CUDA Applications I

John E. Stone

Theoretical and Computational Biophysics Group

Beckman Institute for Advanced Science and Technology

University of Illinois at Urbana-Champaign

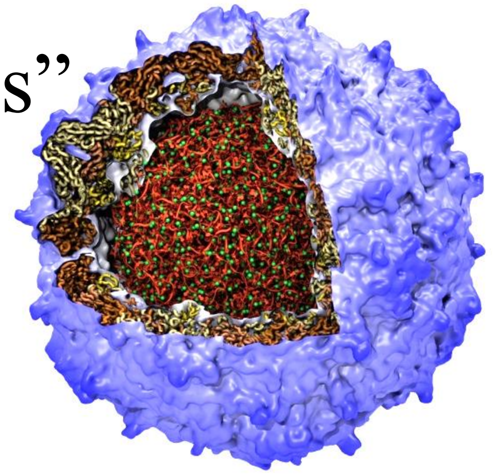
http://www.ks.uiuc.edu/Research/gpu/

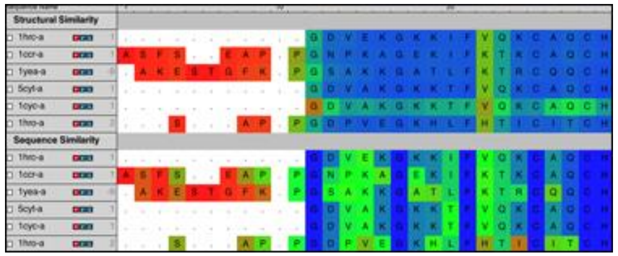
Cape Town GPU Workshop

Cape Town, South Africa, May 2, 2013

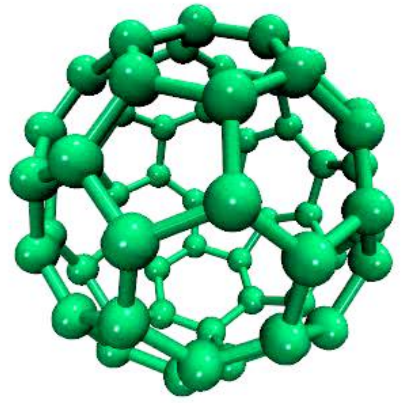
VMD – "Visual Molecular Dynamics"

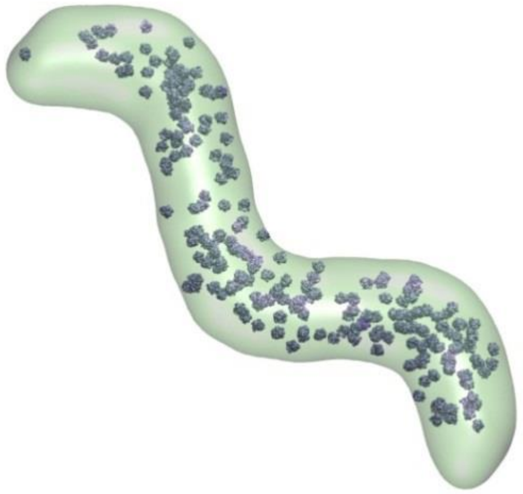
- Visualization and analysis of:
  - molecular dynamics simulations
  - quantum chemistry calculations
  - particle systems and whole cells
  - sequence data
- User extensible w/ scripting and plugins
- http://www.ks.uiuc.edu/Research/vmd/

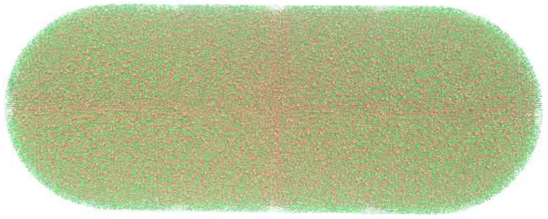

Poliovirus



Ribosome Sequences

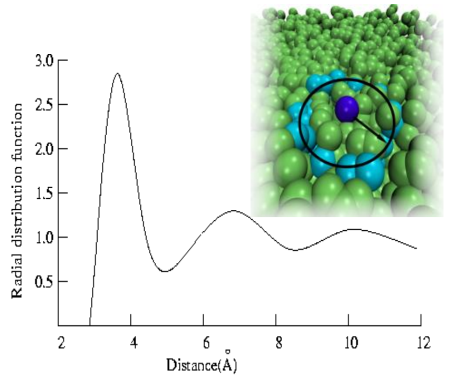

Electrons in
Vibrating Buckyball

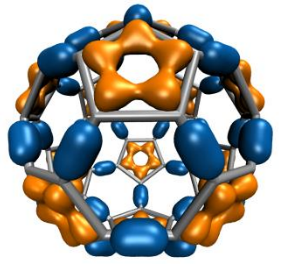

Cellular Tomography,
Cryo-electron Microscopy


Whole Cell Simulations

GPU Accelerated Trajectory Analysis and Visualization in VMD

GPU-Accelerated Feature	Peak speedup vs. single CPU core
Molecular orbital display	120x
Radial distribution function	92x
Electrostatic field calculation	44x
Molecular surface display	40x
Ion placement	26x
MDFF density map synthesis	26x
Implicit ligand sampling	25x
Root mean squared fluctuation	25x
Radius of gyration	21x
Close contact determination	20x
Dipole moment calculation	15x





Ongoing VMD GPU Development

- Development of new CUDA kernels for common molecular dynamics trajectory analysis tasks
- Increased memory efficiency of CUDA kernels for visualization and analysis of large structures
- Improving CUDA performance for batch mode MPI version of VMD used for in-place trajectory analysis calculations:
  - GPU-accelerated commodity clusters
  - GPU-accelerated Cray XK7 supercomputers: NCSA Blue Waters, ORNL Titan

Interactive Display & Analysis of Terabytes of Data:

Out-of-Core Trajectory I/O w/ Solid State Disks **and GPUs**

450MB/sec

to 8GB/sec

TWO DVD movies per second!

