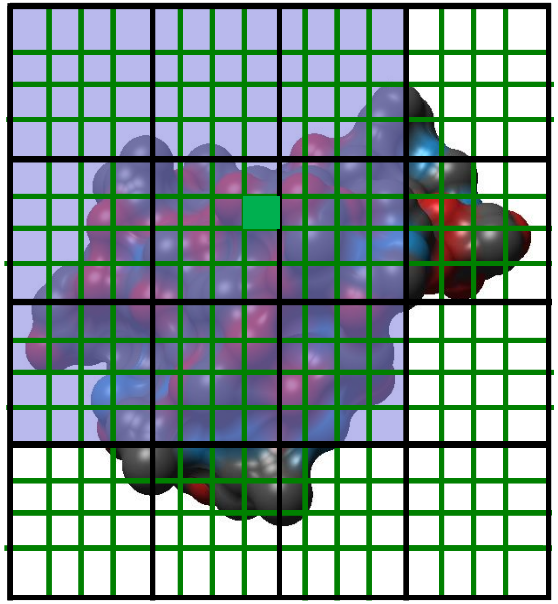
***Inactive threads, region of discarded output***

***Padding optimizes global memory performance, guaranteeing coalesced global memory accesses***

***Grid of thread blocks***

# QuickSurf Density Map Algorithm

- Spatial acceleration grid cells are sized to match the cutoff radius for the exponential, beyond which density contributions are negligible
- Density map lattice points computed by summing density contributions from particles in 3x3x3 grid of neighboring spatial acceleration cells
- Volumetric texture map is computed by summing particle colors normalized by their individual density contribution



**3-D density map lattice point and the neighboring spatial acceleration cells it references**

# QuickSurf Density Map Kernel Optimizations

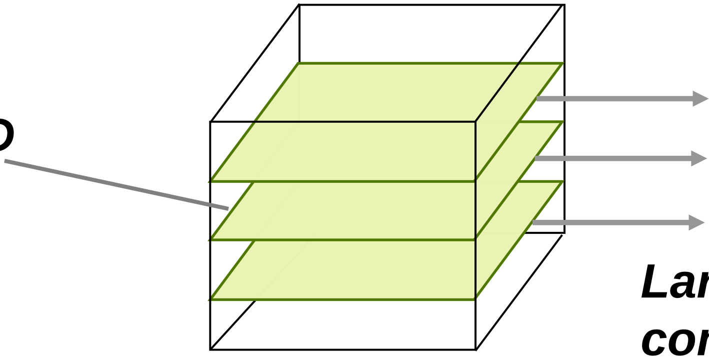
- Compute reciprocals, prefactors, other math on the host CPU prior to kernel launch
- Use of **intN** and **floatN** vector types in CUDA kernels for improved global memory bandwidth
- **Thread coarsening**: one thread computes multiple output densities and colors
- Input data and **register tiling**: share blocks of input, partial distances in regs shared among multiple outputs
- Global memory **(L1 cache) broadcasts**: all threads in the block traverse the same atom/particle at the same time

# QuickSurf Density Map Kernel Snippet…

```
for (zab=zabmin; zab<=zabmax; zab++) {
  for (yab=yabmin; yab<=yabmax; yab++) {
   for (xab=xabmin; xab<=xabmax; xab++) {
    int abcellidx = zab * acplanesz + yab * acncells.x + xab;
    uint2 atomstartend = cellStartEnd[abcellidx];
    if (atomstartend.x != GRID_CELL_EMPTY) {
     for (unsigned int atomid=atomstartend.x; atomid<atomstartend.y; atomid++) {
      float4 atom = sorted_xyzr[atomid];
      float dx = coorx - atom.x;        float dy = coory - atom.y;      float dz = coorz - atom.z;
      float dxy2 = dx*dx + dy*dy;
      float r21 = (dxy2 + dz*dz) * atom.w;
      densityval1 += exp2f(r21);
       /// Loop unrolling and register tiling benefits begin here……
      float dz2 = dz + gridspacing;
      float r22 = (dxy2 + dz2*dz2) * atom.w;
      densityval2 += exp2f(r22);
      /// More loop unrolling ….
```

# QuickSurf Marching Cubes Isosurface Extraction

- Isosurface is extracted from each density map "chunk", and either copied back to the host, or **rendered directly** out of GPU global memory via **CUDA/OpenGL interop**
- All MC memory buffers are pre-allocated to prevent significant overhead when animating a simulation trajectory



***QuickSurf 3-D density map decomposes into thinner 3-D slabs/slices (CUDA grids)***

***…***

***Chunk 2***

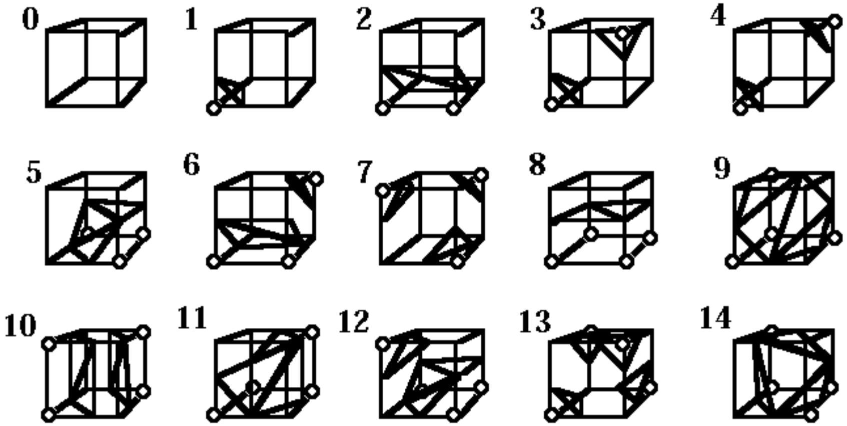
***Chunk 1***

***Chunk 0***

***Large volume computed in multiple passes***

# Brief Marching Cubes Isosurface Extraction Overview

- Given a 3-D volume of scalar density values and a requested surface density value, marching cubes computes vertices and triangles that compose the requested surface triangle mesh
- Each MC "cell" (a cube with 8 density values at its vertices) produces a variable number of output vertices depending on how many edges of the cell contain the requested isovalue…
- Use **scan()** to compute the output indices so that each worker thread has **conflict-free output** of vertices/triangles



# Brief Marching Cubes Isosurface Extraction Overview

- Once the output vertices have been computed and stored, we compute surface normals and colors for each of the vertices
- Although the separate normals+colors pass reads the density map again, molecular surfaces tend to generate a small percentage of MC cells containing triangles, we avoid wasting interpolation work
- We use CUDA **tex3D()** hardware 3-D texture mapping:
  - Costs double the texture memory and a one copy from GPU global memory to the target texture map with **cudaMemcpy3D()**
  - Still roughly 2x faster than doing color interpolation without the texturing hardware, at least on GT200 and Fermi hardware
  - Kepler has new texture cache memory path that may make it feasible to do our own color interpolation and avoid the use of extra 3-D texture memory and associated copy, with acceptable performance

# QuickSurf Marching Cubes Isosurface Extraction

- Our optimized MC implementation computes per-vertex surface normals, colors, and outperforms the NVIDIA SDK sample by a fair margin on Fermi GPUs
- Complications:
  - Even on a 6GB Quadro 7000, GPU global memory is under great strain when working with large molecular complexes, e.g. viruses
  - Marching cubes involves a parallel prefix sum (scan) to compute target indices for writing resulting vertices
  - We use Thrust for scan, has the same memory allocation issue mentioned earlier for the sort, so we use the same workaround
  - The number of output vertices can be huge, but we rarely have sufficient GPU memory for this – we use a fixed size vertex output buffer and hope our heuristics don't fail us

# QuickSurf Performance GeForce GTX 580

| **Molecular system** | **Atoms** | **Resolution** | $\mathbf{T_{sort}}$ | $\mathbf{T_{density}}$ | $\mathbf{T_{MC}}$ | **# vertices** | **FPS** |
|---|---|---|---|---|---|---|---|
| MscL | 111,016 | 1.0Å | 0.005 | 0.023 | 0.003 | 0.7 M | 28 |
| STMV capsid | 147,976 | 1.0Å | 0.007 | 0.048 | 0.009 | 2.4 M | 13.2 |
| Poliovirus capsid | 754,200 | 1.0Å | 0.01 | 0.18 | 0.05 | 9.2 M | 3.5 |
| STMV w/ water | 955,225 | 1.0Å | 0.008 | 0.189 | 0.012 | 2.3 M | 4.2 |
| Membrane | 2.37 M | 2.0Å | 0.03 | 0.17 | 0.016 | 5.9 M | 3.9 |
| Chromatophore | 9.62 M | 2.0Å | 0.16 | 0.023 | 0.06 | 11.5 M | 3.4 |
| Membrane w/ water | 22.77 M | 4.0Å | 4.4 | 0.68 | 0.01 | 1.9 M | 0.18 |

**Fast Visualization of Gaussian Density Surfaces for Molecular Dynamics and Particle System Trajectories.**

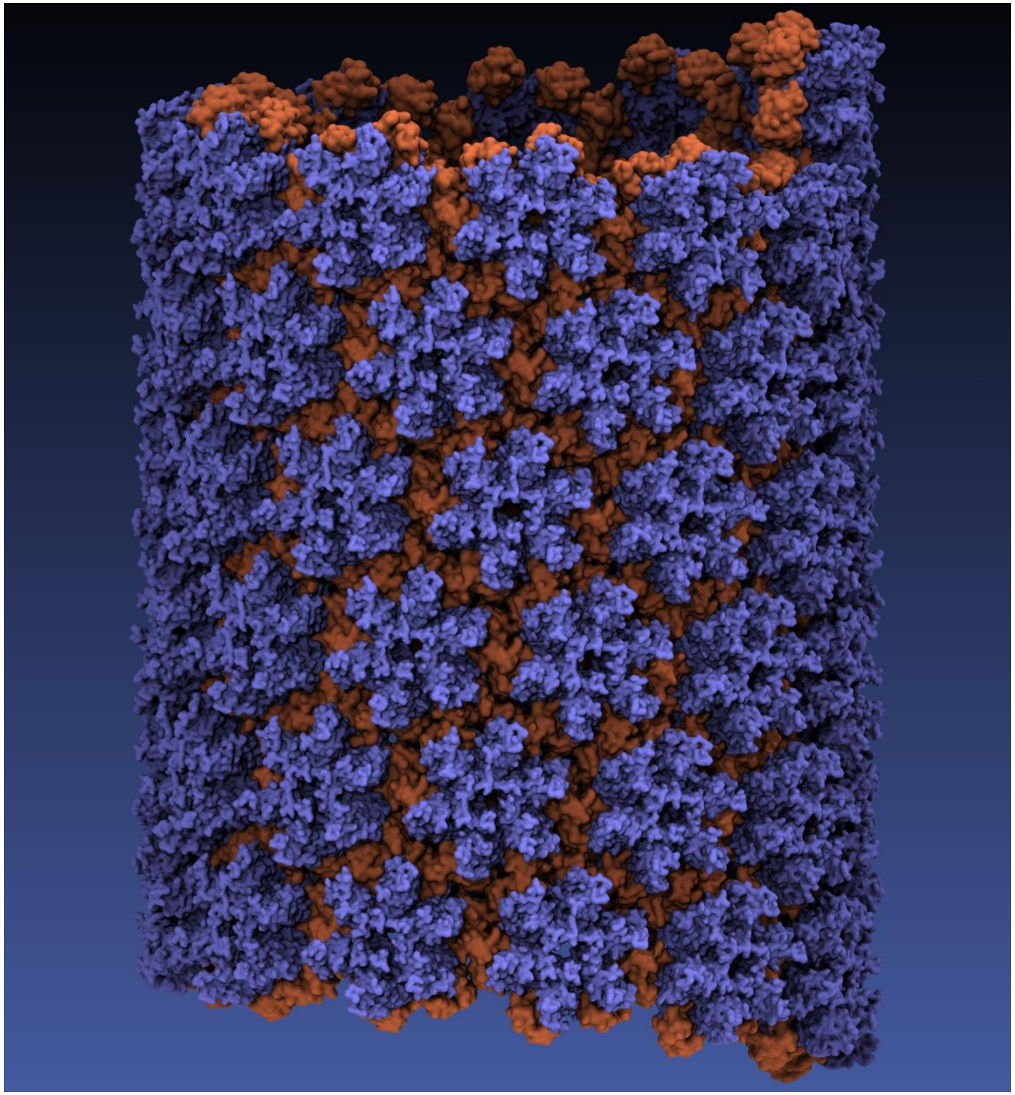
M. Krone, J. E. Stone, T. Ertl, K. Schulten. *EuroVis Short Papers*, pp. 67-71, 2012

# Extensions and Analysis Uses for QuickSurf Triangle Mesh

- Curved PN triangles:
  - We have performed tests with post-processing the resulting triangle mesh and using curved PN triangles to generate smooth surfaces with a larger grid spacing, for increased performance
  - Initial results demonstrate some potential, but there can be pathological cases where MC generates long skinny triangles, causing unsightly surface creases
- Analysis uses (beyond visualization):
  - Minor modifications to the density map algorithm allow rapid computation of solvent accessible surface area by summing the areas in the resulting triangle mesh
  - Modifications to the density map algorithm will allow it to be used for MDFF (molecular dynamics flexible fitting)
  - Surface triangle mesh can be used as the input for computing the electrostatic potential field for mesh-based algorithms

# Challenge: Support Interactive QuickSurf for Large Structures on Mid-Range GPUs

- Structures such as HIV initially needed large (6GB) GPU memory to generate fully-detailed surface renderings
- Goals and approach:
  - **Avoid slow CPU-fallback!**
  - Incrementally change algorithm phases to use more compact data types, while maintaining performance
  - Specialize code for different performance/memory capacity cases



# Improving QuickSurf Memory Efficiency

- Both host and GPU memory capacity limitations are a significant concern when rendering surfaces for virus structures such as HIV or for large cellular models which can contain hundreds of millions of particles
- The original QuickSurf implementation used single-precision floating point for output vertex arrays and textures
- Judicious use of reduced-precision numerical representations, cut the overall memory footprint of the entire QuickSurf algorithm to half of the original
  - Data type changes made throughout the entire chain from density map computation through all stages of Marching Cubes

# Supporting Multiple Data Types for QuickSurf Density Maps and Marching Cubes Vertex Arrays

- The major algorithm components of QuickSurf are now used for many other purposes:
  - Gaussian density map algorithm now used for MDFF Cryo EM density map fitting methods in addition to QuickSurf
  - Marching Cubes routines also used for Quantum Chemistry visualizations of molecular orbitals
- Rather than simply changing QuickSurf to use a particular internal numerical representation, it is desirable to instead use **CUDA C++ templates** to make type-generic versions of the key objects, kernels, and output vertex arrays
- Accuracy-sensitive algorithms use high-precision data types, performance and memory capacity sensitive cases use quantized or reduced precision approaches

# Minimizing the Impact of Generality on QuickSurf Code Complexity

- A critical factor in the simplicity of supporting multiple QuickSurf data types arises from the so-called ***"gather"*** oriented algorithm we employ
  - Internally, all in-register arithmetic is single-precision
  - Data conversions to/from compressed or reduced precision data types are performed on-the-fly as needed
- Small **inlined** type conversion routines are defined for each of the cases we want to support
- Key QuickSurf kernels are genericized using C++ template syntax, and the compiler "connects the dots" to automatically generate type-specific kernels as needed

# Example Templated Density Map Kernel

```
template<class DENSITY, class VOLTEX>
__global__ static void
gaussdensity_fast_tex_norm(int natoms,
                            const float4 * RESTRICT sorted_xyzr,
                            const float4 * RESTRICT sorted_color,
                            int3 numvoxels,
                            int3 acncells,
                           float acgridspacing,
                           float invacgridspacing,
                           const uint2 * RESTRICT cellStartEnd,
                           float gridspacing, unsigned int z,
                           DENSITY * RESTRICT densitygrid,
                           VOLTEX * RESTRICT voltexmap,
                          float invisovalue) {
```

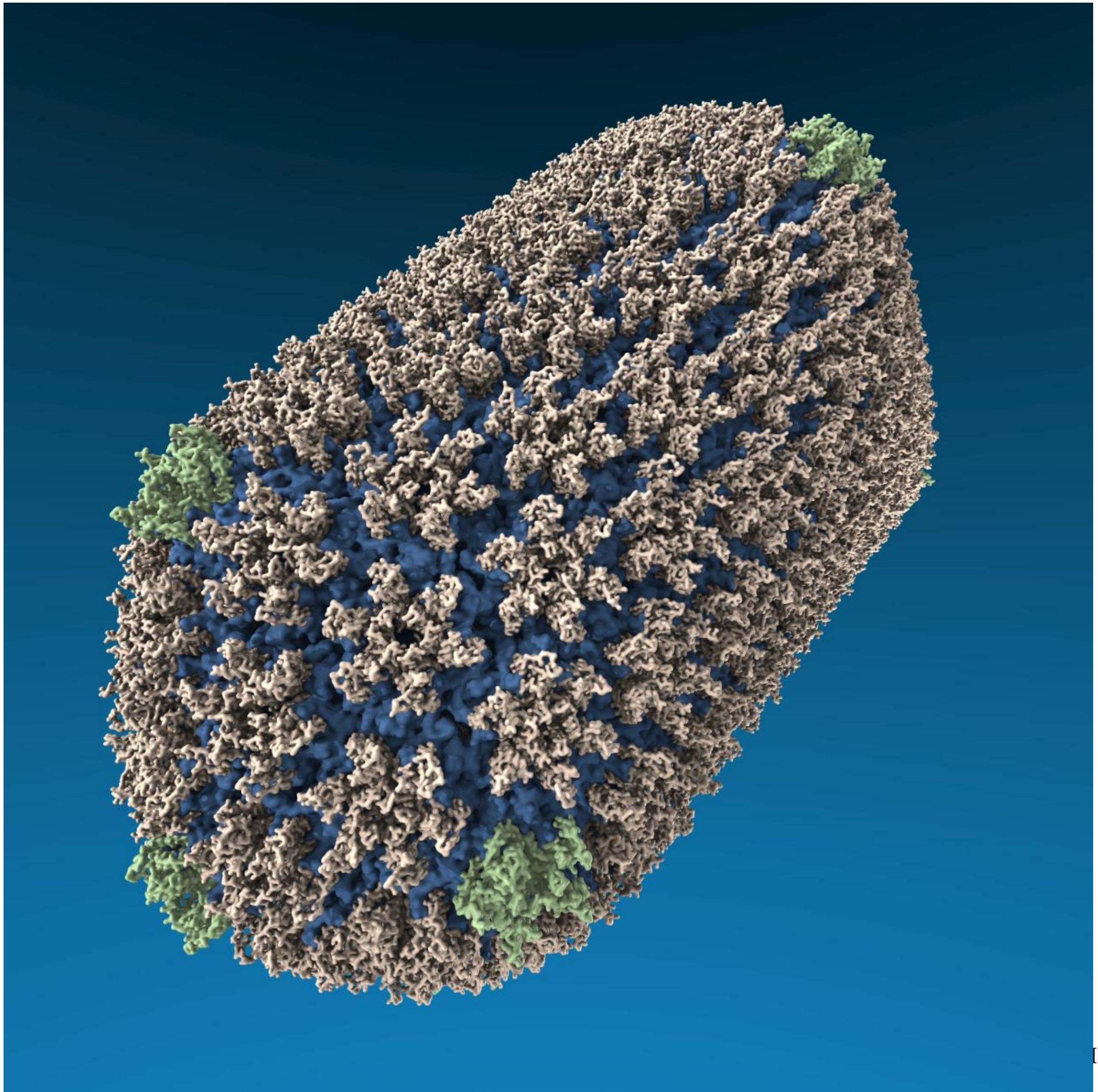
# Example Templated Density Map Kernel

```
template<class DENSITY, class VOLTEX>
__global__ static void
gaussdensity_fast_tex_norm( …  ) {

  … Triple-nested and unrolled inner loops here …

  DENSITY densityout;
  VOLTEX texout;
  convert_density(densityout, densityval1);
  densitygrid[outaddr          ] = densityout;
  convert_color(texout, densitycol1);
  voltexmap[outaddr          ] = texout;
```

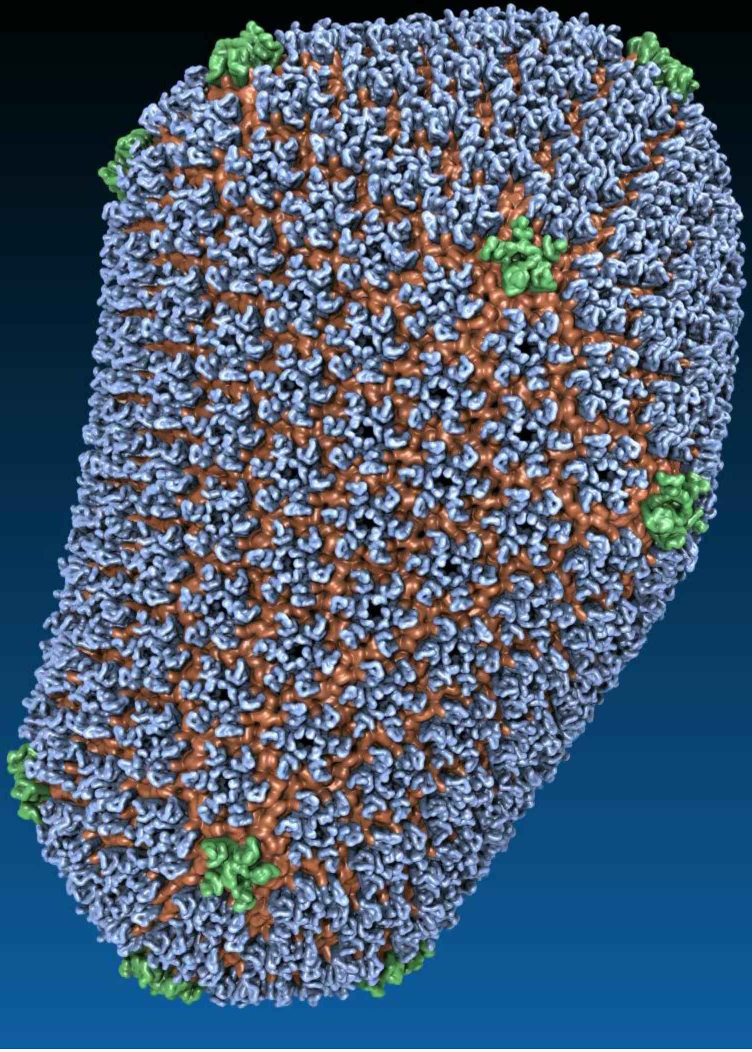
# Net Result of QuickSurf Memory Efficiency Optimizations

- **Halved** overall GPU memory use
- Achieved **1.5x to 2x performance gain**:
  - The “gather” density map algorithm keeps type conversion operations out of the innermost loop
  - Density map global memory writes reduced to half
  - Multiple stages of Marching Cubes operate on smaller input and output data types
  - Same code path supports multiple precisions
- Users now get full GPU-accelerated QuickSurf in many cases that previously triggered CPU-fallback, all platforms (laptop/desk/super) benefit!

# High Resolution HIV Surface



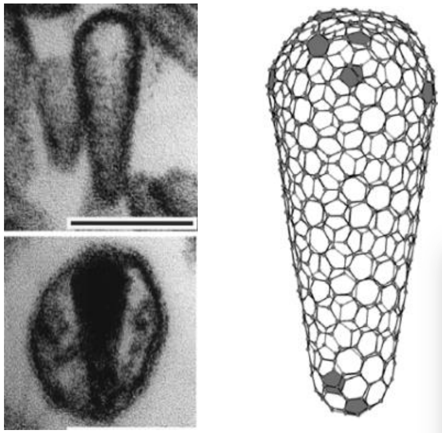
Beckman Institute,
llinois at Urbana-Champaign

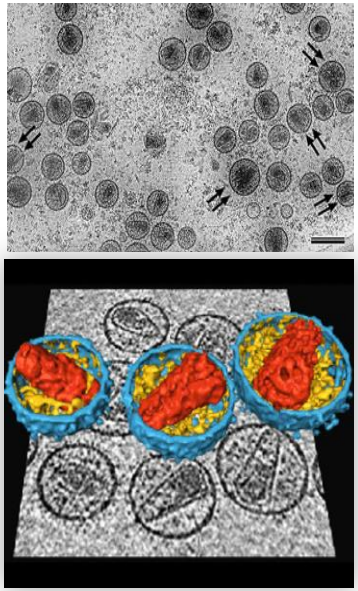


# Structural Route to the all-atom HIV-1 Capsid

1st TEM (1999)

1st tomography (2003)





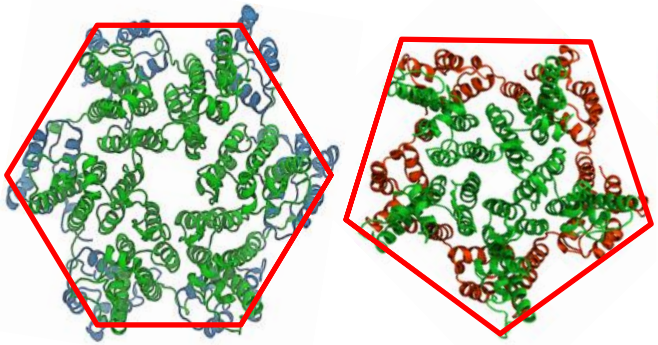
Ganser et al. *Science*, 1999

Briggs et al. *EMBO J*, 2003

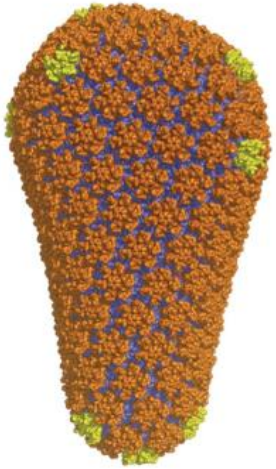
Briggs et al. *Structure*, 2006

cryo-ET (2006)

Crystal structures of separated hexamer and pentamer

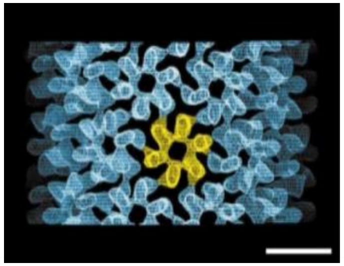


Pornillos et al. , *Cell* 2009, *Nature* 2011

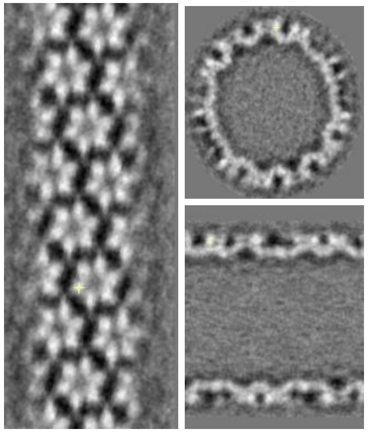


High res. EM of hexameric tubule, tomography of capsid,
**all-atom model of capsid by MDFF w/ NAMD & VMD,**
**NSF/NCSA Blue Waters computer at Illinois**

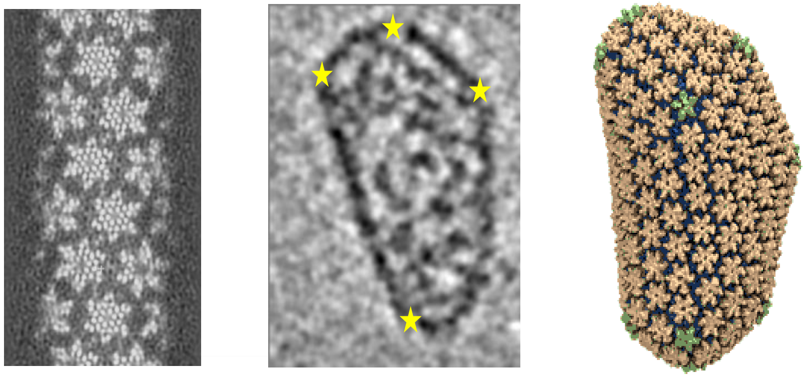
hexameric tubule



Li et al., *Nature*, 2000

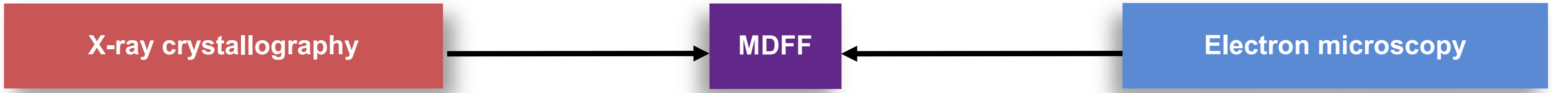


Byeon et al., *Cell* 2009



**Zhao et al. , *Nature* 497: 643-646 (2013)**

# Molecular Dynamics Flexible Fitting (MDFF)



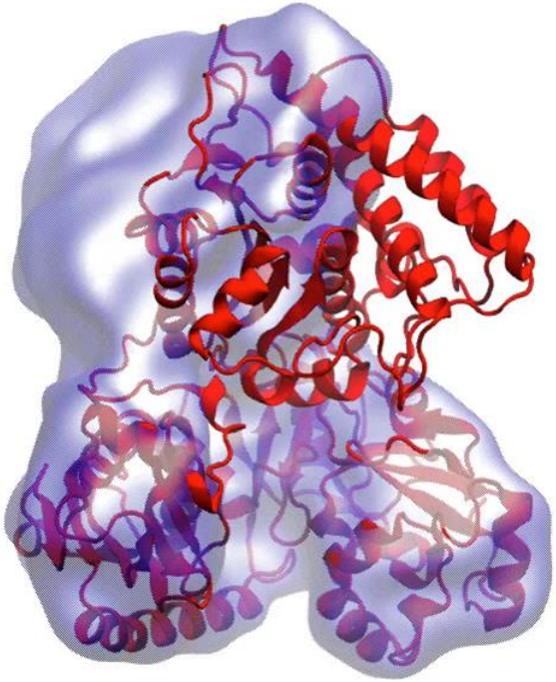
X-ray crystallography

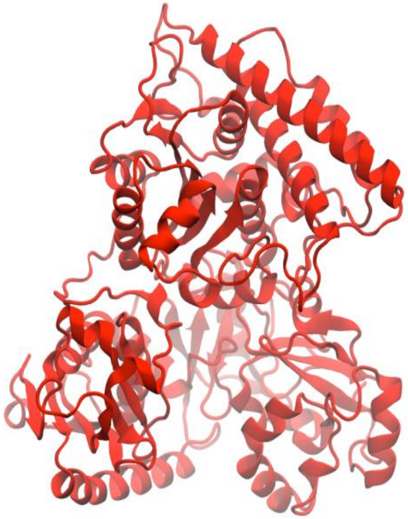
MDFF

Electron microscopy

APS at Argonne

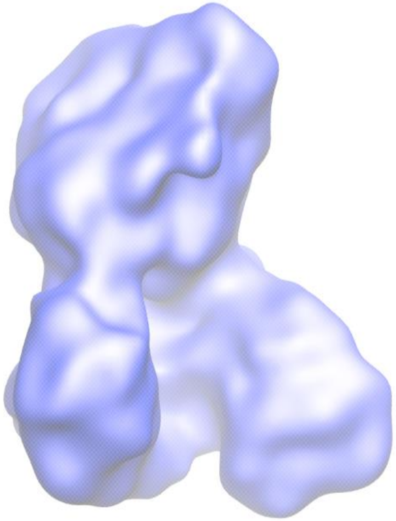
FEI microscope





Acetyl - CoA Synthase

ORNL Titan

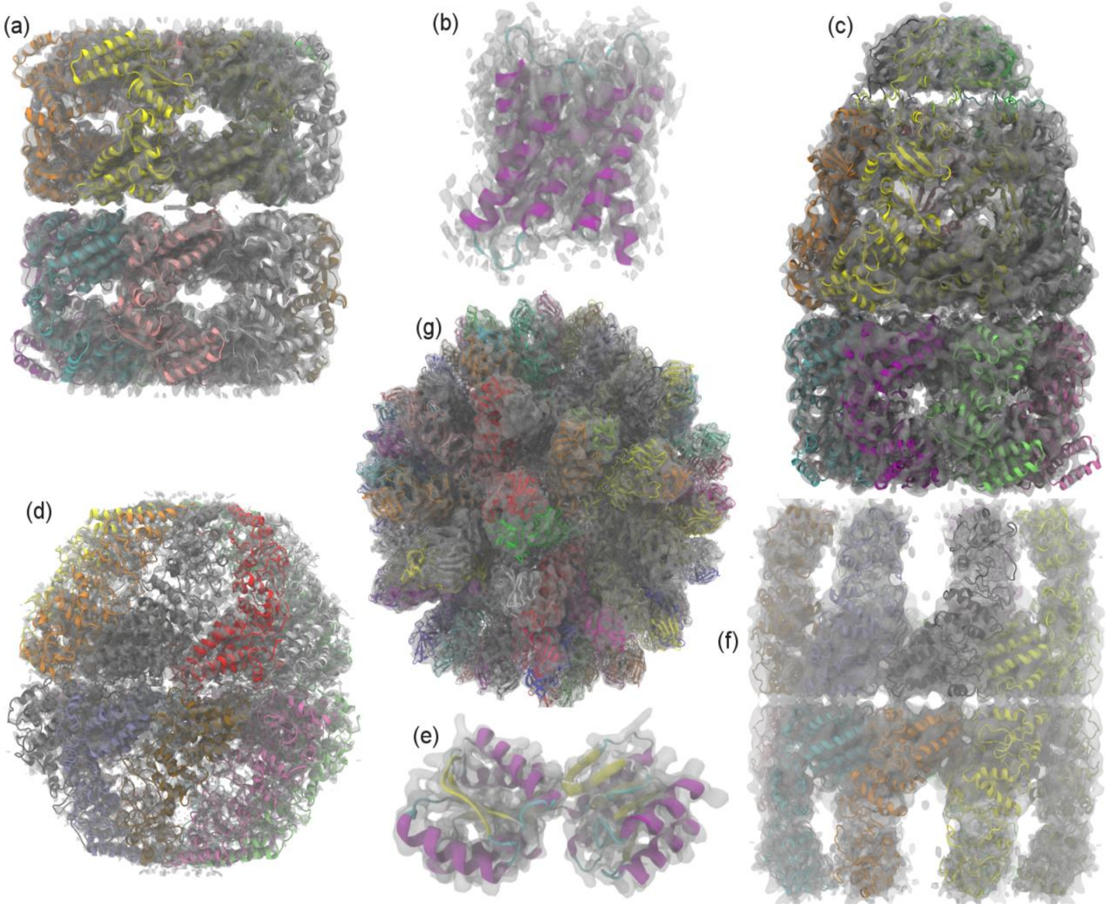


Flexible fitting of atomic structures into electron microscopy maps using molecular dynamics.