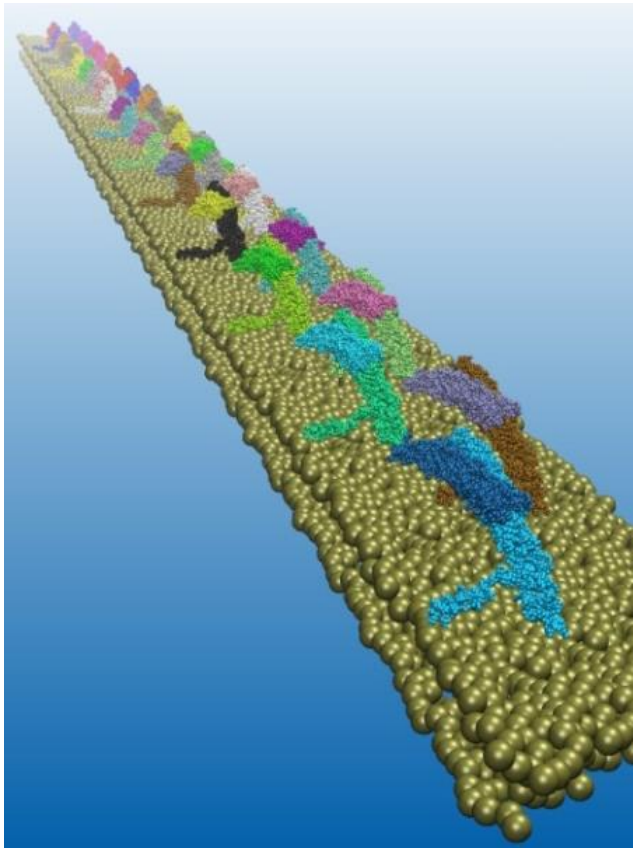
Commodity SSD, SSD RAID

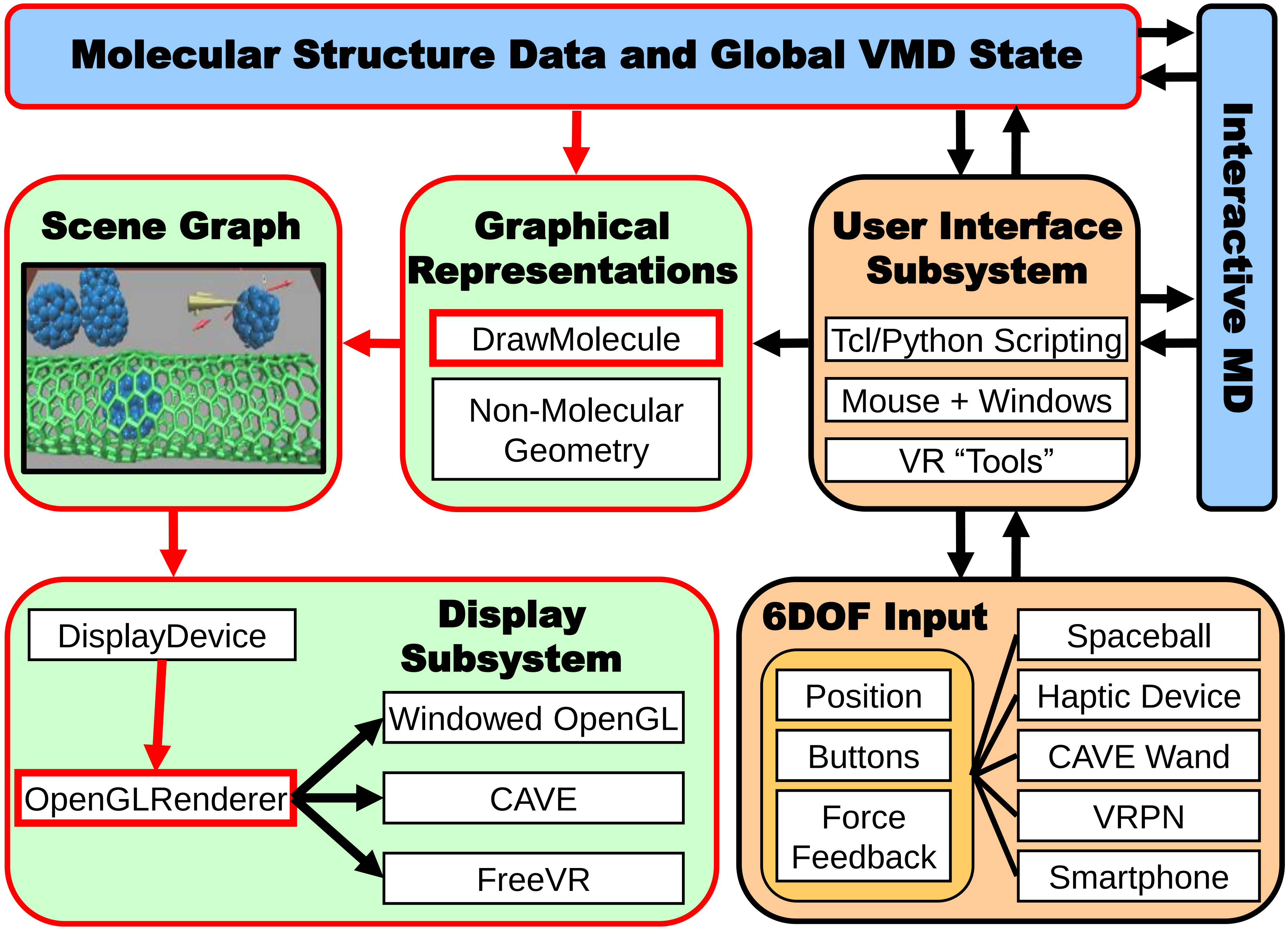
- Timesteps loaded on-the-fly (out-of-core)
  - Eliminates memory capacity limitations, even for multi-terabyte trajectory files
  - High performance achieved by new trajectory file formats, optimized data structures, and efficient I/O
- **GPUs accelerate per-timestep calculations**
- Analyze long trajectories significantly faster using just a personal computer

Immersive out-of-core visualization of large-size and long-timescale molecular dynamics trajectories. J. Stone, K. Vandivort, and K. Schulten. *Lecture Notes in Computer Science*, 6939:1-12, 2011.

Challenges for Immersive Visualization of Dynamics of Large Structures

- Graphical representations re-computed each trajectory timestep
- Visualizations often focus on interesting regions of substructure
- Fast display updates require rapid sparse traversal+gathering of molecular data for use in GPU computations and OpenGL display
  - Hand-vectorized SSE/AVX CPU atom selection traversal code increased performance of per-frame updates by another ~6x for several 100M atom test cases
- Graphical representation optimizations:
  - Reduce host-GPU bandwidth for displayed geometry
  - Optimized graphical representation generation routines for large atom counts, sparse selections


116M atom BAR domain test case:
200,000 selected atoms,
stereo trajectory animation 70 FPS,
static scene in stereo 116 FPS

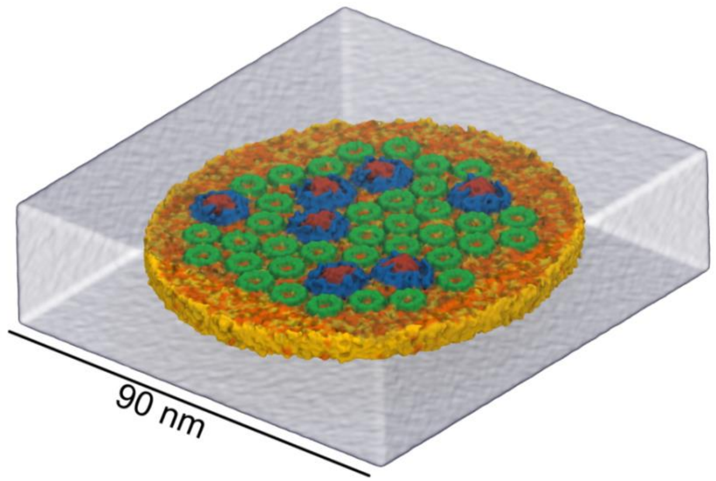

Molecular Structure Data and Global VMD State
Interactive MD
Scene Graph
Graphical Representations
DrawMolecule
Non-Molecular Geometry
User Interface Subsystem
Tcl/Python Scripting
Mouse + Windows
VR “Tools”
Display Subsystem
DisplayDevice
OpenGLRenderer
Windowed OpenGL
CAVE
FreeVR
6DOF Input
Position
Buttons
Force Feedback
Spaceball
Haptic Device
CAVE Wand
VRPN
Smartphone

Improving Performance for Large Datasets

- As the performance of GPUs has continued to increase, formerly "insignificant" CPU routines are becoming bottlenecks
  - A key feature of VMD is the ability to perform visualization and analysis operations on arbitrary user-selected subsets of the molecular structure
  - CPU-side atom selection traversal performance has begun to be a potential bottleneck when working with large structures of tens of millions of atoms
  - Both OpenGL rendering and CUDA analysis kernels (currently) depend on the CPU to gather selected atom data into buffers that are sent to the GPU
  - Hand-coded SSE/AVX optimizations have now improved the performance of these CPU preprocessing steps by up to 6x, keeping the CPU "out of the way"

20M atoms:

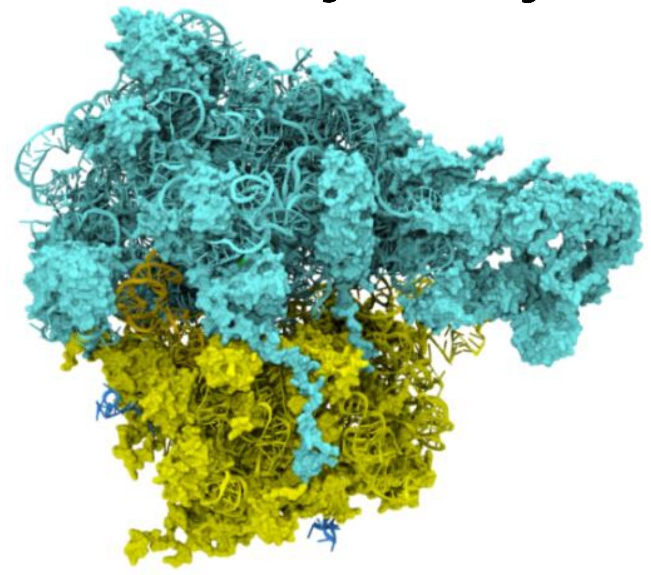
membrane patch and solvent

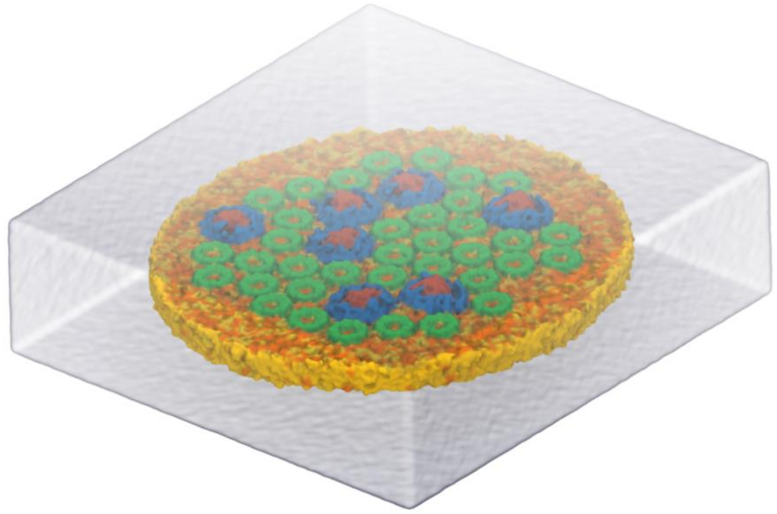


Improving Performance for Large Datasets: Make Key Data Structures GPU-Resident

- Eliminating the dependency on the host CPU to traverse, collect, and pack atom data will enable much higher GPU performance
- Long-term, best performance will be obtained by storing all molecule data locally in on-board GPU memory
  - GPU needs enough memory to store **both** molecular information, as well as the generated vertex arrays and texture maps used for rendering
  - With sufficient memory, only per-timestep time-varying data will have to copied into the GPU on-the-fly, and most other data can remain GPU-resident
  - Today’s GPUs have insufficient memory for very large structures, where the resulting performance increases would have the greatest impact
  - Soon we should begin to see GPUs with 16GB of on-board memory – enough to keep all of the static molecular structure data on the GPU full-time
- Once the full molecular data is GPU-resident, CUDA kernels can directly incorporate atom selection traversal for themselves
- CUDA Dynamic Parallelism will make more GPUs self sufficient

VMD Out-of-Core Trajectory I/O Performance: SSD Trajectory Format, PCIe3 8-SSD RAID