L. Trabuco, E. Villa, K. Mitra, J. Frank, and K. Schulten. Structure, 16:673-683, 2008.

# Evaluating Quality-of-Fit for Structures Solved by Hybrid Fitting Methods

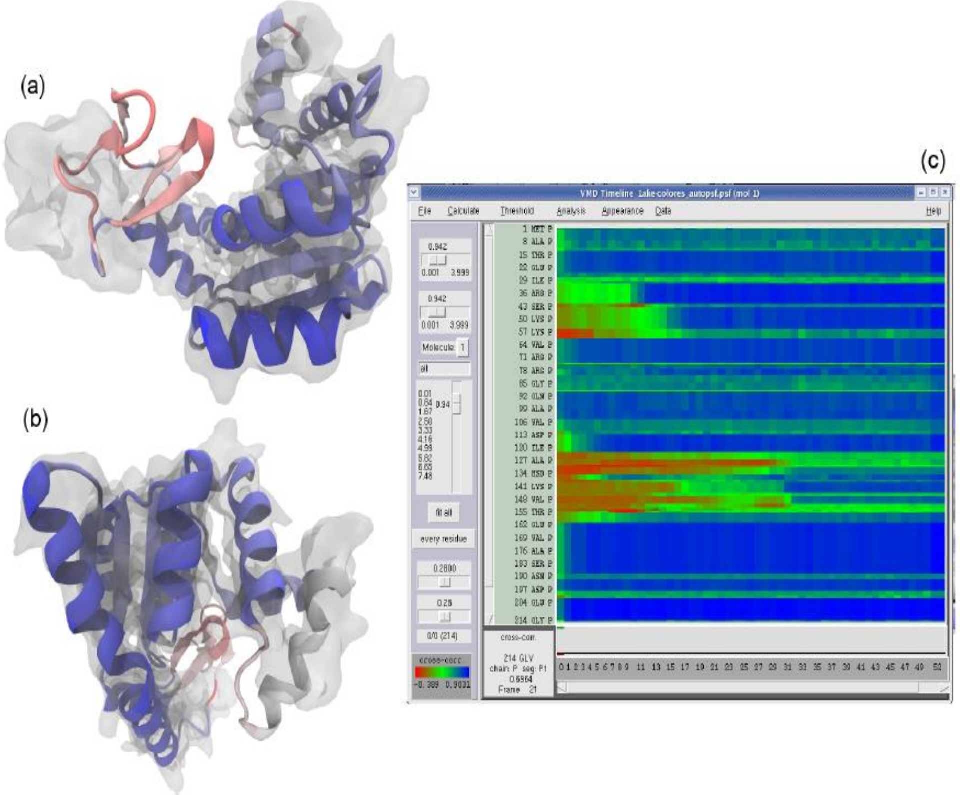
Compute Pearson correlation to evaluate the fit of a reference cryo-EM density map with a simulated density map produced from an all-atom structure.



# GPUs Can Reduce Trajectory Analysis Runtimes from Hours to Minutes

GPUs enable laptops and desktop workstations to handle tasks that would have previously required a cluster, or a very long wait…

GPU-accelerated petascale supercomputers enable analyses were previously impossible, allowing detailed study of very large structures such as viruses



**GPU-accelerated MDFF Cross Correlation Timeline**

**Regions with poor fit** **Regions with good fit**

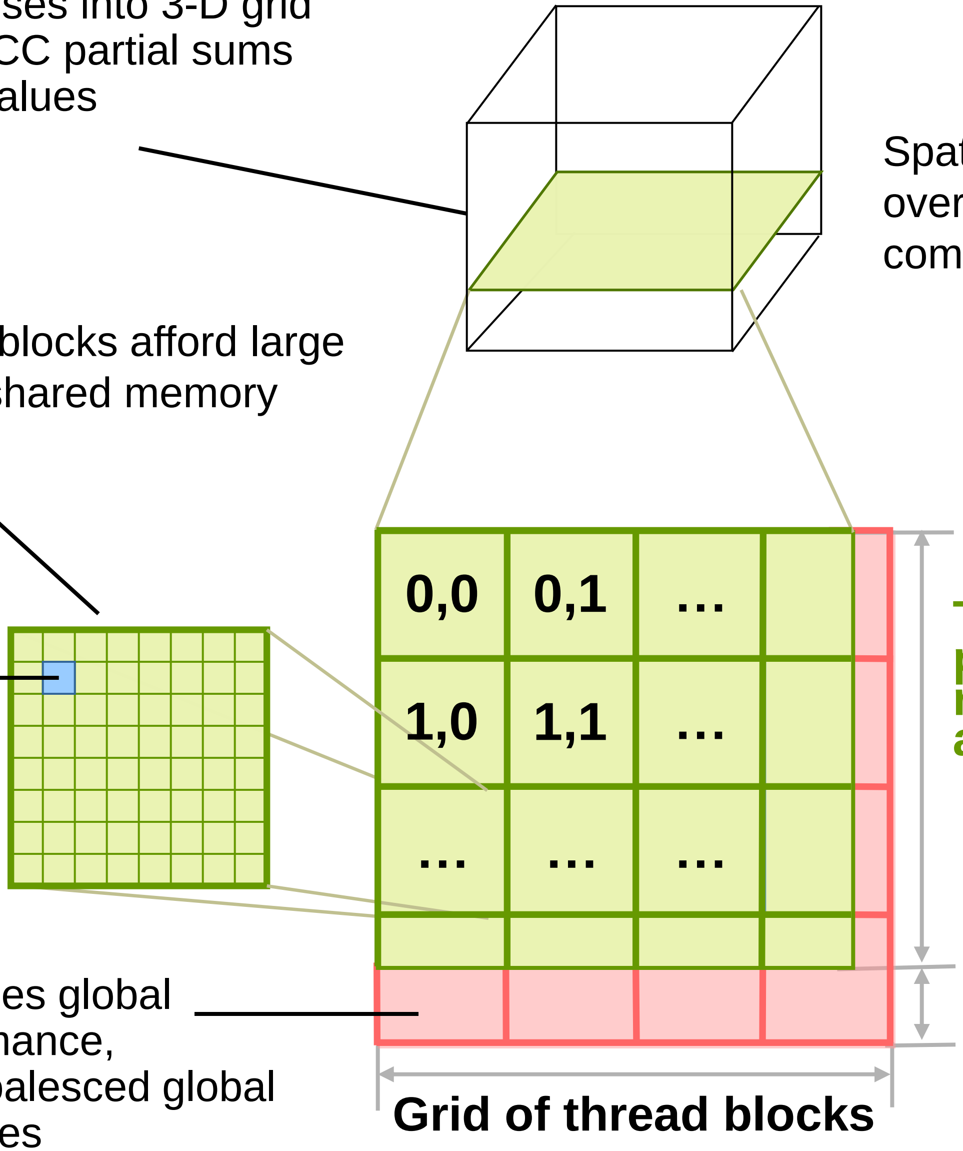
# Single-Pass MDFF GPU Cross-Correlation

3-D density map decomposes into 3-D grid of 8x8x8 tiles containing CC partial sums and local CC values

Spatial CC map and overall CC value computed in a single pass

Small 8x8x2 CUDA thread blocks afford large per-thread register count, shared memory

Each thread computes 4 z-axis density map lattice points and associated CC partial sums



Threads producing results that are used

Inactive threads, region of discarded output

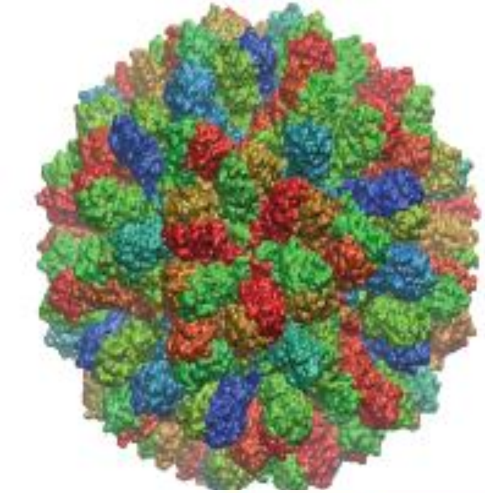
Padding optimizes global memory performance, guaranteeing coalesced global memory accesses

# VMD GPU Cross Correlation Performance

| | RHDV | Mm-cpn<br>open | GroEL | Aquaporin |
|---|---|---|---|---|
| Resolution (Å) | 6.5 | 8 | 4 | 3 |
| Atoms | 702K | 61K | 54K | 1.6K |
| VMD-CUDA<br>Quadro K6000 | 0.458s<br>34.6x | 0.06s<br>25.7x | 0.034s<br>36.8x | 0.007s<br>55.7x |
| VMD-CPU-SSE<br>32-threads, 2x Xeon E5-2687W | 0.779s<br>20.3x | 0.085s<br>18.1x | 0.159s<br>7.9x | 0.033s<br>11.8x |
| Chimera<br>1-thread Xeon E5-2687W | 15.86s<br>1.0x | 1.54s<br>1.0x | 1.25s<br>1.0x | 0.39s<br>1.0x |

**GPU-accelerated analysis and visualization of large structures solved by molecular dynamics flexible fitting.** J. E. Stone, R. McGreevy, B. Isralewitz, and K. Schulten. Faraday Discussion 169, 2014. (In press).

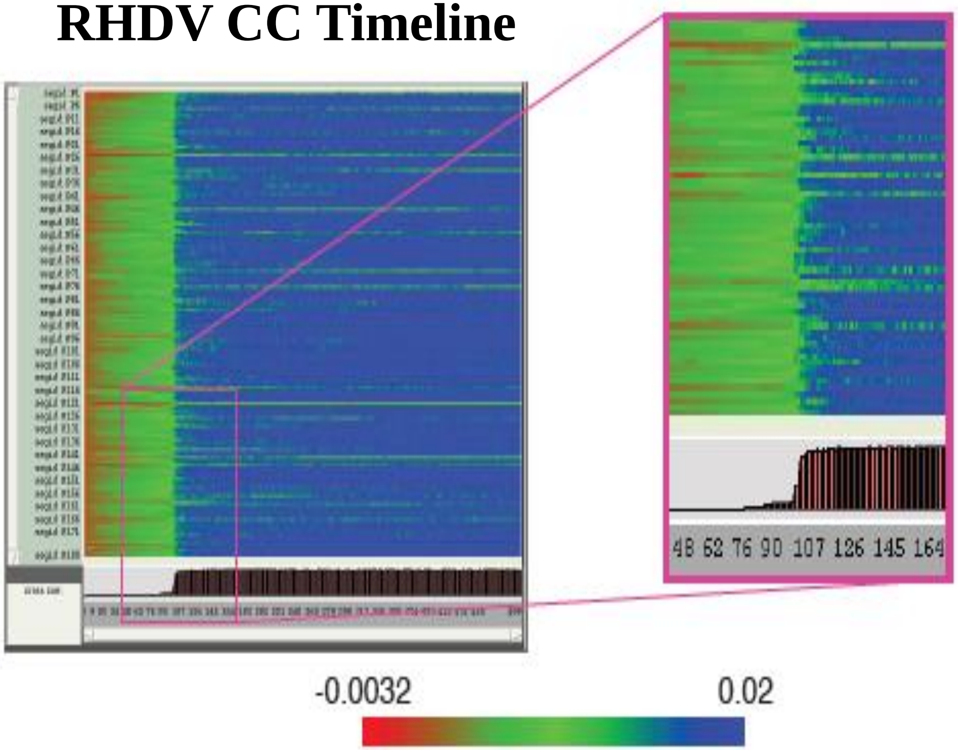
# VMD RHDV Cross Correlation Timeline on Cray XK7



| | RHDV |
|---|---|
| Atoms | 702K |
| Component Selections | 720 |
| Single-node XK7 (projected) | 336 hours (14 days) |
| 128-node XK7 | 3.2 hours<br>105x speedup |

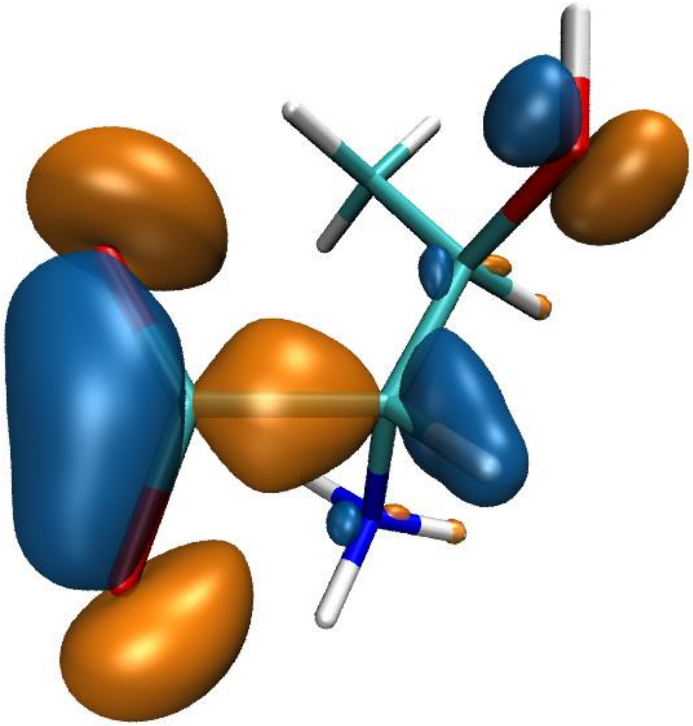
Calculation would take **5 years** using conventional non-GPU software on a workstation!!

**RHDV CC Timeline**



# Molecular Orbitals

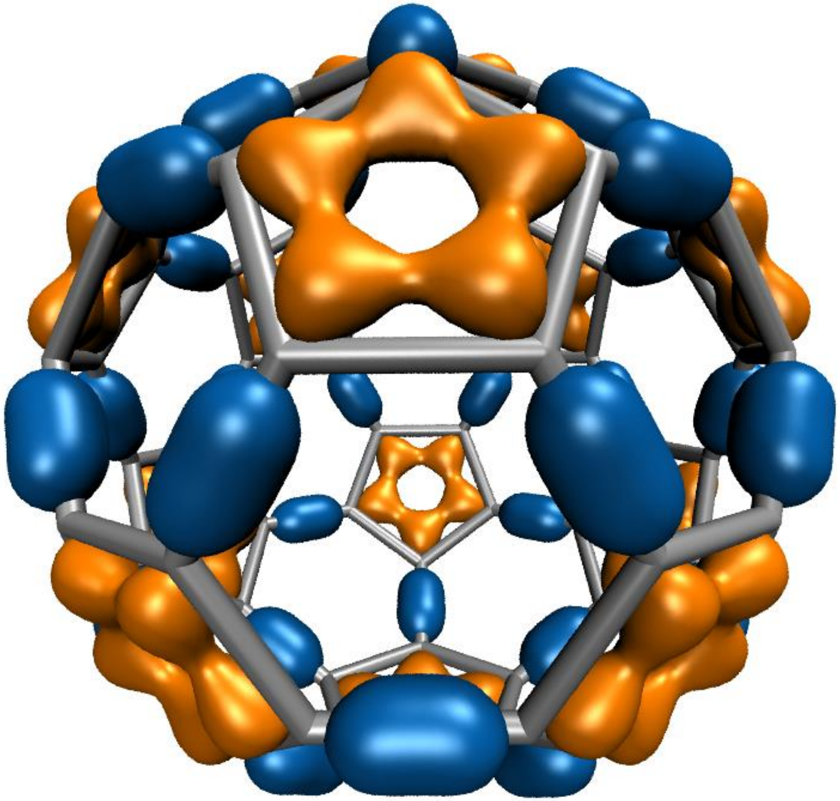
- Visualization of MOs aids in understanding the chemistry of molecular system
- MO spatial distribution is correlated with probability density for an electron(s)
- Algorithms for computing other molecular properties are similar, and can share code



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 Pricessing Units (GPGPU-2), ACM International Conference Proceeding Series*, volume 383, pp. 9-18, 2009.

# Computing Molecular Orbitals

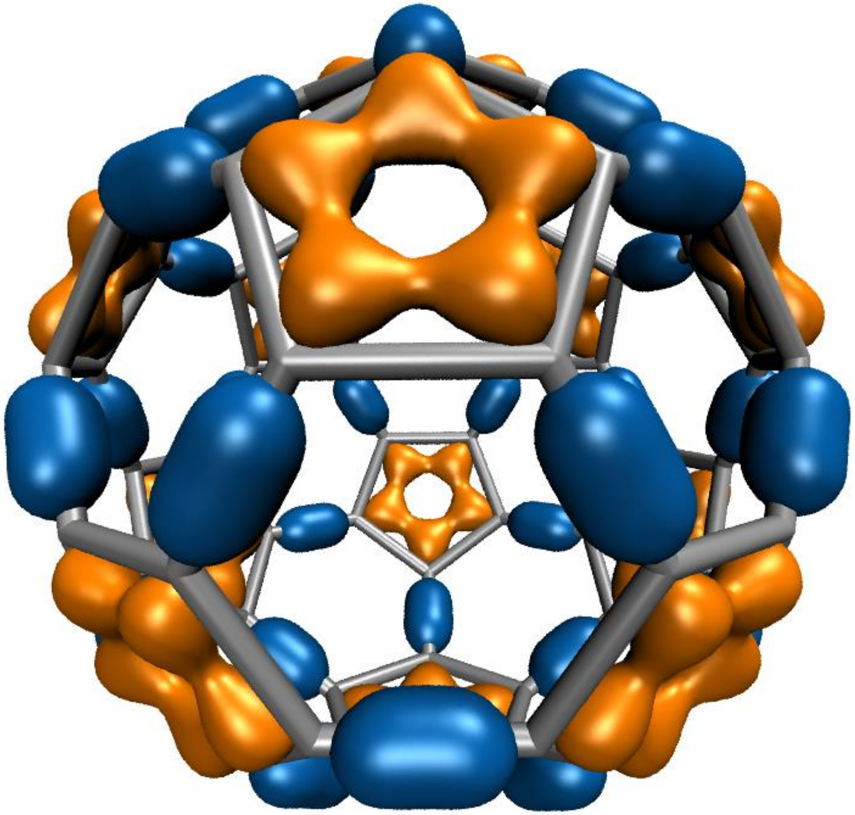
- Calculation of high resolution MO grids can require tens to hundreds of seconds in existing tools
- Existing tools cache MO grids as much as possible to avoid recomputation:
  - Doesn’t eliminate the wait for initial calculation, hampers interactivity
  - Cached grids consume 100x-1000x more memory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



$C_{60}$

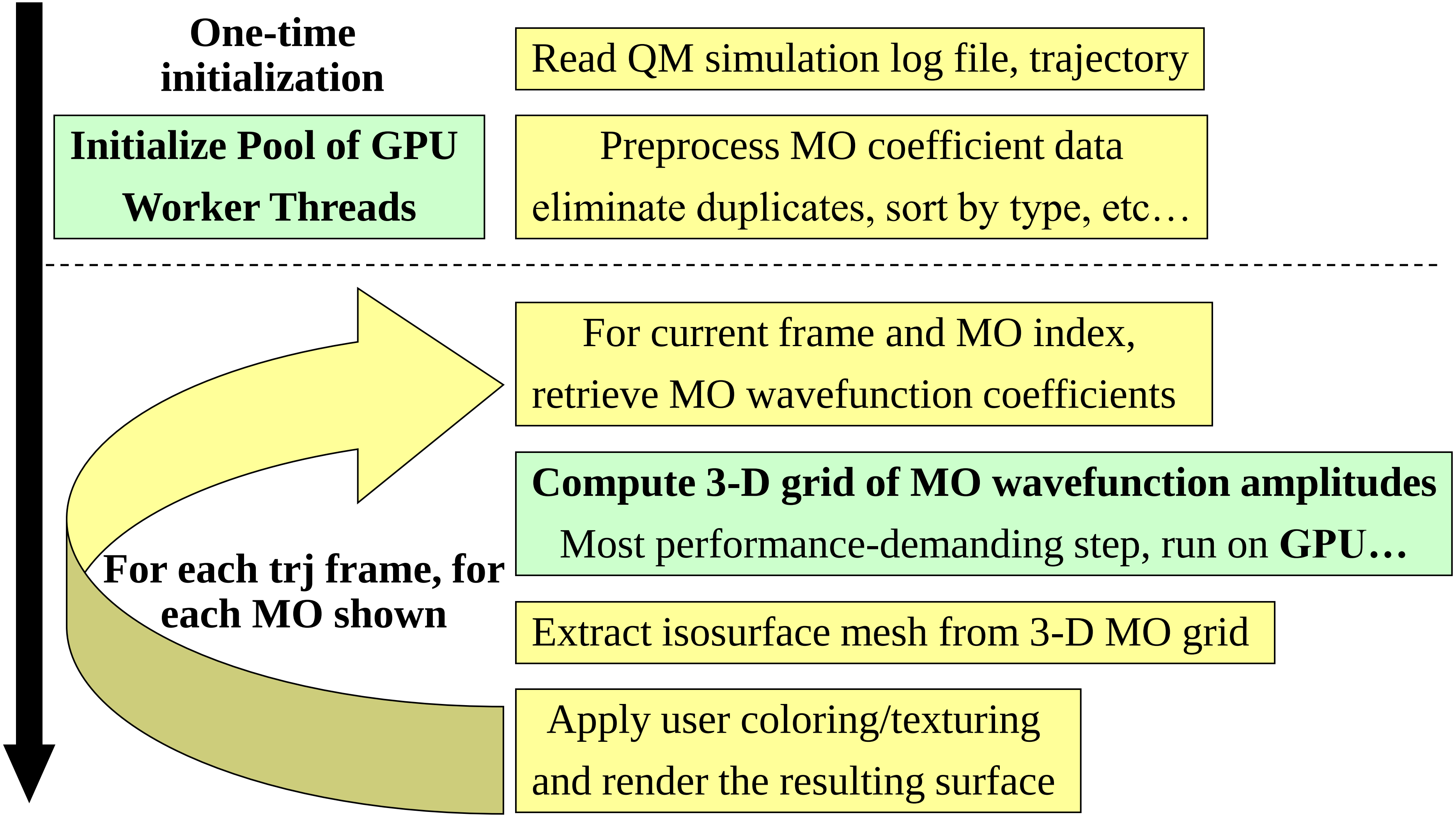
# Animating Molecular Orbitals

- 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

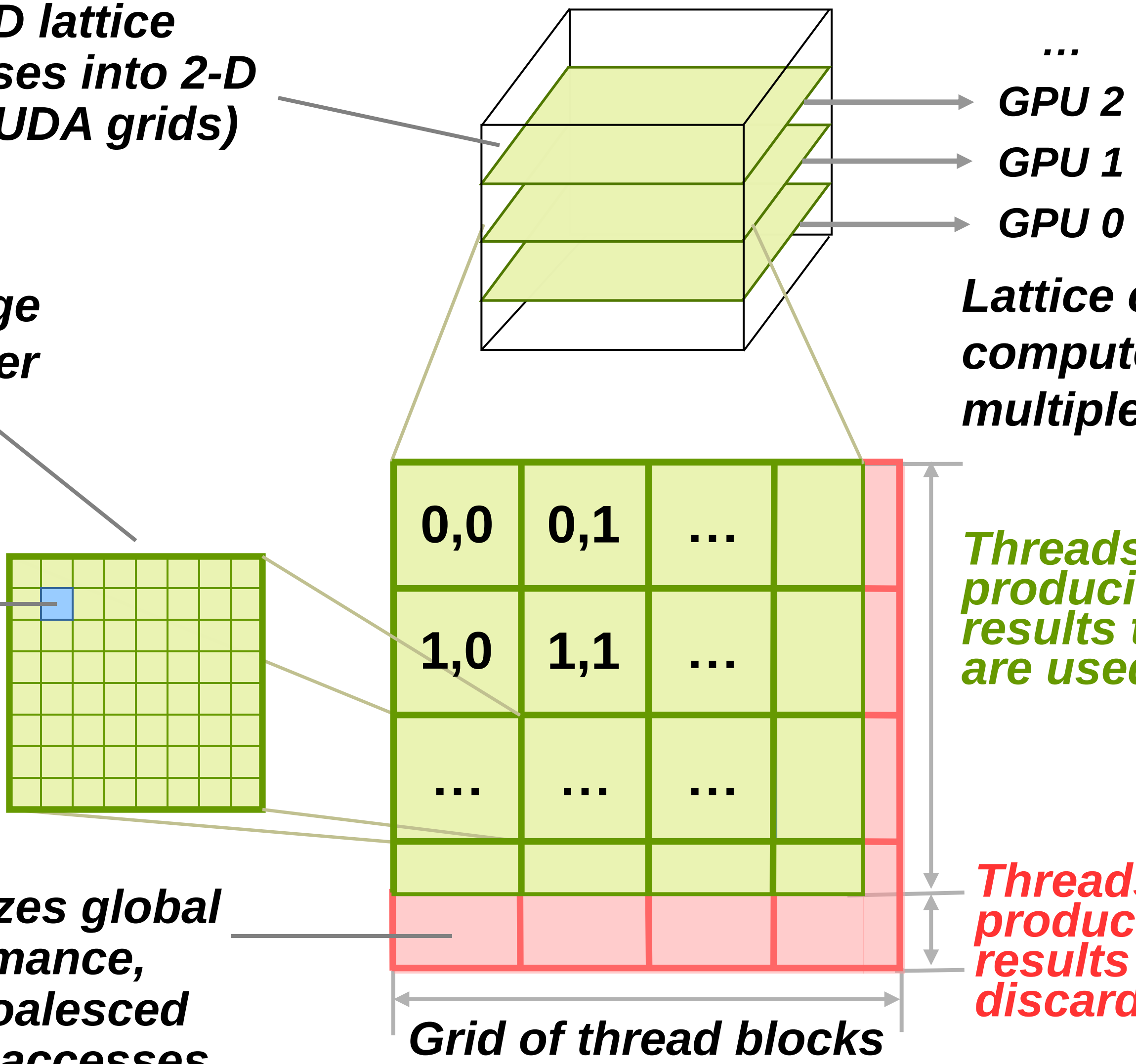
and render the resulting surface

# MO GPU Parallel Decomposition

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

***Small 8x8 thread blocks afford large per-thread register count, shared memory***

***Each thread computes one MO lattice point.***



***Lattice can be computed using multiple GPUs***

***Threads producing results that are used***

***Threads producing results that are discarded***

***Padding optimizes global memory performance, guaranteeing coalesced global memory accesses***

# MO GPU Kernel Snippet:

## Contracted GTO Loop, Use of Constant Memory

```
[… outer loop over atoms …]
  float dist2 = xdist2 + ydist2 + zdist2;
  // Loop over the shells belonging to this atom (or basis function)
  for (shell=0; shell < maxshell; shell++) {
    float contracted_gto = 0.0f;
    // Loop over Gaussian primitives of this contracted basis function to build the atomic orbital
    int maxprim = const_num_prim_per_shell[shell_counter];
    int shelltype = const_shell_types[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 * __expf(-exponent*dist2);
      prim_counter += 2;
    }
[… continue on to angular momenta loop …]
```

Constant memory: nearly register-speed when array elements accessed in unison by all threads….

# MO GPU Kernel Snippet:

## Unrolled Angular Momenta Loop

```
    /* multiply with the appropriate wavefunction coefficient */
    float tmpshell=0;
    switch (shelltype) {
     case S_SHELL:
       value += const_wave_f[ifunc++] * contracted_gto;
       break;
[… P_SHELL case …]
     case D_SHELL:
       tmpshell += const_wave_f[ifunc++] * xdist2;
       tmpshell += const_wave_f[ifunc++] * xdist * ydist;
       tmpshell += const_wave_f[ifunc++] * ydist2;
       tmpshell += const_wave_f[ifunc++] * xdist * zdist;
       tmpshell += const_wave_f[ifunc++] * ydist * zdist;
       tmpshell += const_wave_f[ifunc++] * zdist2;
       value += tmpshell * contracted_gto;
       break;
[... Other cases: F_SHELL, G_SHELL, etc …]
} // end switch
```

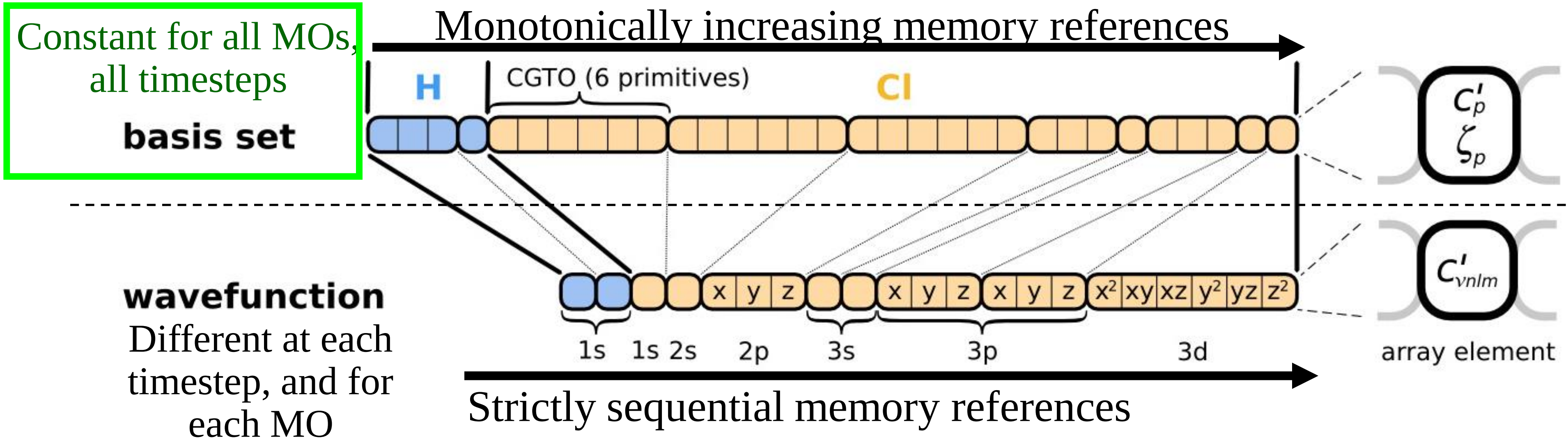
Loop unrolling:

- Saves registers (important for GPUs!)
- Reduces loop control overhead
- Increases arithmetic intensity

# Preprocessing of Atoms, Basis Set, and Wavefunction Coefficients

- Must make effective use of high bandwidth, low-latency GPU on-chip shared memory, or L1 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