Ribosome w/ solvent	Membrane patch w/ solvent
3M atoms	20M atoms
3 frames/sec w/ HD	0.4 frames/sec w/ HD
77 frames/sec w/ SSDs	10 frames/sec w/ SSDs

New SSD Trajectory File Format 2x Faster vs. Existing Formats

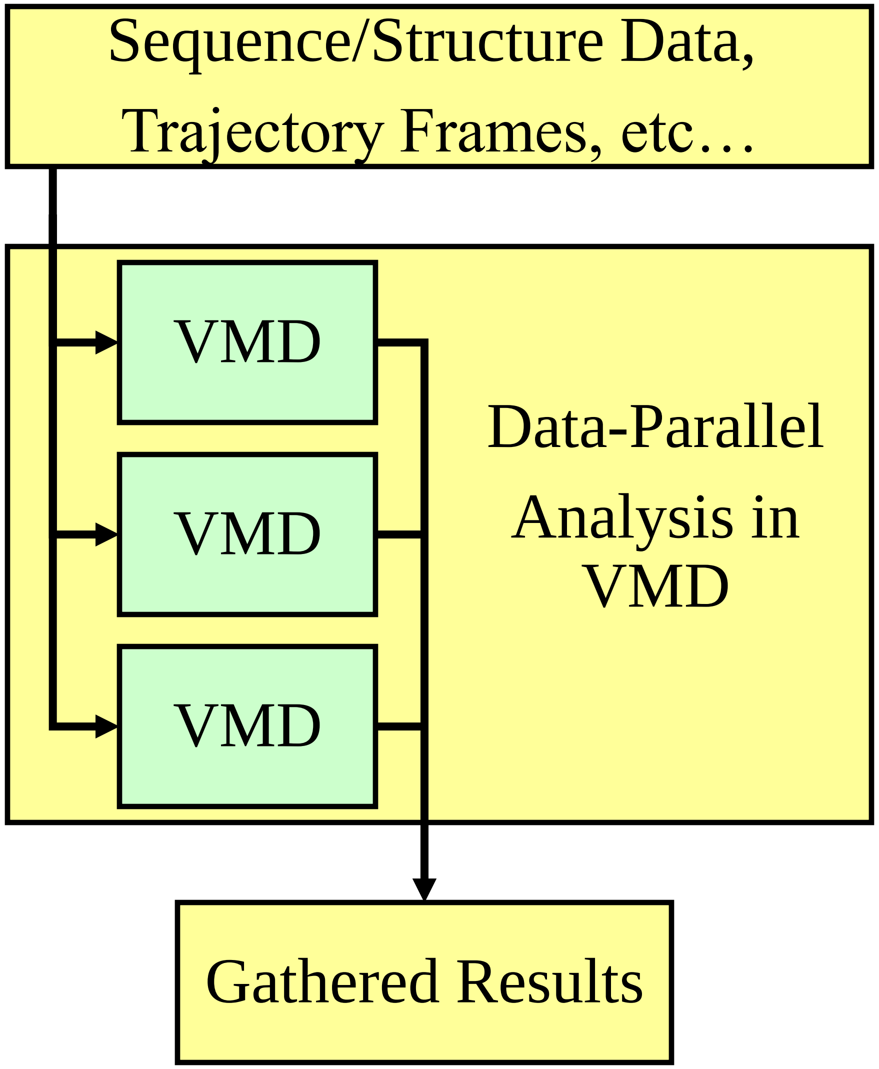
VMD I/O rate ~2.7 GB/sec w/ 8 SSDs in a single PCIe3 RAID0

Challenges for High Throughput Trajectory Visualization and Analysis

- It is not currently possible to fully exploit full I/O bandwidths when streaming data from SSD arrays (>4GB/sec) to GPU global memory **due to copies**
- Need to eliminated copies from disk controllers to host memory – bypass host entirely and perform zero-copy DMA operations straight from disk controllers to GPU global memory
- **Goal: GPUs directly pull in pages from storage systems bypassing host memory entirely**

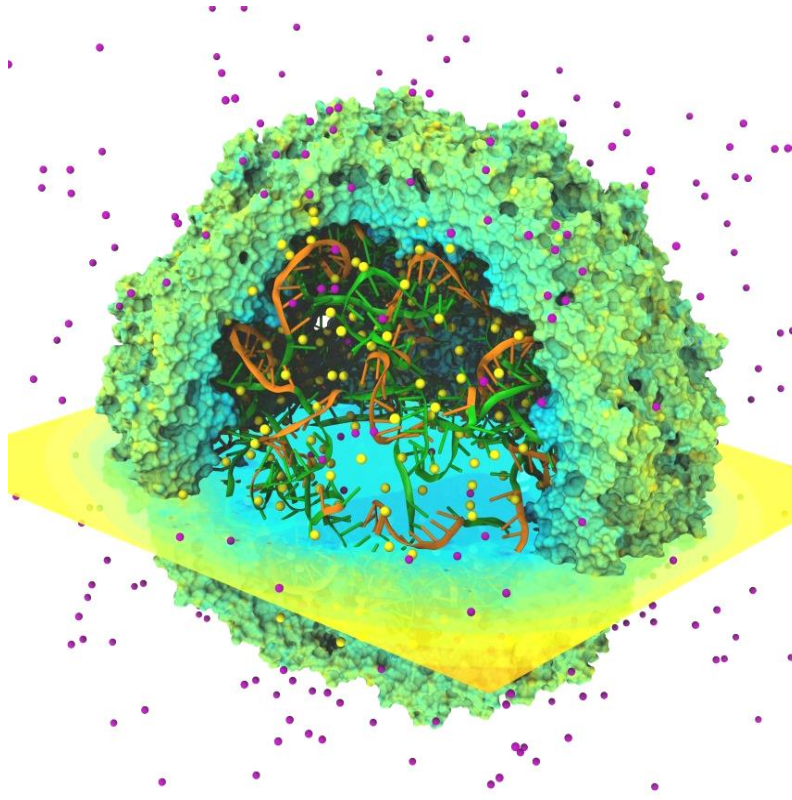
VMD for Demanding Analysis Tasks Parallel VMD Analysis w/ MPI

- Analyze trajectory frames, structures, or sequences in parallel on clusters and supercomputers:
  - Compute time-averaged electrostatic fields, MDFF quality-of-fit, etc.
  - Parallel rendering, movie making
- Addresses computing requirements beyond desktop
- User-defined parallel reduction operations, data types
- Dynamic load balancing:
  - Tested with up to 15,360 CPU cores
- **Supports GPU-accelerated clusters and supercomputers**



Time-Averaged Electrostatics Analysis on Energy-Efficient GPU Cluster

- **1.5 hour** job (CPUs) reduced to **3 min** (CPUs+GPU)
- Electrostatics of thousands of trajectory frames averaged
- Per-node power consumption on NCSA "AC" GPU cluster:
  - CPUs-only: 299 watts
  - CPUs+GPUs: 742 watts
- GPU Speedup: **25.5x**
- Power efficiency gain: **10.5x**



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. *The Work in Progress in Green Computing*, pp. 317-324, 2010.

NCSA Blue Waters Early Science System
Cray XK6 nodes w/ NVIDIA Tesla X2090

Time-Averaged Electrostatics Analysis on NCSA Blue Waters

NCSA Blue Waters Node Type	**Seconds per trajectory frame for one compute node**
Cray XE6 Compute Node: 32 CPU cores (2xAMD 6200 CPUs)	9.33
Cray XK6 GPU-accelerated Compute Node: 16 CPU cores + **NVIDIA X2090 (Fermi) GPU**	2.25
Speedup for GPU XK6 nodes vs. CPU XE6 nodes	**GPU nodes are 4.15x faster overall**
Early tests on XK7 nodes indicate MSM is becoming CPU-bound with the Kepler K20X GPU **Performance is not much faster (yet) than Fermi X2090** **May need to move spatial hashing and other algorithms onto the GPU.**	**In progress….**

Preliminary performance for VMD time-averaged electrostatics w/ Multilevel Summation Method on the NCSA Blue Waters Early Science System