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



Constant for all MOs, all timesteps

**basis set**

Monotonically increasing memory references

**wavefunction**

Different at each timestep, and for each MO

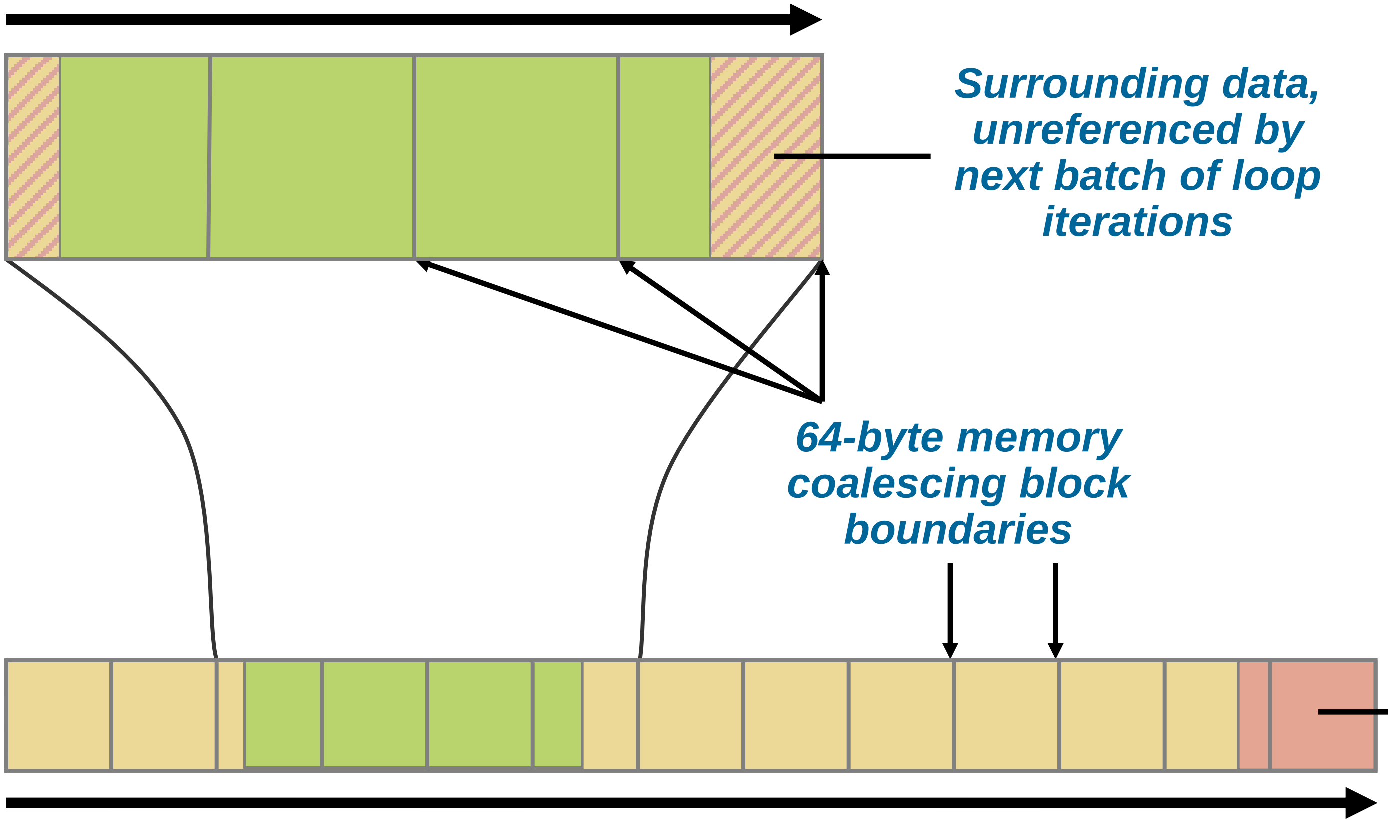
Strictly sequential memory references

- Loop iterations always access same or consecutive array elements for all threads in a thread block:
  - Yields good constant memory and L1 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:
  - Tiles sized large enough to service entire inner loop runs, broadcast to all 64 threads in a block
  - Complications: nested loops, multiple arrays, varying length
  - Key to performance is to locate tile loading checks outside of the two performance-critical inner loops
  - Only 27% slower than hardware caching provided by constant memory (on GT200)
- Fermi/Kepler GPUs have larger on-chip shared memory, L1/L2 caches, greatly reducing control overhead

***Array tile loaded in GPU shared memory.*** Tile size is a power-of-two, a multiple of coalescing size, and allows simple indexing in inner loops. Global memory array indices are merely offset to reference an MO coefficient within a tile loaded in fast on-chip shared memory.



*Surrounding data, unreferenced by next batch of loop iterations*

*64-byte memory coalescing block boundaries*

*Full tile padding*

***MO coefficient array in GPU global memory. Tiles are referenced in consecutive order.***

# VMD MO GPU Kernel Snippet:

## Loading Tiles Into Shared Memory On-Demand

```
[… outer loop over atoms …]
    if ((prim_counter + (maxprim<<1)) >= SHAREDSIZE) {
      prim_counter += sblock_prim_counter;
      sblock_prim_counter = prim_counter & MEMCOAMASK;
      s_basis_array[sidx      ] = basis_array[sblock_prim_counter + sidx      ];
      s_basis_array[sidx +  64] = basis_array[sblock_prim_counter + sidx +  64];
      s_basis_array[sidx + 128] = basis_array[sblock_prim_counter + sidx + 128];
      s_basis_array[sidx + 192] = basis_array[sblock_prim_counter + sidx + 192];
      prim_counter -= sblock_prim_counter;
      __syncthreads();
    }
    for (prim=0; prim < maxprim;  prim++) {
      float exponent       = s_basis_array[prim_counter     ];
      float contract_coeff = s_basis_array[prim_counter + 1];
      contracted_gto += contract_coeff * __expf(-exponent*dist2);
      prim_counter += 2;
    }
[… continue on to angular momenta loop …]
```

Shared memory tiles:

- Tiles are checked and loaded, if necessary, immediately prior to entering key arithmetic loops
- Adds additional control overhead to loops, even with optimized implementation

# New GPUs Bring Opportunities for Higher Performance and Easier Programming

- NVIDIA’s Fermi, Kepler, Maxwell GPUs bring:
  - **Greatly increased** peak single- and double-precision arithmetic rates
  - **Moderately** increased global memory bandwidth
  - Increased capacity on-chip memory partitioned into shared memory and an L1 cache for global memory
  - Concurrent kernel execution
  - Bidirectional asynchronous host-device I/O
  - ECC memory, faster atomic ops, many others…

# VMD MO GPU Kernel Snippet:

## Kernel based on L1 cache  (Fermi) or Read-only Data Cache (Maxwell)

```
[… outer loop over atoms …]
 // loop over the shells/basis funcs belonging to this atom
 for (shell=0; shell < maxshell; shell++) {
   float contracted_gto = 0.0f;
   int maxprim    = shellinfo[(shell_counter<<4)      ];
   int shell_type = shellinfo[(shell_counter<<4) + 1];
   for (prim=0; prim < maxprim; 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;
  }
  [… continue on to angular momenta loop …]
```

L1 cache:

- Simplifies code!
- Reduces control overhead
- Gracefully handles arbitrary-sized problems
- Matches performance of constant memory on Fermi and Maxwell

# MO Kernel for One Grid Point (Naive C)

```
…
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++) {
      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;
      }
      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;
      }
      value += tmpshell * contracted_gto;
      shell_counter++;
   }
} …..
```

Loop over atoms

Loop over shells

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

Loop over angular momenta

(unrolled in real code)

# 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. |

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

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

# VMD Single-GPU Molecular Orbital Performance Results for $C_{60}$ on Fermi

## Intel X5550 CPU, GeForce GTX 480 GPU

| Kernel | Cores/GPUs | Runtime (s) | Speedup |
|---|---|---|---|
| Xeon 5550 ICC-SSE | 1 | 30.64 | 1.0 |
| Xeon 5550 ICC-SSE | 8 | 4.13 | 7.4 |
| CUDA shared mem | 1 | 0.37 | 83 |
| **CUDA L1-cache (16KB)** | **1** | **0.27** | **113** |
| CUDA const-cache | 1 | 0.26 | 117 |
| CUDA const-cache, zero-copy | 1 | 0.25 | 122 |

Fermi GPUs have caches: match perf. of hand-coded shared memory kernels. Zero-copy memory transfers improve overlap of computation and host-GPU I/Os.

# Preliminary Single-GPU Molecular Orbital Performance Results for $C_{60}$ on Kepler

## Intel X5550 CPU, GeForce GTX 680 GPU

| Kernel | Cores/GPUs | Runtime (s) | Speedup |
|---|---|---|---|
| Xeon 5550 ICC-SSE | 1 | 30.64 | 1.0 |
| Xeon 5550 ICC-SSE | 8 | 4.13 | 7.4 |
| CUDA shared mem | 1 | 0.264 | 116 |
| CUDA RO-data-cache | 1 | 0.228 | 134 |
| CUDA const-cache | 1 | 0.104 | 292 |
| CUDA const-cache, zero-copy | 1 | 0.0938 | 326 |

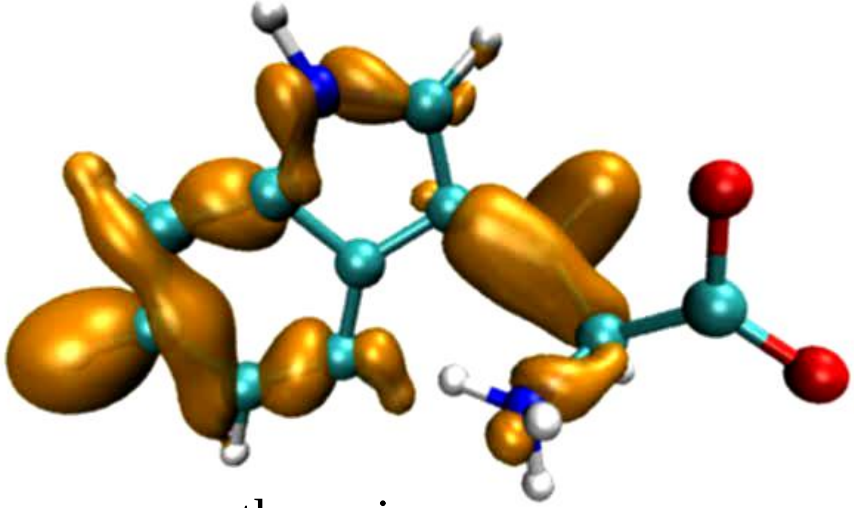
Kepler GK104 (GeForce 680) strongly prefers the constant cache kernels vs. the others.

# 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. | **0.016 s** |
| CPU surface gen, volume gradient, and GPU rendering | **0.033 s** |
| **Total runtime** | **0.049 s** |
| **Frame rate** | **20 FPS** |



threonine

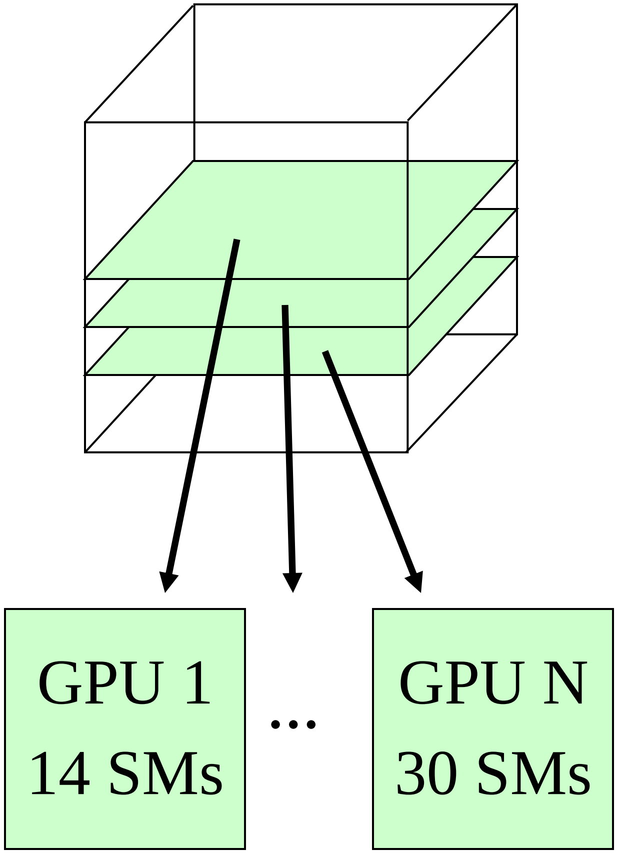
With GPU speedups over **100x**, previously insignificant CPU surface gen, gradient calc, and rendering are now **66%** of runtime.

Needed GPU-accelerated surface gen next…

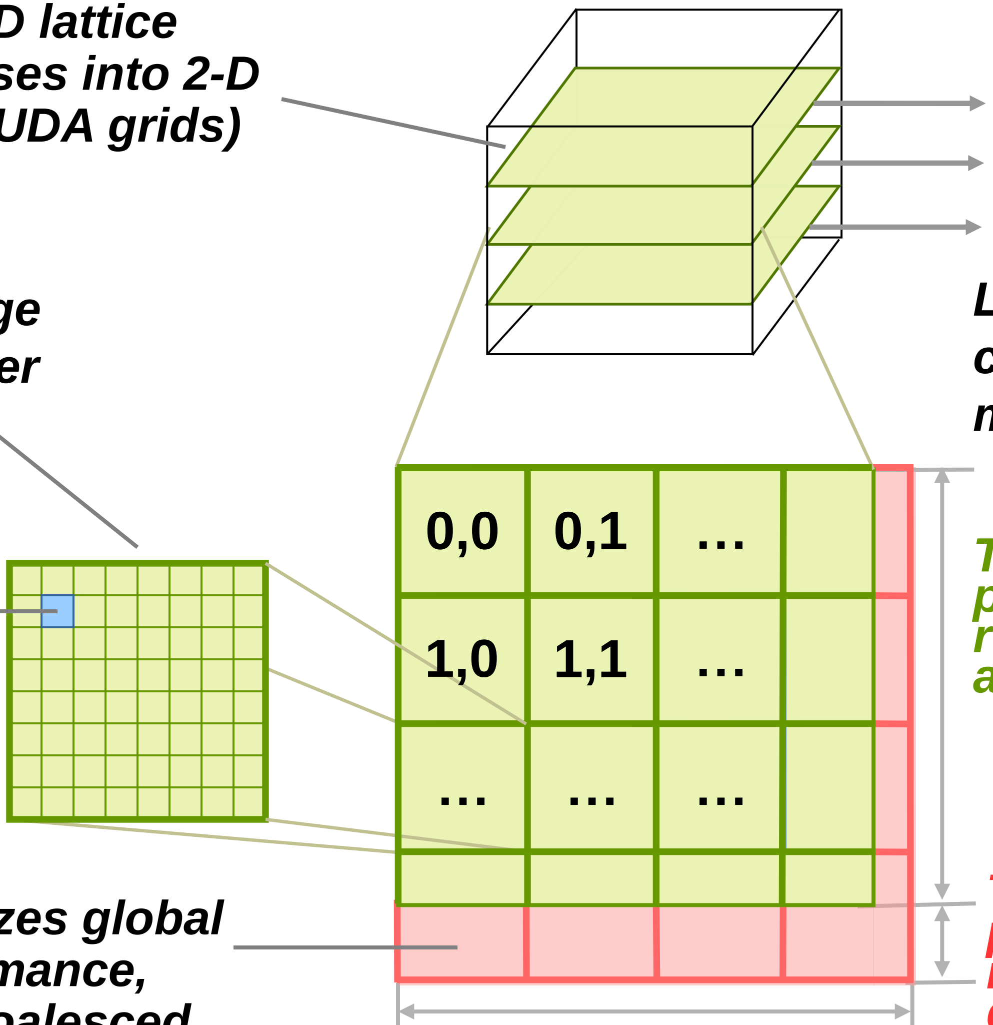
Wrote CUDA Marching Cubes to address surface gen perf gap.