Early Experiences with Kepler

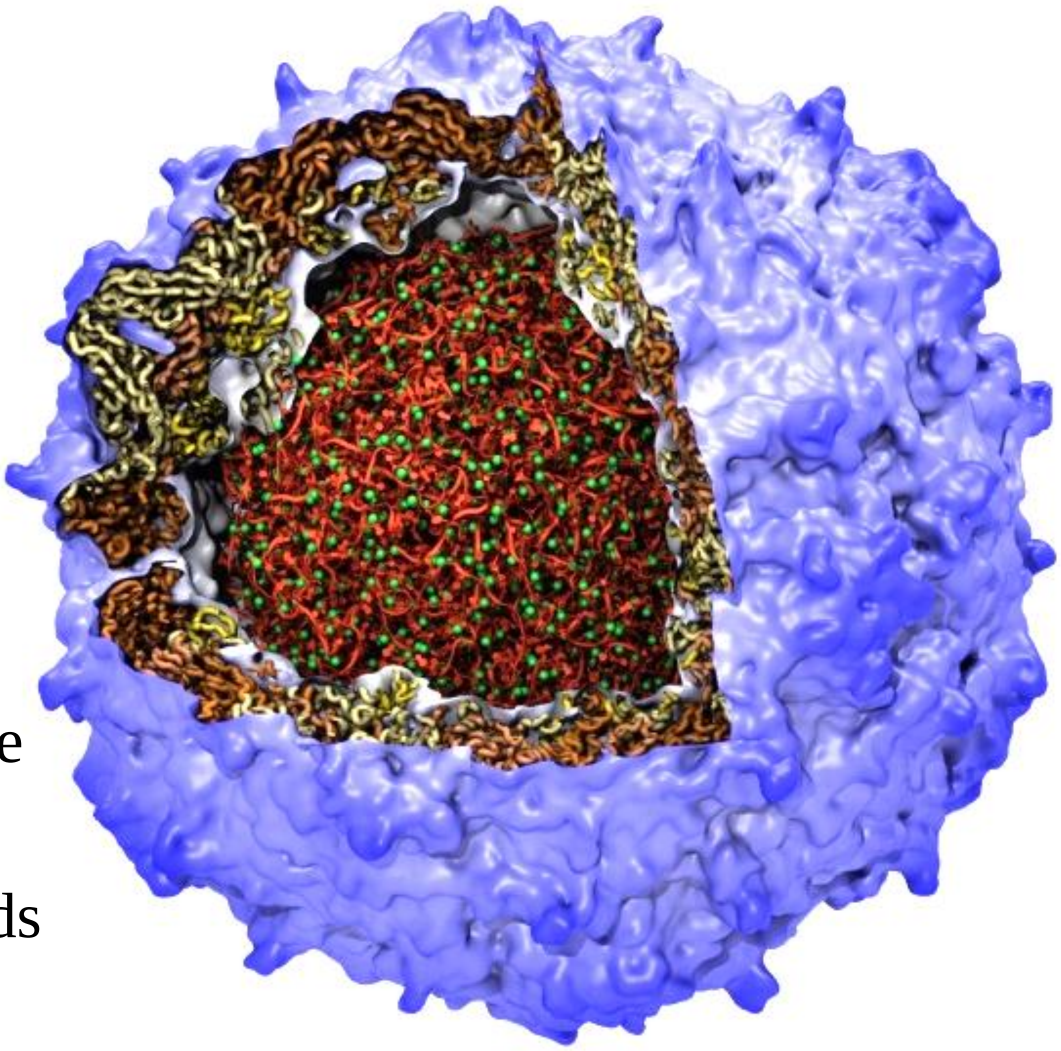
Preliminary Observations

- **Arithmetic is cheap, memory references are costly** (trend is certain to continue & intensify…)
- Different performance ratios for registers, shared mem, and various floating point operations vs. Fermi
- Kepler GK104 (e.g. GeForce 680) brings improved performance for some special functions vs. Fermi:

CUDA Kernel	**Dominant Arithmetic Operations**	**Kepler (GeForce 680) Speedup vs. Fermi (Quadro 7000)**
Direct Coulomb summation	rsqrtf()	2.4x
Molecular orbital grid evaluation	expf(), exp2f(), Multiply-Add	1.7x

Molecular Surface Visualization

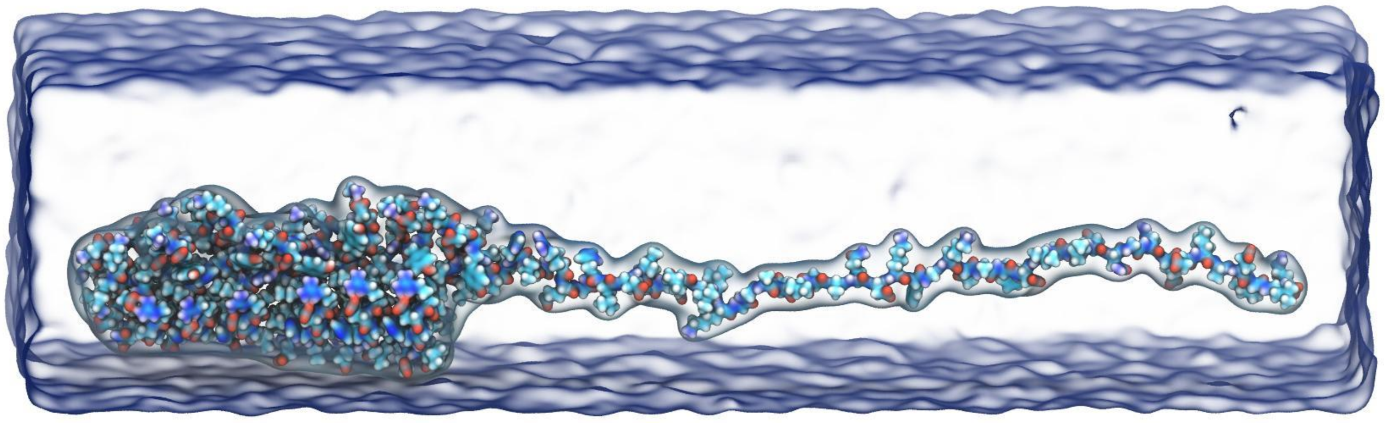
- Large biomolecular complexes are difficult to interpret with atomic detail graphical representations
- Even secondary structure representations become cluttered
- Surface representations are easier to use when greater abstraction is desired, but are computationally costly
- Most surface display methods 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

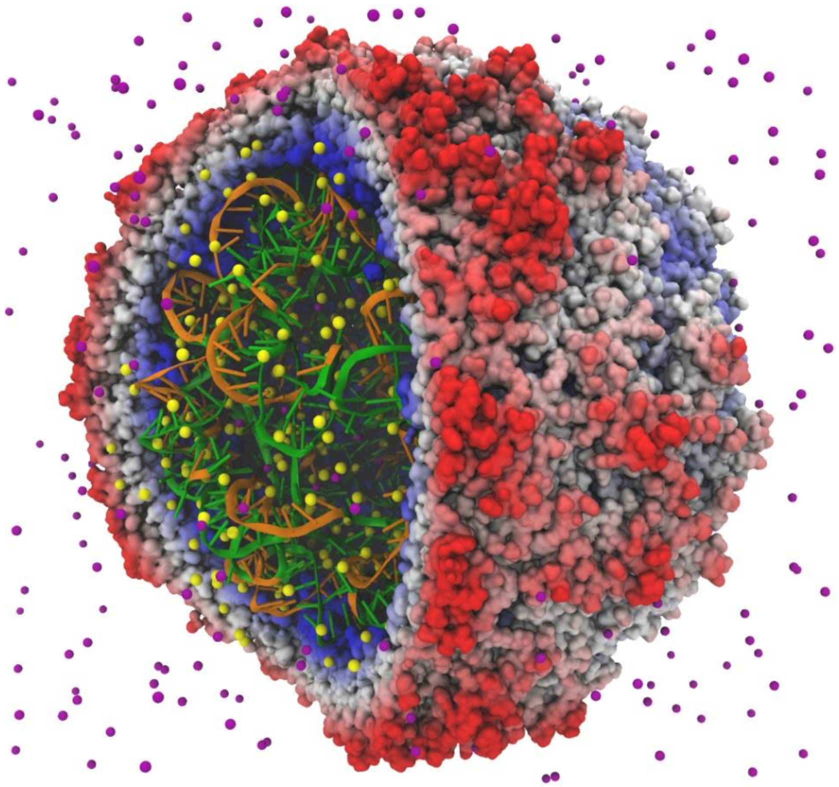


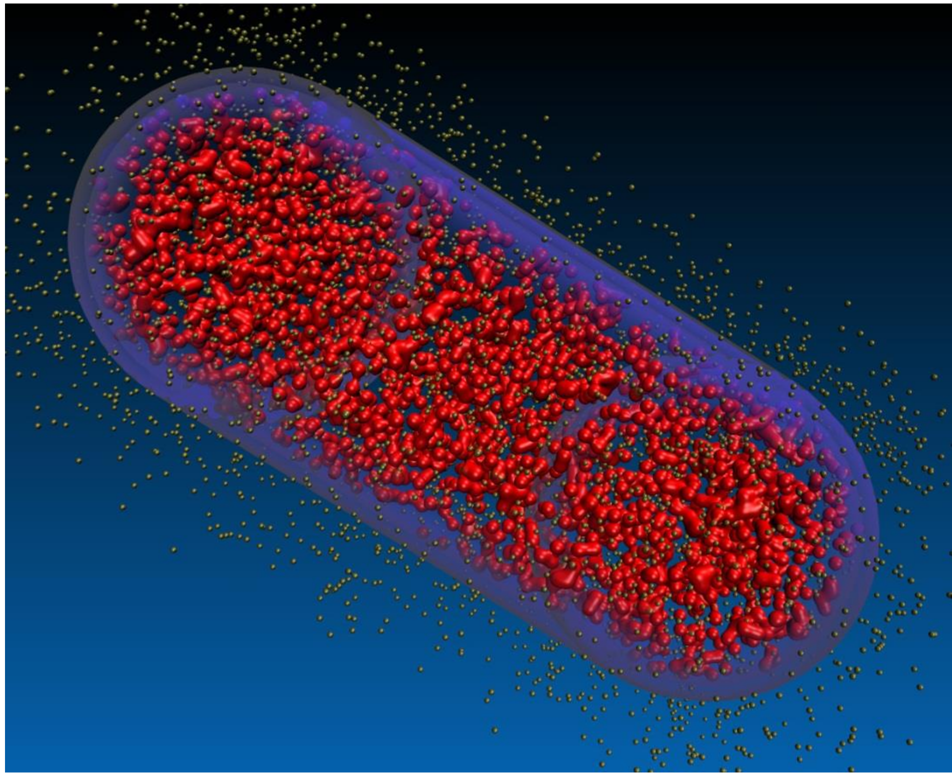
Fast Visualization of Gaussian Density Surfaces for Molecular Dynamics and Particle System Trajectories.

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

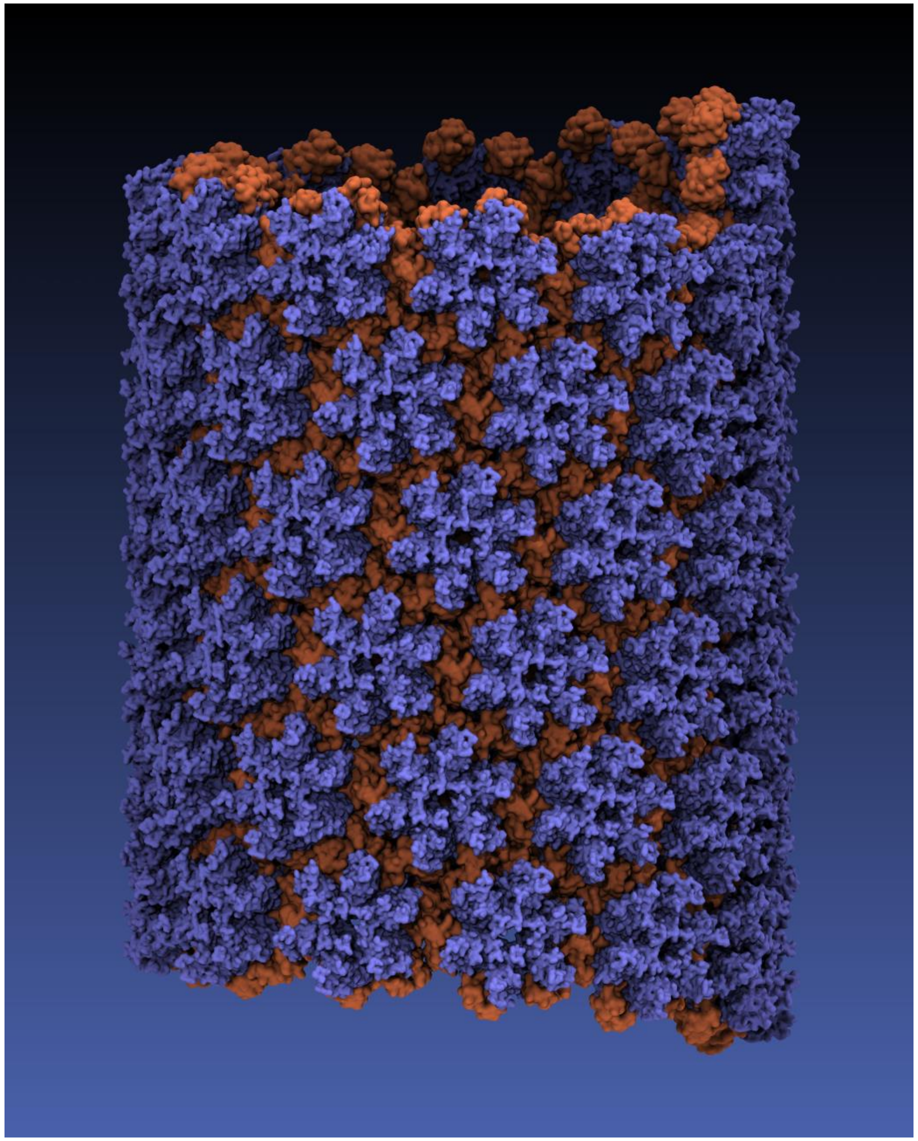
VMD "QuickSurf" Representation

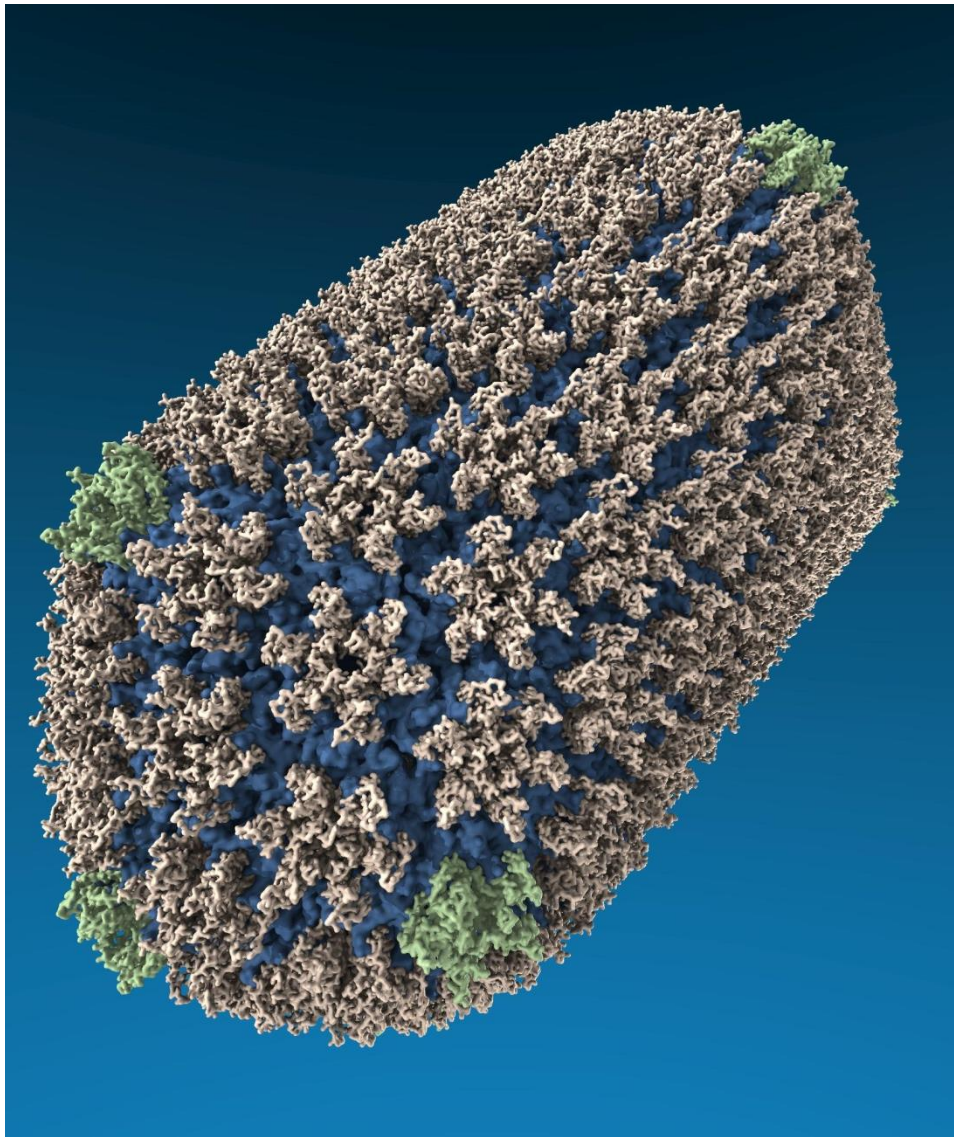
- Uses multi-core CPUs and GPU acceleration to enable **smooth real-time animation** of MD trajectories
- Linear-time algorithm, scales to millions of particles, as limited by memory capacity