# Multi-GPU Load Balance

- Many early CUDA codes assumed all GPUs were identical
- Host machines may contain a diversity of GPUs of varying capability (discrete, IGP, etc)
- Different GPU on-chip and global memory capacities may need different problem "tile" sizes
- Static decomposition works poorly for non-uniform workload, or diverse GPUs



# MO GPU Parallel Decomposition



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

***Lattice can be computed using multiple GPUs***

***Small 8x8 thread blocks afford large per-thread register count, shared memory***

***Each thread computes one MO lattice point.***

***Threads producing results that are used***

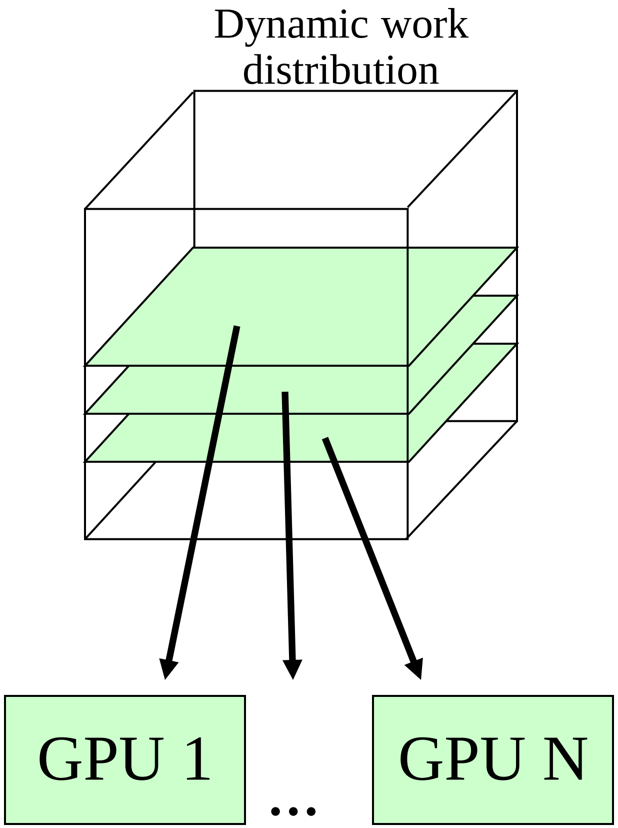
***Threads producing results that are discarded***

***Padding optimizes global memory performance, guaranteeing coalesced global memory accesses***

***Grid of thread blocks***

# Multi-GPU Dynamic Work Distribution

```
// Each GPU worker thread loops over
// subset of work items…
while (!threadpool_next_tile(&parms,
    tilesize, &tile){
  // Process one work item…
  // Launch one CUDA kernel for each
  //   loop iteration taken…
  // Shared iterator automatically
  //   balances load on GPUs
}
```



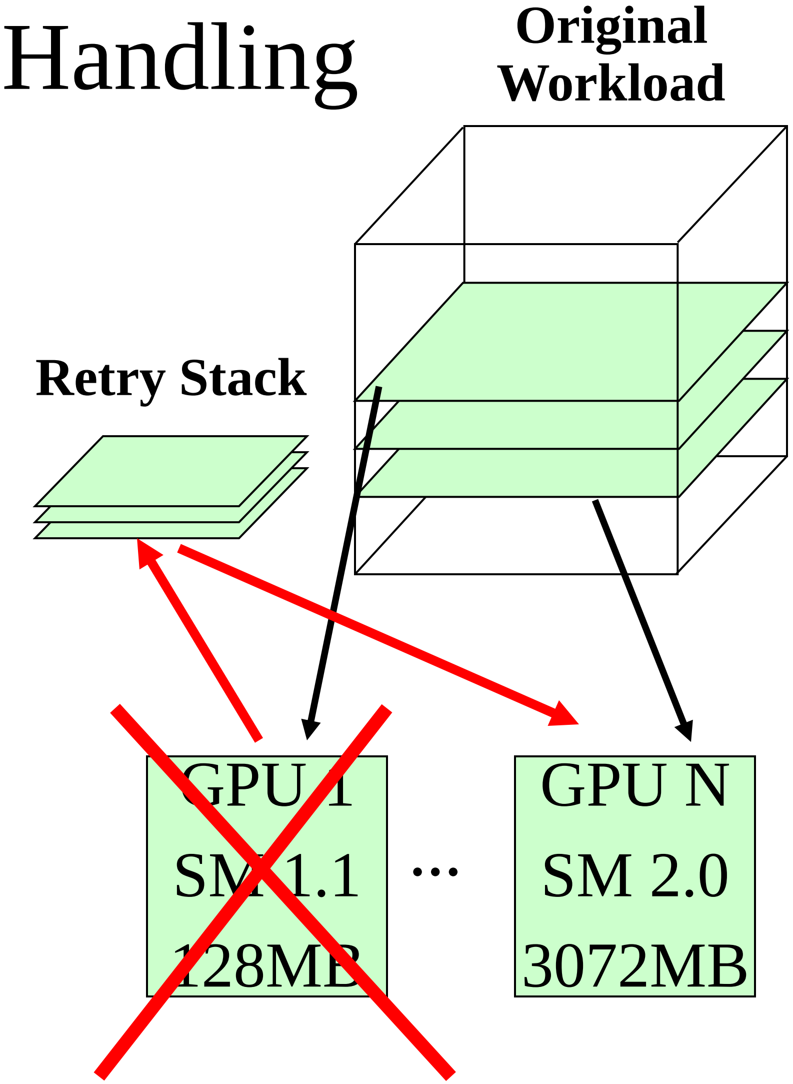
# Example Multi-GPU Latencies

## 4 C2050 GPUs, Intel Xeon 5550

| | |
|---|---|
| 6.3us | CUDA empty kernel (immediate return) |
| 9.0us | Sleeping barrier primitive (non-spinning barrier that uses POSIX condition variables to prevent idle CPU consumption while workers wait at the barrier) |
| 14.8us | pool wake, host fctn exec, sleep cycle (no CUDA) |
| 30.6us | pool wake, 1x(tile fetch, simple CUDA kernel launch), sleep |
| 1817.0us | pool wake, 100x(tile fetch, simple CUDA kernel launch), sleep |

# Multi-GPU Runtime Error/Exception Handling

- Competition for resources from other applications can cause runtime failures, e.g. GPU out of memory half way through an algorithm
- Handle exceptions, e.g. convergence failure, NaN result, insufficient compute capability/features
- Handle and/or reschedule failed tiles of work



# VMD Multi-GPU Molecular Orbital Performance Results for $C_{60}$

| 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

# Performance Evaluation:

## Molekel, MacMolPlt, and VMD

## Sun Ultra 24: Intel Q6600, NVIDIA GTX 280

| | **$C_{60}$-A** | **$C_{60}$-B** | **Thr-A** | **Thr-B** | **Kr-A** | **Kr-B** |
|---|---|---|---|---|---|---|
| Atoms | 60 | 60 | 17 | 17 | 1 | 1 |
| Basis funcs (unique) | **300 (5)** | **900 (15)** | **49 (16)** | **170 (59)** | **19 (19)** | **84 (84)** |

| Kernel | **Cores**<br>**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 Multi-GPU Molecular Orbital Performance Results for $C_{60}$

## Intel X5550 CPU, 4x GeForce GTX 480 GPUs,

| Kernel | Cores/GPUs | Runtime (s) | Speedup |
|---|---|---|---|
| Intel X5550-SSE | 1 | 30.64 | 1.0 |
| Intel X5550-SSE | 8 | 4.13 | 7.4 |
| GeForce GTX 480 | 1 | 0.255 | 120 |
| GeForce GTX 480 | 2 | 0.136 | 225 |
| GeForce GTX 480 | 3 | 0.098 | 312 |
| GeForce GTX 480 | 4 | 0.081 | 378 |

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

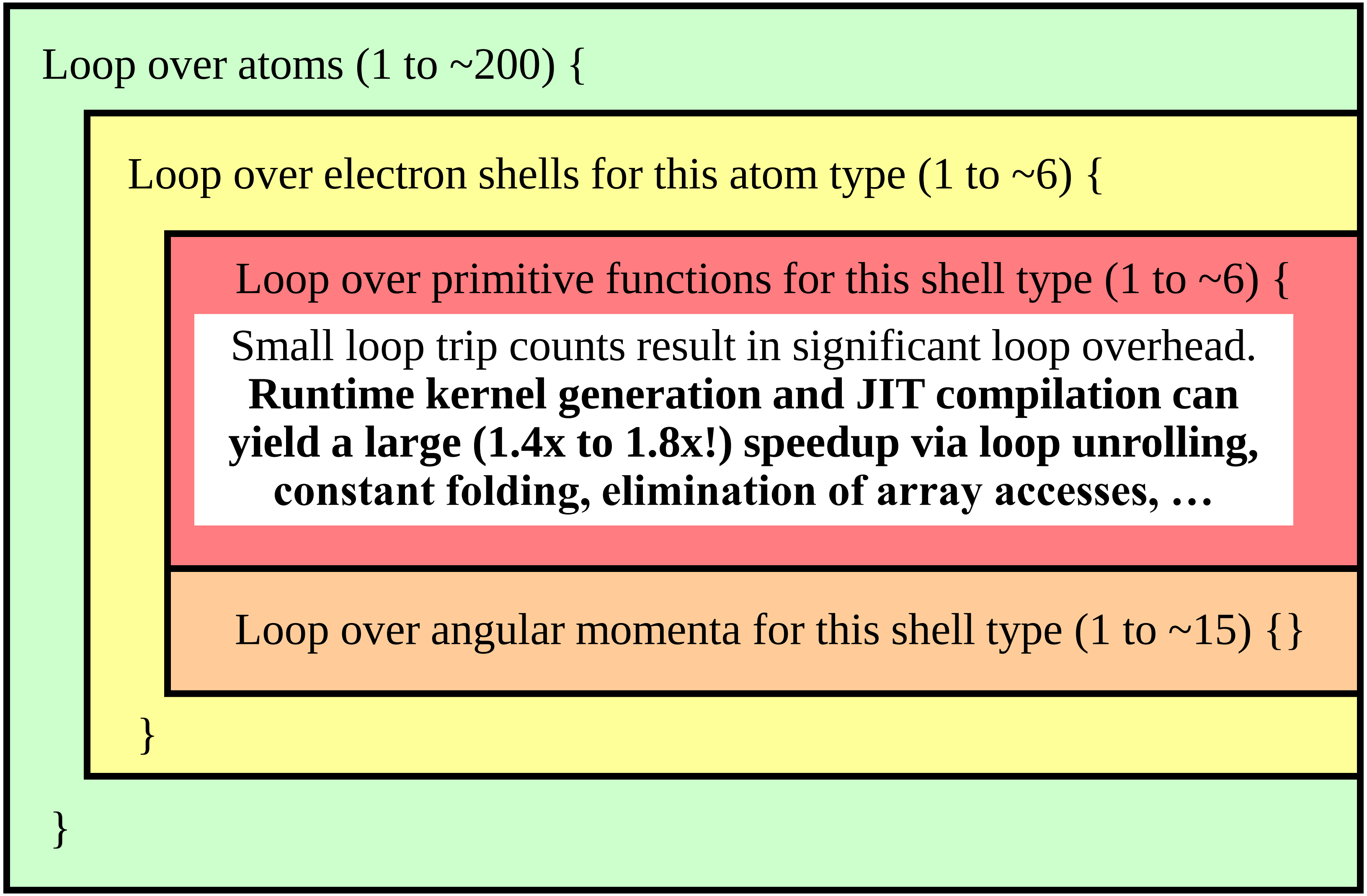
# Molecular Orbital Dynamic Scheduling Performance with Heterogeneous GPUs

| Kernel | Cores/GPUs | Runtime (s) | Speedup |
|---|---|---|---|
| Intel X5550-SSE | 1 | 30.64 | 1.0 |
| Quadro 5800 | 1 | 0.384 | 79 |
| Tesla C2050 | 1 | 0.325 | 94 |
| GeForce GTX 480 | 1 | 0.255 | 120 |
| GeForce GTX 480 +<br>Tesla C2050 +<br>Quadro 5800 | 3 | 0.114 | 268<br>(91% of ideal perf) |

Dynamic load balancing enables mixture of GPU generations, SM counts, and clock rates to perform well.

# 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) {

Small loop trip counts result in significant loop overhead. **Runtime kernel generation and JIT compilation can yield a large (1.4x to 1.8x!) speedup via loop unrolling, constant folding, elimination of array accesses, …**

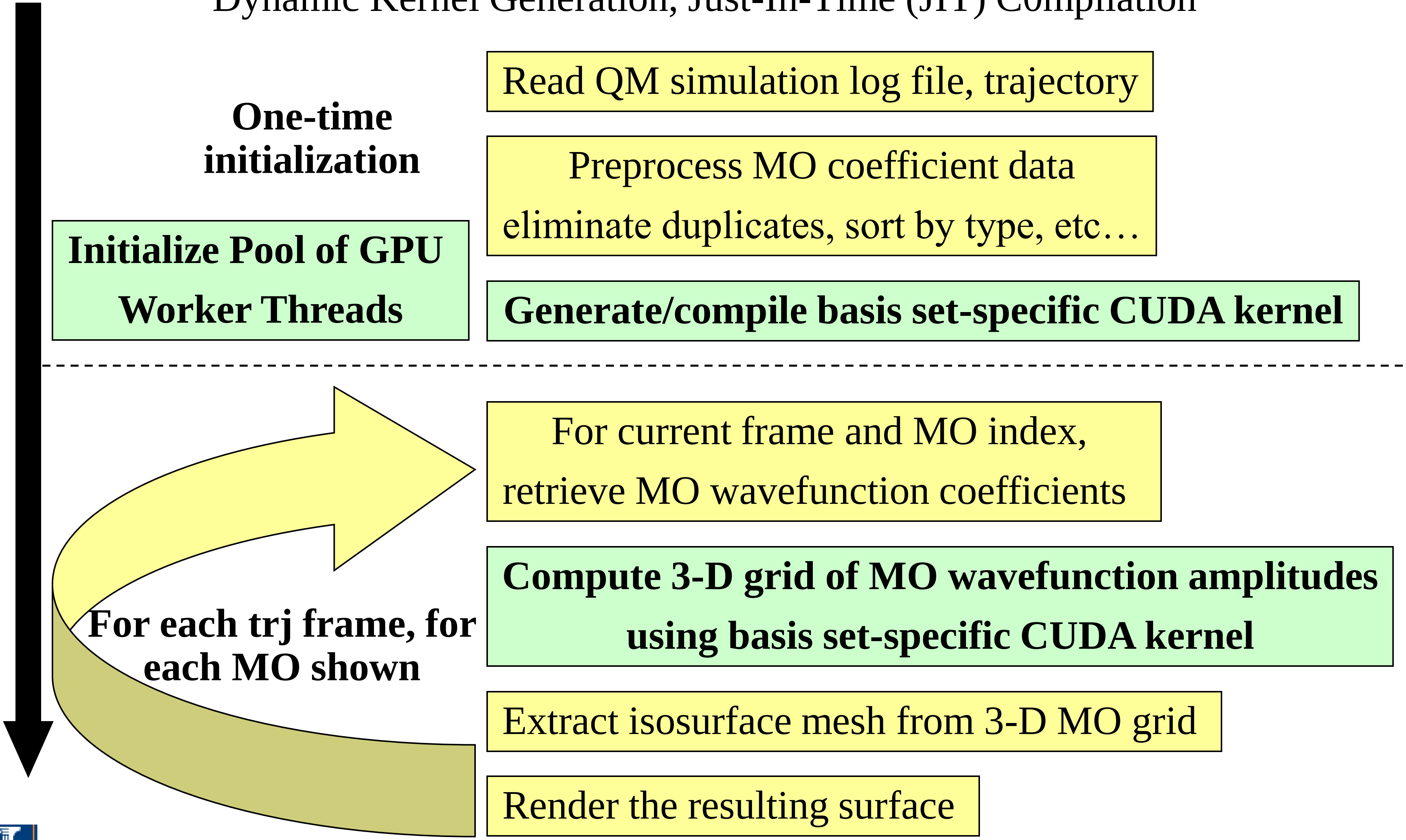
Loop over angular momenta for this shell type (1 to ~15) {}

}

}

# Molecular Orbital Computation and Display Process

## Dynamic Kernel Generation, Just-In-Time (JIT) C0mpilation



**One-time initialization**

Read QM simulation log file, trajectory

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

**Initialize Pool of GPU Worker Threads**

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

**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 using basis set-specific CUDA kernel**

Extract isosurface mesh from 3-D MO grid

Render the resulting surface

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

## 2.6GHz Intel X5550 vs. NVIDIA C2050

| Kernel | Cores/GPUs | Runtime (s) | Speedup |
|---|---|---|---|
| CPU ICC-SSE | 1 | 30.64 | 1.0 |
| CPU ICC-SSE | 8 | 4.13 | 7.4 |
| CUDA-JIT, Zero-copy | 1 | 0.174 | 176 |

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

JIT kernels eliminate overhead for low trip count for loops, replace dynamic table lookups with constants, and increase floating point arithmetic intensity

# VMD Molecular Orbital NVRTC JIT Performance Results for $C_{60}$

## Intel X5550 CPU, Quadro M6000 GPU

| Kernel | Cores/GPUs | Runtime (s) | Speedup vs socket (vs core) |
|---|---|---|---|
| Intel X5550-SSE | 1 | 30.64 | 0.13 ( 1.0) |
| Intel X5550-SSE | 8 | 4.13 | 1.00 ( 7.4) |
| Quadro M6000 const | 1 | 0.069 | 60.0 (444.) |
| Quadro M6000 shared | 1 | 0.102 | 30.4 (225.) |
| **Quadro M6000 NVRTC JIT** | **1** | **0.0404** | **102 (758.)** |

**NVRTC JIT w/ data-specific kernel yields a 1.7x speed increase over the fastest offline-compiled fully-general loop based kernel (constant memory kernel).**

```
for (shell=0; shell < maxshell; shell++)  {
  float contracted_gto = 0.0f;

  // Loop over the Gaussian primitives of CGTO
  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  * expf(-exponent*dist2);
    prim_counter += 2;
  }
```

General loop-based
data-dependent MO
CUDA kernel

Runtime-generated
data-specific MO
CUDA kernel compiled
via **CUDA 7.0**
**NVRTC** 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);
```

**1.8x Faster**

```
for (shell=0; shell < maxshell; shell++)  {
  float contracted_gto = 0.0f;

  // Loop over the Gaussian primitives of CGTO
  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  * expf(-exponent*dist2);
    prim_counter += 2;
  }

  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
data-dependent  MO
CUDA kernel

←

Runtime-generated
data-specific MO
CUDA kernel compiled
via **CUDA 7.0**
**NVRTC** JIT…