Satellite Tobacco Mosaic Virus


Lattice Cell Simulations

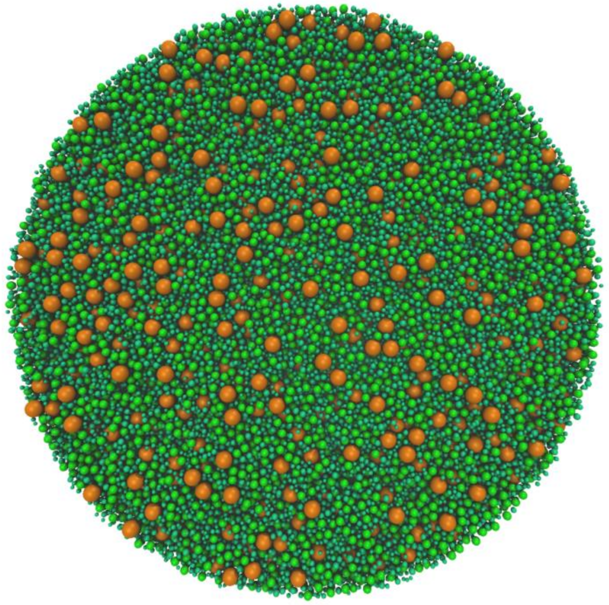
VMD “QuickSurf” Representation

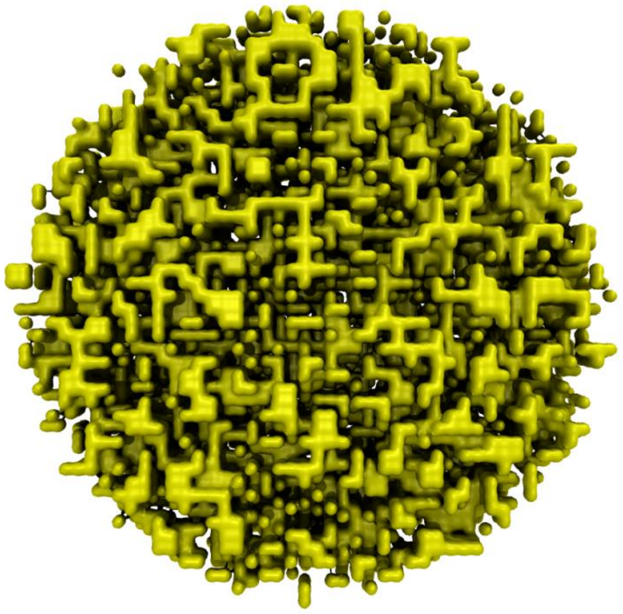


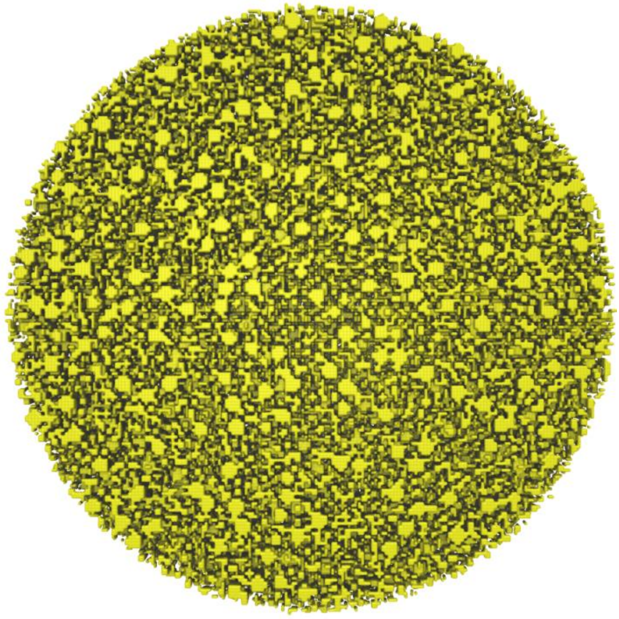


All-atom HIV capsid simulations

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

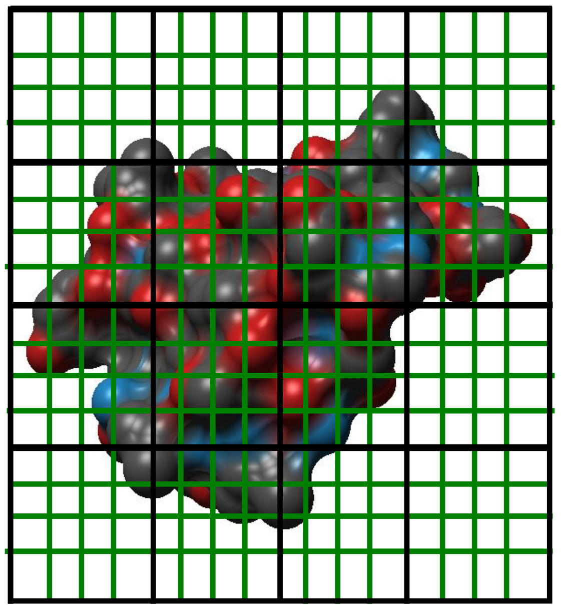
Lattice Microbes: High-performance stochastic simulation method for the reaction-diffusion master equation
E. Roberts, J. E. Stone, and Z. Luthey-Schulten.
J. Computational Chemistry 34 (3), 245-255, 2013.

QuickSurf Algorithm Overview

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

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

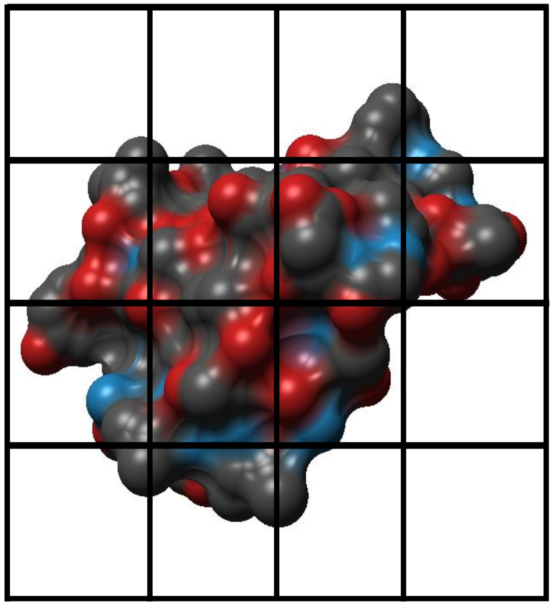
- Extract isosurface for a user-defined density value



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

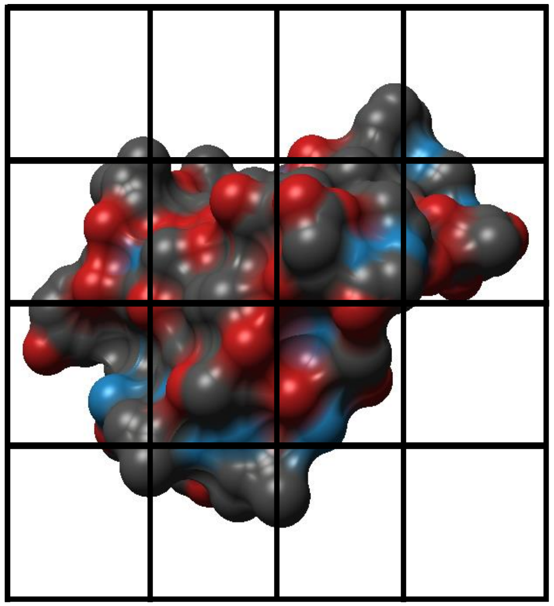
Spatial Hashing Algorithm Steps/Kernels

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

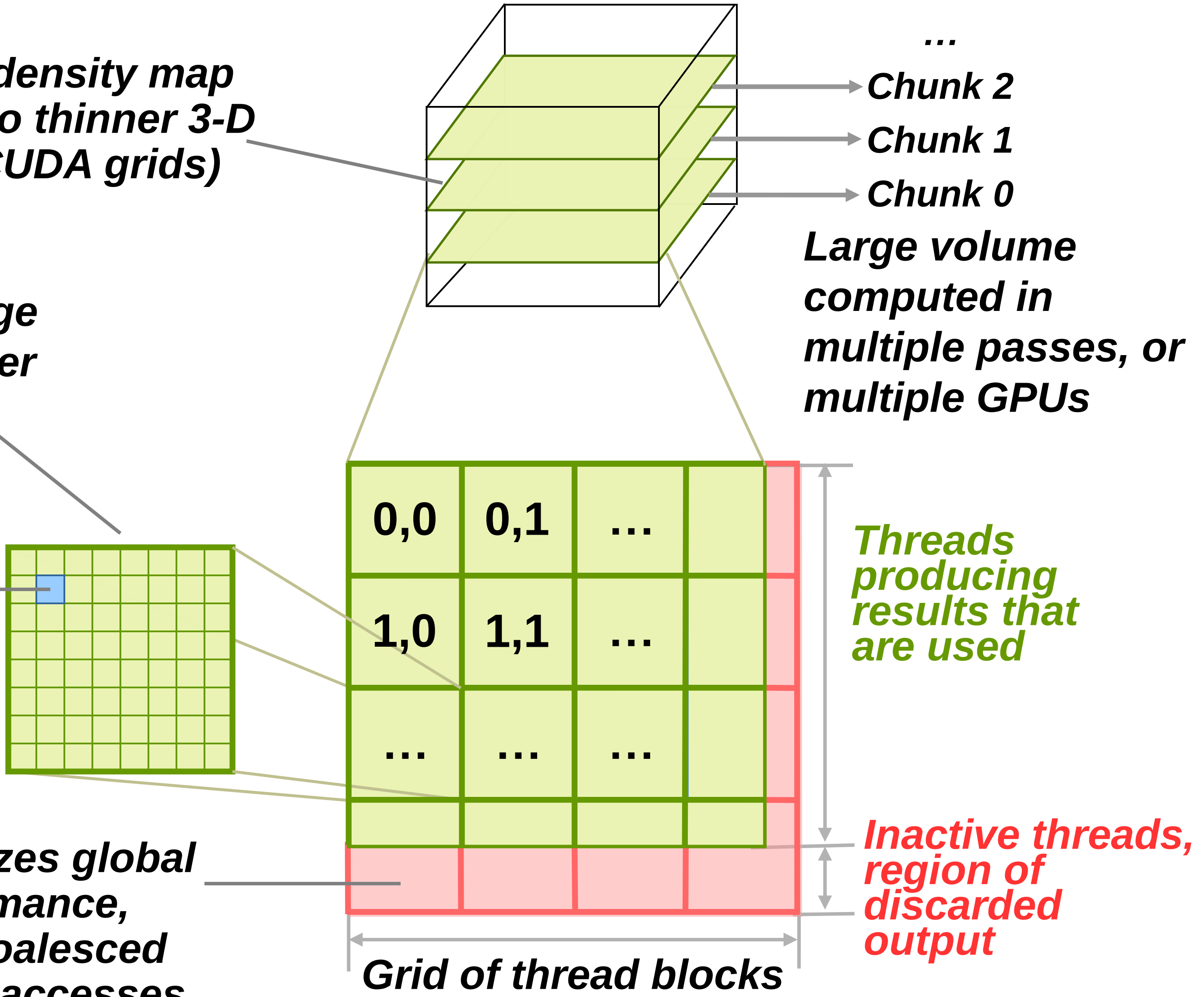


QuickSurf uniform grid spatial subdivision data structure

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


…
Chunk 2
Chunk 1
Chunk 0
0,0
0,1
…
1,0
1,1
…
…
…
…
Threads producing results that are used
Inactive threads, region of discarded output
Grid of thread blocks