→

**1.8x Faster**

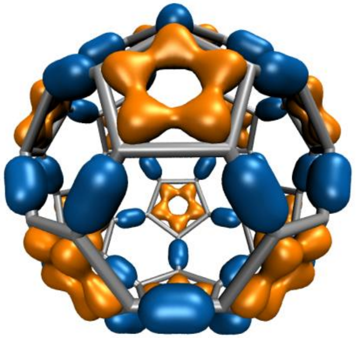
```
contracted_gto = 1.832937 * expf(-7.868272*dist2);
contracted_gto += 1.405380 * expf(-1.881289*dist2);
contracted_gto += 0.701383 * expf(-0.544249*dist2);
// P_SHELL
tmpshell = const_wave_f[ifunc++] * xdist;
tmpshell += const_wave_f[ifunc++] * ydist;
tmpshell += const_wave_f[ifunc++] * zdist;
value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

# Experiments Porting VMD CUDA Kernels to OpenCL

- Why mess with OpenCL?
  - OpenCL is very similar to CUDA, though a few years behind in terms of HPC features, aims to be the "OpenGL" of heterogeneous computing
  - As with CUDA, OpenCL provides a low-level language for writing high performance kernels, until compilers do a much better job of generating this kind of code
  - Potential to eliminate hand-coded SSE for CPU versions of compute intensive code, looks more like C and is easier for non-experts to read than hand-coded SSE or other vendor-specific instruction sets, intrinsics

# Molecular Orbital Inner Loop, Hand-Coded SSE Hard to Read, Isn't It? (And this is the "pretty" version!)

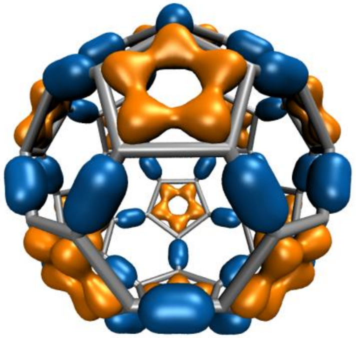
```
for (shell=0; shell < maxshell; shell++) {
    __m128 Cgto = _mm_setzero_ps();
    for (prim=0; prim<num_prim_per_shell[shell_counter]; prim++) {
        float exponent        = -basis_array[prim_counter      ];
        float contract_coeff =  basis_array[prim_counter + 1];
        __m128 expval = _mm_mul_ps(_mm_load_ps1(&exponent), dist2);
        __m128 ctmp = _mm_mul_ps(_mm_load_ps1(&contract_coeff), exp_ps(expval));
        Cgto = _mm_add_ps(contracted_gto, ctmp);
        prim_counter += 2;
    }
    __m128 tshell = _mm_setzero_ps();
    switch (shell_types[shell_counter]) {
        case S_SHELL:
            value = _mm_add_ps(value, _mm_mul_ps(_mm_load_ps1(&wave_f[ifunc++]), Cgto));    break;
        case P_SHELL:
            tshell = _mm_add_ps(tshell, _mm_mul_ps(_mm_load_ps1(&wave_f[ifunc++]), xdist));
            tshell = _mm_add_ps(tshell, _mm_mul_ps(_mm_load_ps1(&wave_f[ifunc++]), ydist));
            tshell = _mm_add_ps(tshell, _mm_mul_ps(_mm_load_ps1(&wave_f[ifunc++]), zdist));
            value = _mm_add_ps(value, _mm_mul_ps(tshell, Cgto));
            break;
```



**Until now, writing SSE kernels for CPUs required assembly language, compiler intrinsics, various libraries, or a really smart autovectorizing compiler and lots of luck...**

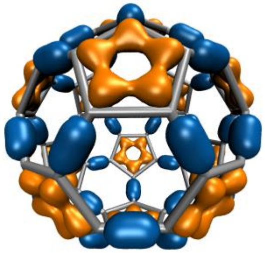
# Molecular Orbital Inner Loop, OpenCL Vec4 Ahhh, much easier to read!!!

```
for (shell=0; shell < maxshell; shell++) {
     float4 contracted_gto = 0.0f;
     for (prim=0; prim < const_num_prim_per_shell[shell_counter];  prim++) {
          float exponent          = const_basis_array[prim_counter      ];
          float contract_coeff = const_basis_array[prim_counter + 1];
          contracted_gto += contract_coeff * native_exp2(-exponent*dist2);
          prim_counter += 2;
     }
     float4 tmpshell=0.0f;
     switch (const_shell_symmetry[shell_counter]) {
          case S_SHELL:
              value += const_wave_f[ifunc++] * contracted_gto;      break;
         case P_SHELL:
              tmpshell += const_wave_f[ifunc++] * xdist;
              tmpshell += const_wave_f[ifunc++] * ydist;
              tmpshell += const_wave_f[ifunc++] * zdist;
              value += tmpshell * contracted_gto;
              break;
```



**OpenCL's C-like kernel language is easy to read, even 4-way vectorized kernels can look similar to scalar CPU code. All 4-way vectors shown in green.**

# Apples to Oranges Performance Results: OpenCL Molecular Orbital Kernels

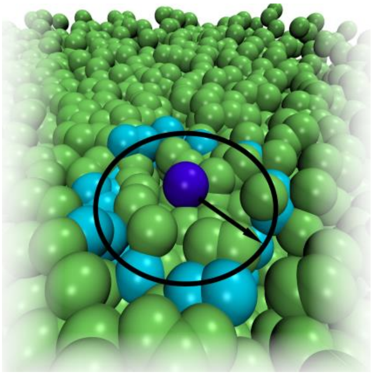


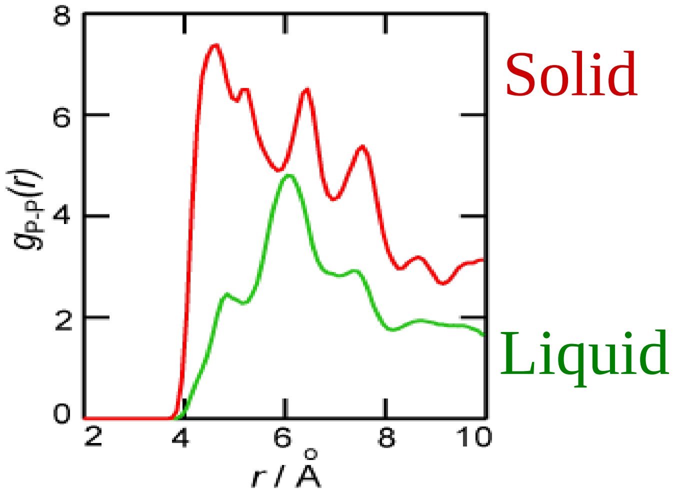
| Kernel | Cores | Runtime (s) | Speedup |
|---|---|---|---|
| Intel QX6700 CPU ICC-SSE (SSE intrinsics) | 1 | 46.580 | 1.00 |
| Intel Core2 Duo CPU OpenCL scalar | 2 | 43.342 | 1.07 |
| Intel Core2 Duo CPU OpenCL vec4 | 2 | 8.499 | 5.36 |
| Cell OpenCL vec4*** no __constant | 16 | 6.075 | 7.67 |
| Radeon 4870 OpenCL scalar | 10 | 2.108 | 22.1 |
| Radeon 4870 OpenCL vec4 | 10 | 1.016 | 45.8 |
| GeForce GTX 285 OpenCL vec4 | 30 | 0.364 | 127.9 |
| GeForce GTX 285 CUDA 2.1 scalar | 30 | 0.361 | 129.0 |
| GeForce GTX 285 OpenCL scalar | 30 | 0.335 | 139.0 |
| GeForce GTX 285 CUDA 2.0 scalar | 30 | 0.327 | 142.4 |

**Minor varations in compiler quality can have a strong effect on “tight” kernels. The two results shown for CUDA demonstrate performance variability with compiler revisions, and that with vendor effort, OpenCL has the potential to match the performance of other APIs.**

# Radial Distribution Function

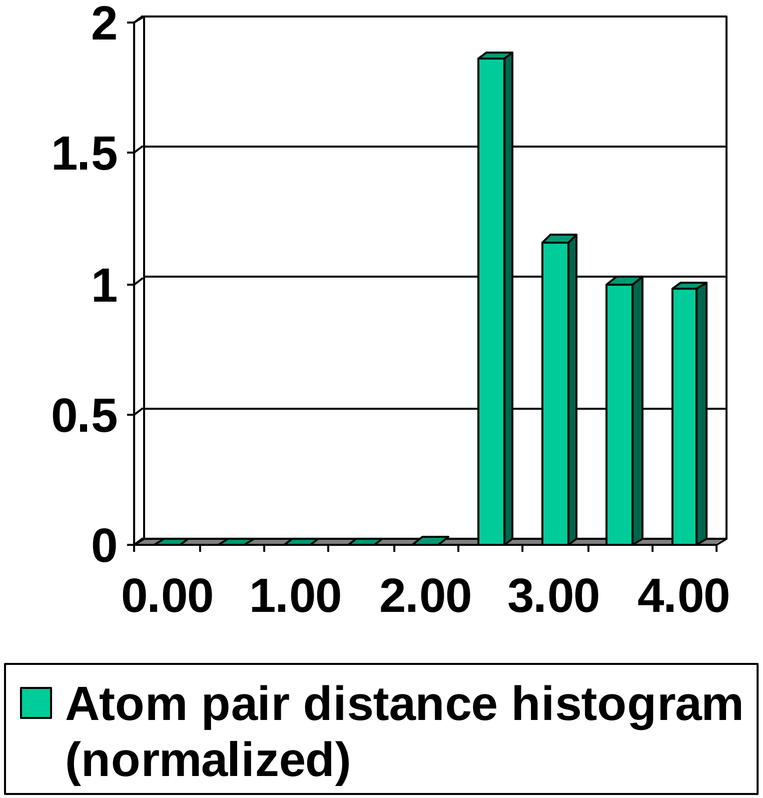
- RDFs describes how atom density varies with distance
- Can be compared with experiments
- Shape indicates phase of matter: sharp peaks appear for solids, smoother for liquids
- Quadratic time complexity $O(N^2)$





# Histogramming

- Partition population of data values into discrete bins
- Compute by traversing input population and incrementing bin counters



# Computing RDFs

- Compute distances for all pairs of atoms between two groups of atoms A and B
- A and B may be the same, or different
- Use nearest image convention for periodic systems
- Each pair distance is inserted into a histogram
- Histogram is normalized one of several ways depending on use, but usually according to the volume of the spherical shells associated with each histogram bin

# Computing RDFs on CPUs

- Atom coordinates can be traversed in a strictly consecutive access pattern, yielding good cache utilization
- Since RDF histograms are usually small to moderate in size, they normally fit entirely in L2 cache
- CPUs can compute the entire histogram in a **single pass**, regardless of the problem size or number of histogram bins

# Histogramming on the CPU (slow-and-simple C)

```
memset(histogram, 0, sizeof(histogram));
for (i=0; i<numdata; i++) {
  float val = data[i];
  if (val >= minval  && val <= maxval) {
    int bin = (val - minval) / bindelta;
    histogram[bin]++;
  }
}
```

Fetch-and-increment:

random access updates to histogram bins…

# Parallel Histogramming on Multi-core CPUs

- Parallel updates to a single histogram bin creates a **potential output conflict**
- CPUs have atomic increment instructions, but they often take hundreds of clock cycles; unsuitable…
- **SSE can't be used effectively:** lacks ability to "scatter" to memory (e.g. no *scatter-add*, no indexed store instructions)
- For small numbers of CPU cores, it is best to **replicate** and **privatize** the histogram for each CPU thread, compute them independently, and combine the separate histograms in a final reduction step

# Computing RDFs on the GPU

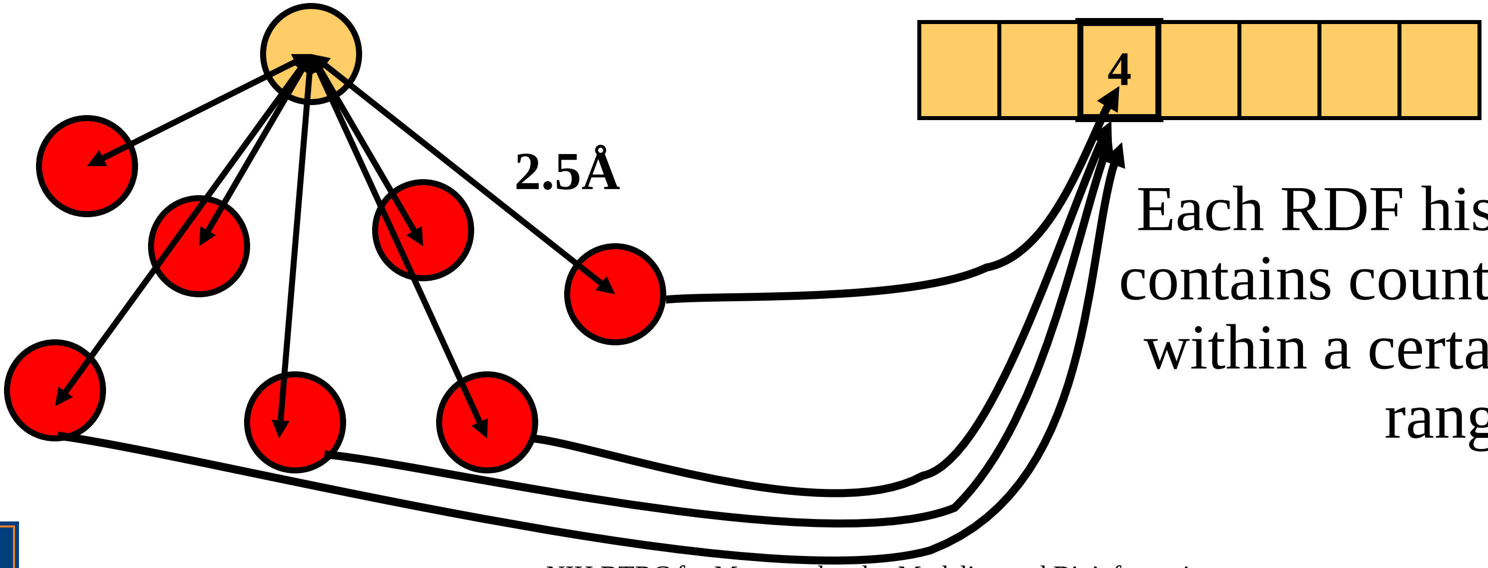
- Need tens of thousands of independent threads
- Each GPU thread computes one or more atom pair distances
- Performance is limited by the speed of histogramming
- Histograms are best stored in fast on-chip shared memory
- Small size of shared memory severely constrains the range of viable histogram update techniques
- **Fast CUDA implementation on Fermi: 30-92x faster than CPU**

# Computing Atom Pair Distances on the GPU

- Memory access pattern is simple
- Primary consideration is **amplification of effective memory bandwidth**, through use of GPU on-chip shared memory, caches, and broadcast of data to multiple or all threads in a thread block

# Radial Distribution Functions on GPUs

- Load blocks of atoms into shared memory and constant memory, compute periodic boundary conditions and atom-pair distances, all in parallel…
- Each thread computes all pair distances between its atom and all atoms in constant memory, incrementing the appropriate bin counter in the RDF histogram..



Each RDF histogram bin contains count of particles within a certain distance range

# GPU Histogramming

- Tens of thousands of threads concurrently computing atom distance pairs…
- Far too many threads for a simple per-thread histogram privatization approach like CPU…
- Viable approach: **per-warp histograms**
- Fixed size shared memory limits histogram size that can be computed in a single pass
- Large histograms require **multiple passes**, but we can skip block pairs that are known not to contribute to a histogram window

# Per-warp Histogram Approach

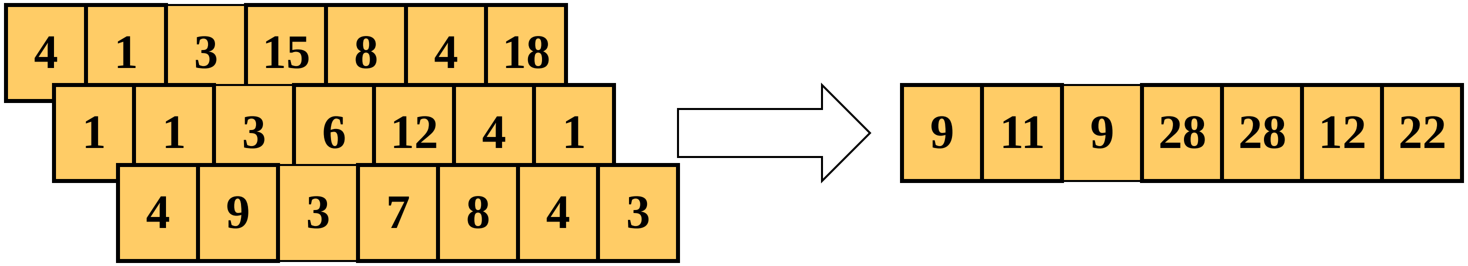
- Each warp maintains its own **private** histogram in on-chip shared memory
- Each thread in the warp computes an atom pair distance and updates a histogram bin in parallel
- Conflicting histogram bin updates are resolved using one of two schemes:
  - Shared memory write combining with thread-tagging technique (older hardware, e.g. G80, G9x)
  - **atomicAdd()** to shared memory (new hardware)

# RDF Inner Loops (abbreviated, xdist-only)