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

Large volume computed in multiple passes, or multiple GPUs

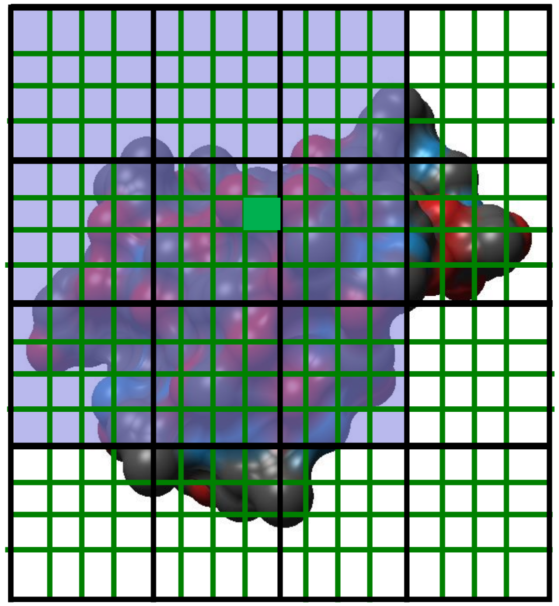
Small 8x8 thread blocks afford large per-thread register count, shared memory

Each thread computes one or more density map lattice points

Padding optimizes global memory performance, guaranteeing coalesced global memory accesses

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

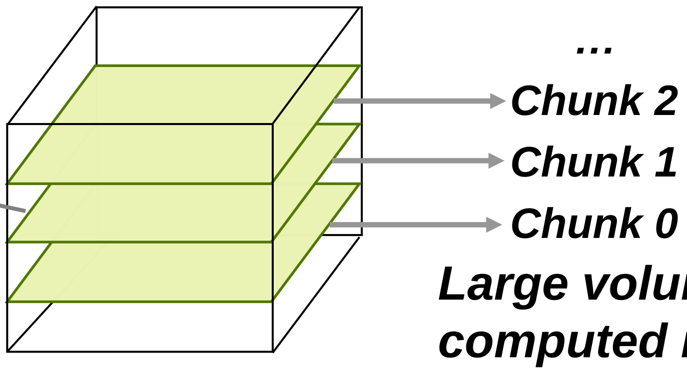
- Compute reciprocals, prefactors, other math on the host CPU prior to kernel launch
- Use of **intN** and **floatN** vector types in CUDA kernels for improved global memory bandwidth
- **Thread coarsening**: one thread computes multiple output densities and colors
- Input data and **register tiling**: 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

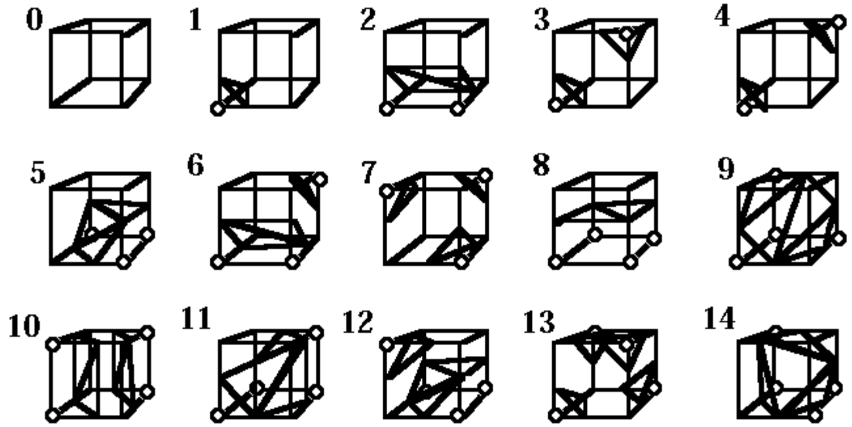


QuickSurf 3-D density map decomposes into thinner 3-D slabs/slices (CUDA grids)

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**	**T_{sort}**	**$T_{density}$**	**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.

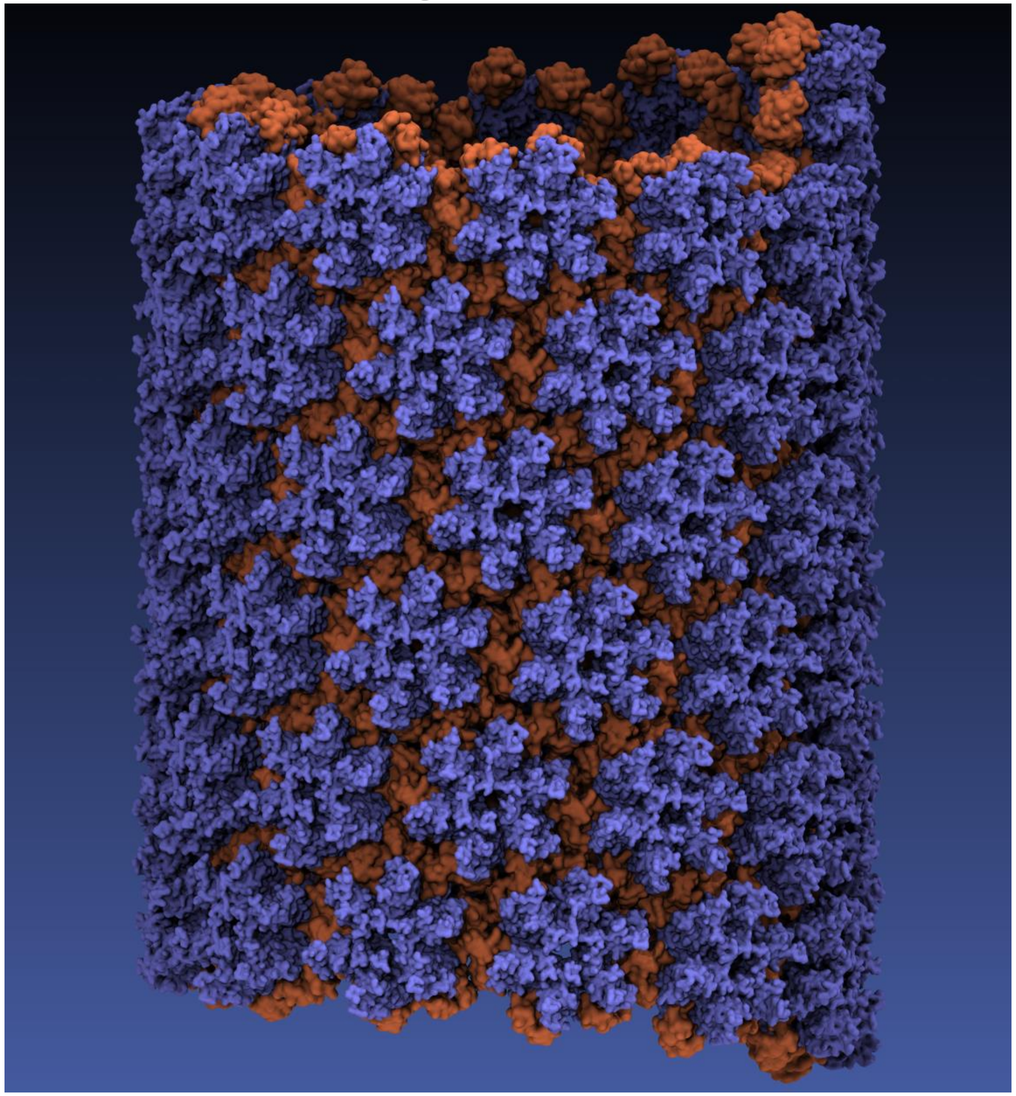
M. Krone, J. E. Stone, T. Ertl, K. Schulten. *EuroVis Short Papers*, pp. 67-71, 2012