```
// loop over all atoms in constant memory
for (iblock=0; iblock<loopmax2; iblock+=3*NCUDABLOCKS*NBLOCK) {
    __syncthreads();
    for (i=0; i<3; i++) xyzi[threadIdx.x + i*NBLOCK]=pxi[iblock + i*NBLOCK]; // load coords…
    __syncthreads();
    for (joffset=0; joffset<loopmax; joffset+=3) {
        rxij=fabsf(xyzi[idxt3  ] - xyzj[joffset  ]);  // compute distance, PBC min image convention
        rxij2=celld.x - rxij;
        rxij=fminf(rxij, rxij2);
        rij=rxij*rxij;
        […other distance components…]
        rij=sqrtf(rij + rxij*rxij);
        ibin=__float2int_rd((rij-rmin)*delr_inv);
        if (ibin<nbins && ibin>=0 && rij>rmin2) {
          atomicAdd(llhists1+ibin, 1U);
        }
    } //joffset
} //iblock
```

# Writing/Updating Histogram in Global Memory

- When thread block completes, add independent per-warp histograms together, and write to per-thread-block histogram in global memory
- Final reduction of all per-thread-block histograms stored in global memory



4 1 3 15 8 4 18

1 1 3 6 12 4 1

4 9 3 7 8 4 3

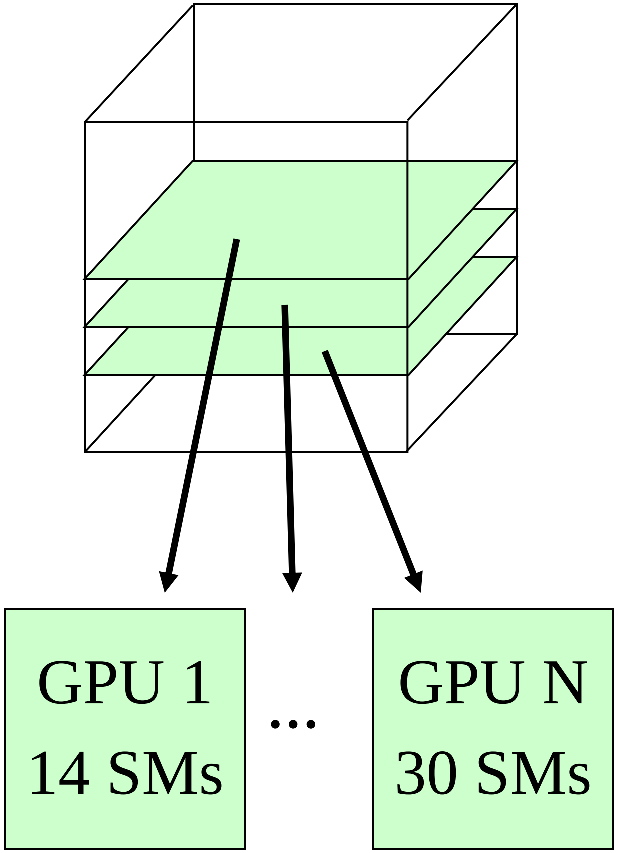
9 11 9 28 28 12 22

# Preventing Integer Overflows

- Since all-pairs RDF calculation computes many billions of pair distances, we have to **prevent integer overflow** for the 32-bit histogram bin counters (supported by the **atomicAdd()** routine)
- We compute full RDF calculation in **multiple kernel launches**, so each kernel launch computes partial histogram
- Host routines read GPUs and increment large (e.g. long, or double) histogram counters in host memory after each kernel completes

# Multi-GPU Load Balance

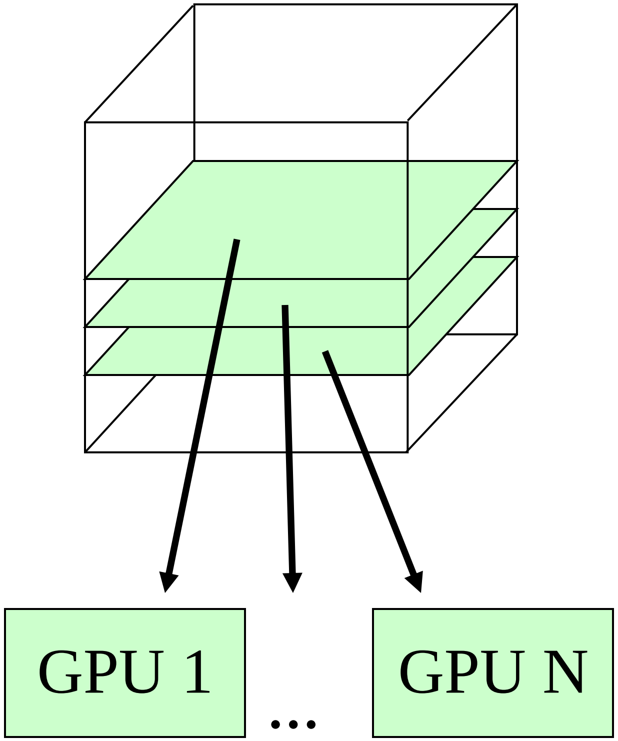
- Many early CUDA codes assumed all GPUs were identical
- Host machines may contain a diversity of GPUs of varying capability (discrete, IGP, etc)
- Different GPU on-chip and global memory capacities may need different problem "tile" sizes
- Static decomposition works poorly for non-uniform workload, or diverse GPUs



# Multi-GPU Dynamic Work Distribution

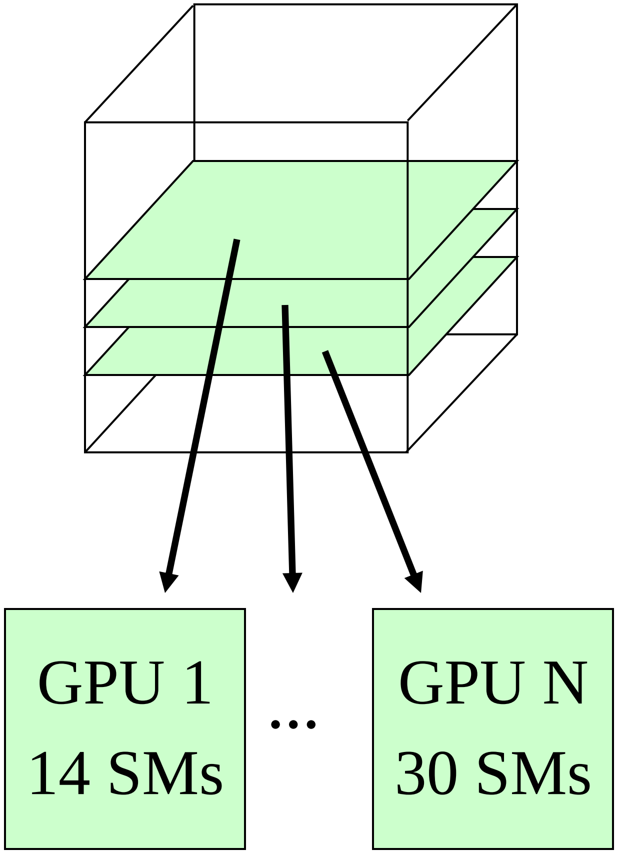
```
// Each GPU worker thread loops over
// subset of work items…
while (!threadpool_next_tile(&parms,
    tilesize, &tile){
  // Process one work item…
  // Launch one CUDA kernel for each
  //   loop iteration taken…
  // Shared iterator automatically
  //   balances load on GPUs
}
```

Dynamic work distribution



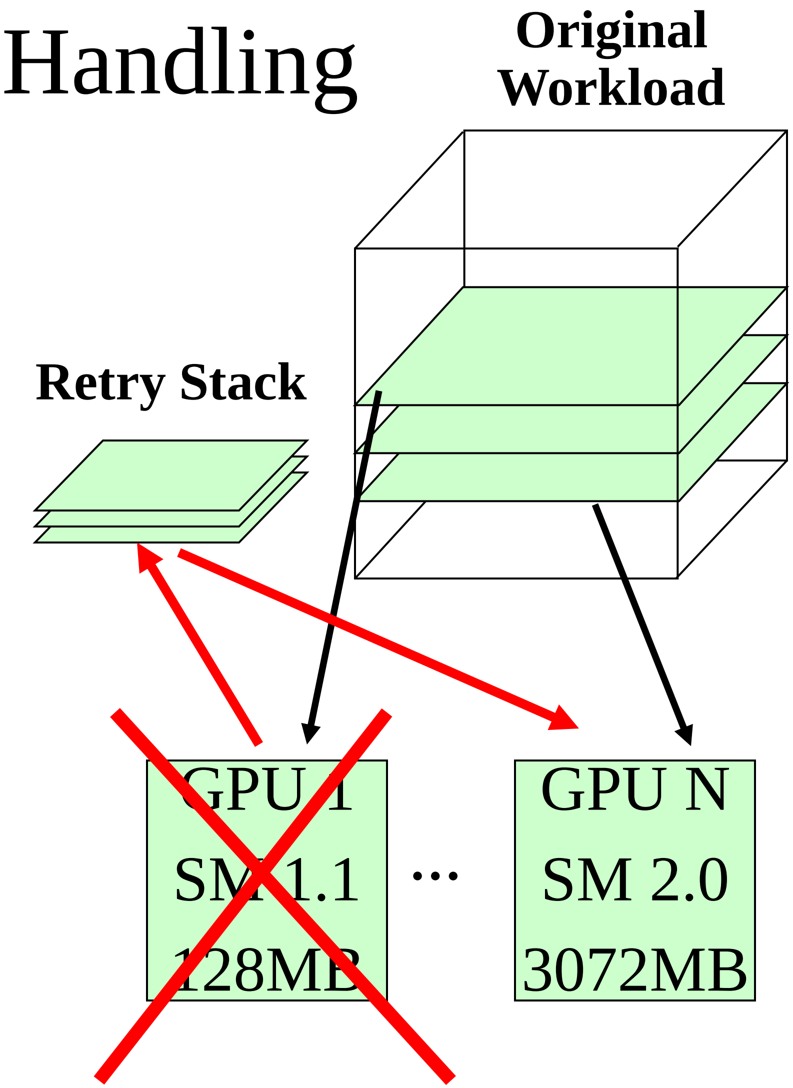
# Multi-GPU RDF Calculation

- Distribute combinations of tiles of atoms and histogram regions to different GPUs
- Decomposed over two dimensions to obtain enough work units to balance GPU loads
- Each GPU computes its own histogram, and all results are combined for final histogram



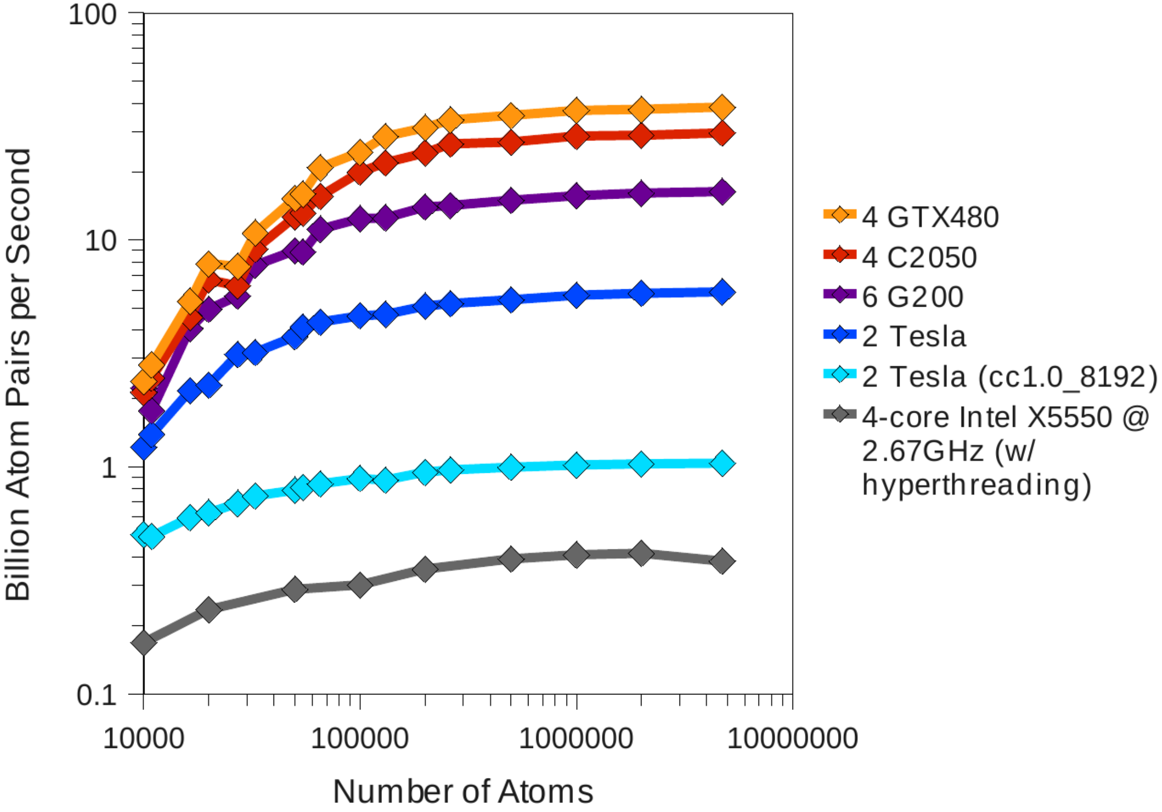
# Multi-GPU Runtime Error/Exception Handling

- Competition for resources from other applications can cause runtime failures, e.g. GPU out of memory half way through an algorithm
- Handle exceptions, e.g. convergence failure, NaN result, insufficient compute capability/features
- Handle and/or reschedule failed tiles of work



# Multi-GPU RDF Performance

- 4 NVIDIA GTX480 GPUs 30 to 92x faster than 4-core Intel X5550 CPU
- Fermi GPUs ~3x faster than GT200 GPUs: larger on-chip shared memory



**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.

# Acknowledgements

- Theoretical and Computational Biophysics Group, University of Illinois at Urbana-Champaign
- NVIDIA CUDA Center of Excellence, University of Illinois at Urbana-Champaign
- NVIDIA CUDA team
- NVIDIA OptiX team
- NCSA Blue Waters Team
- Funding:
  - NSF OCI 07-25070
  - NSF PRAC “The Computational Microscope”
  - NIH support: 9P41GM104601, 5R01GM098243-02

NIH BTRC for Macromolecular Modeling and Bioinformatics
1990-2017
Beckman Institute
University of Illinois at
Urbana-Champaign

# GPU Computing Publications

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

- **Runtime and Architecture Support for Efficient Data Exchange in Multi-Accelerator Applications** Javier Cabezas, Isaac Gelado, John E. Stone, Nacho Navarro, David B. Kirk, and Wen-mei Hwu. IEEE Transactions on Parallel and Distributed Systems, 26(5):1405-1418, 2015.
- **Unlocking the Full Potential of the Cray XK7 Accelerator** Mark Klein and John E. Stone. Cray Users Group, Lugano Switzerland, May 2014.
- **Simulation of reaction diffusion processes over biologically relevant size and time scales using multi-GPU workstations** Michael J. Hallock, John E. Stone, Elijah Roberts, Corey Fry, and Zaida Luthey-Schulten. Journal of Parallel Computing, 40:86-99 2014.
- **GPU-Accelerated Analysis and Visualization of Large Structures Solved by Molecular Dynamics Flexible Fitting** John E. Stone, Ryan McGreevy, Barry Isralewitz, and Klaus Schulten. Faraday Discussions, 169:265-283, 2014.
- **GPU-Accelerated Molecular Visualization on Petascale Supercomputing Platforms.** J. Stone, K. L. Vandivort, and K. Schulten. UltraVis'13: Proceedings of the 8th International Workshop on Ultrascale Visualization, pp. 6:1-6:8, 2013.
- **Early Experiences Scaling VMD Molecular Visualization and Analysis Jobs on Blue Waters.** J. E. Stone, B. Isralewitz, and K. Schulten. Extreme Scaling Workshop (XSW), pp. 43-50, 2013.
- **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**.**

# GPU Computing Publications

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

- **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.
- **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.
- **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.
- **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.

# GPU Computing Publications
## http://www.ks.uiuc.edu/Research/gpu/

- **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 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 347-358, 2010.
- **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 Pricessing 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.