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) {
```

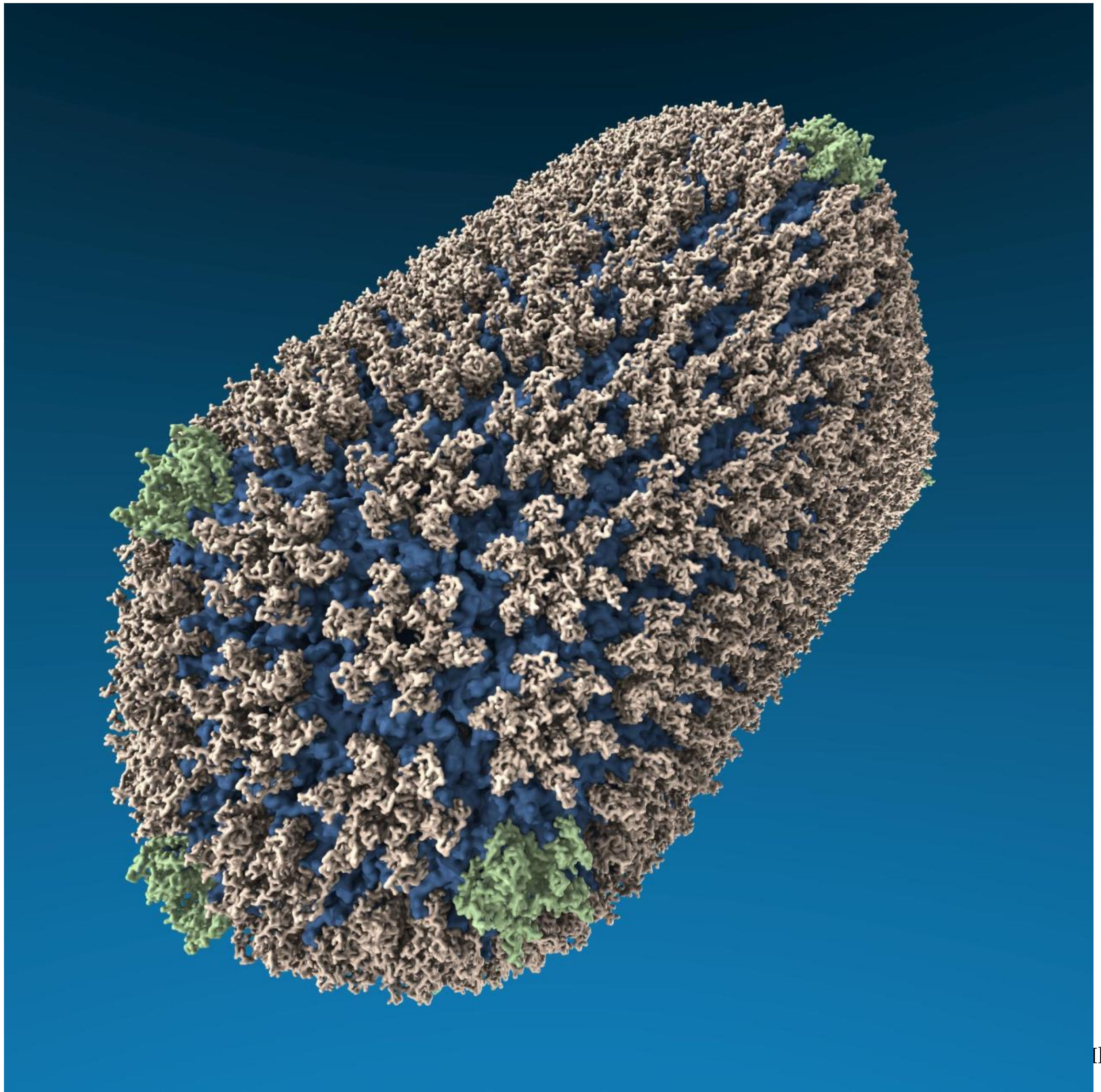
Example Templated Density Map Kernel

```
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!

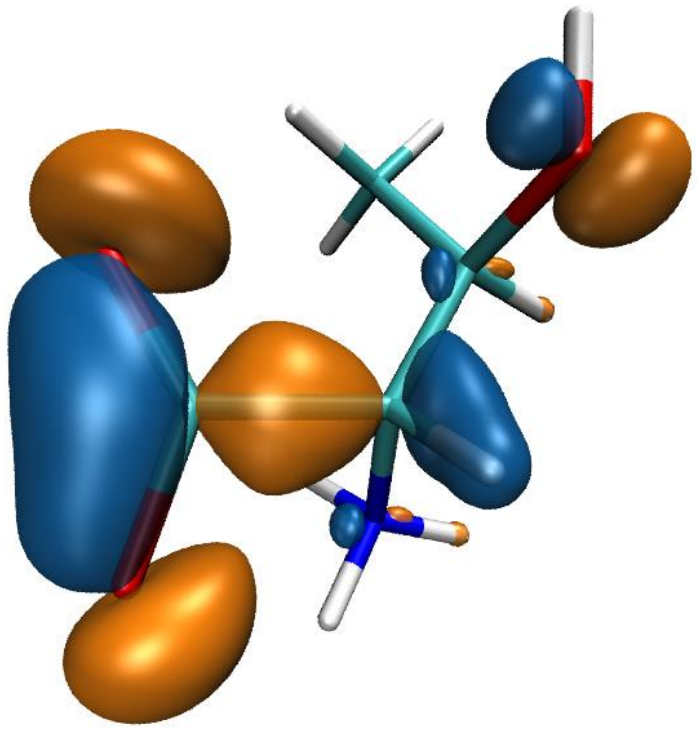
High Resolution HIV Surface



Beckman Institute,
llinois at Urbana-Champaign

Molecular Orbitals

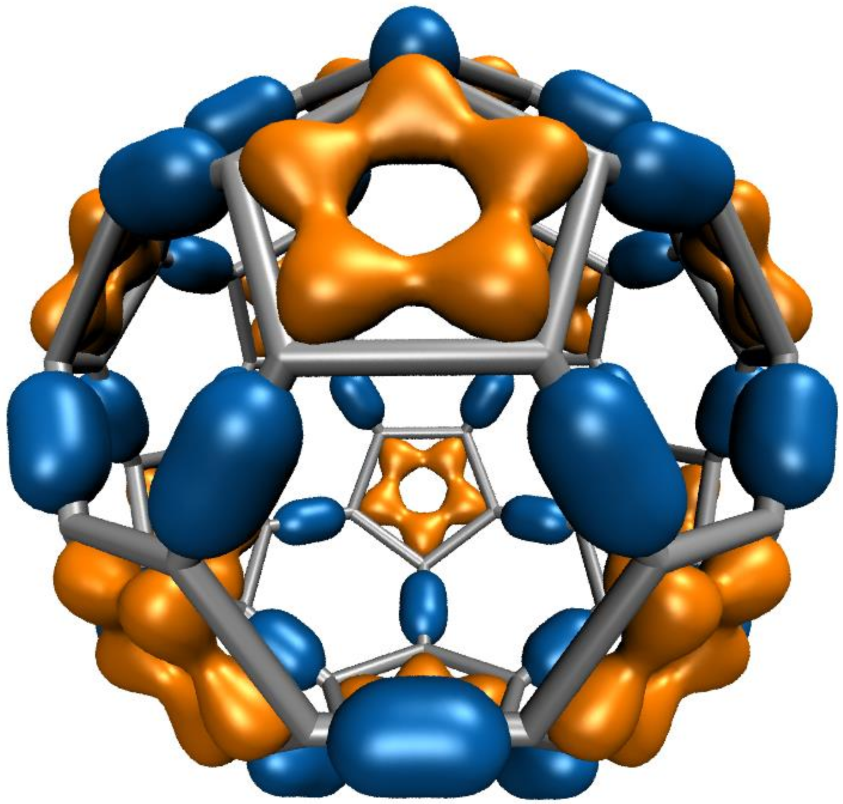
- Visualization of MOs aids in understanding the chemistry of molecular system
- MO spatial distribution is correlated with probability density for an electron(s)
- Algorithms for computing other molecular properties are similar, and can share code



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.

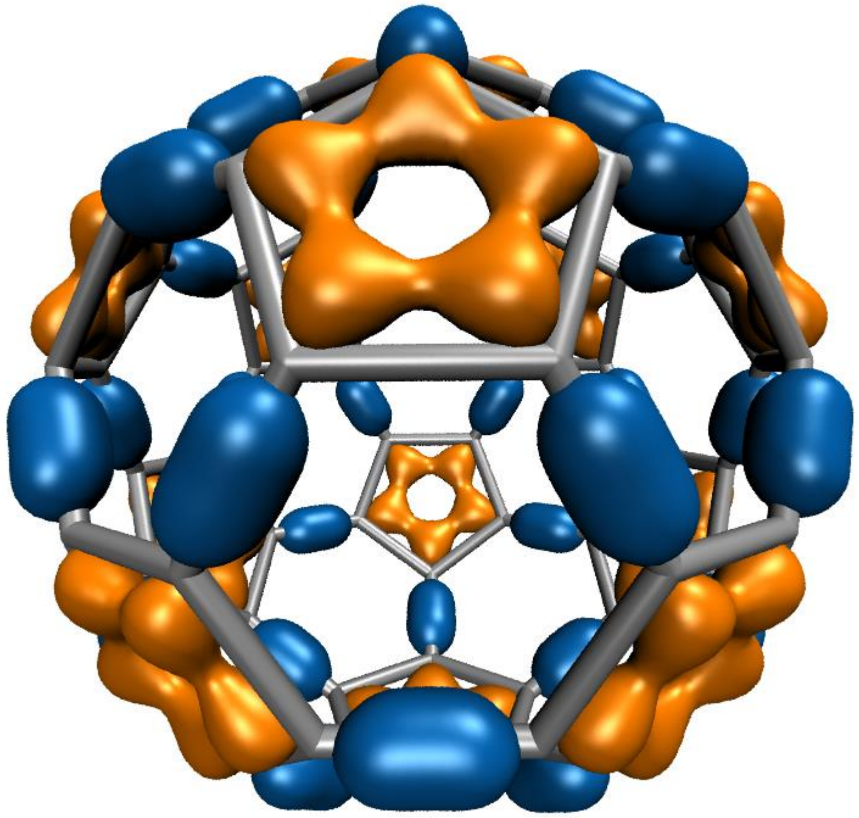
Computing Molecular Orbitals

- 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

Read QM simulation log file, trajectory

Initialize Pool of GPU Worker Threads

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

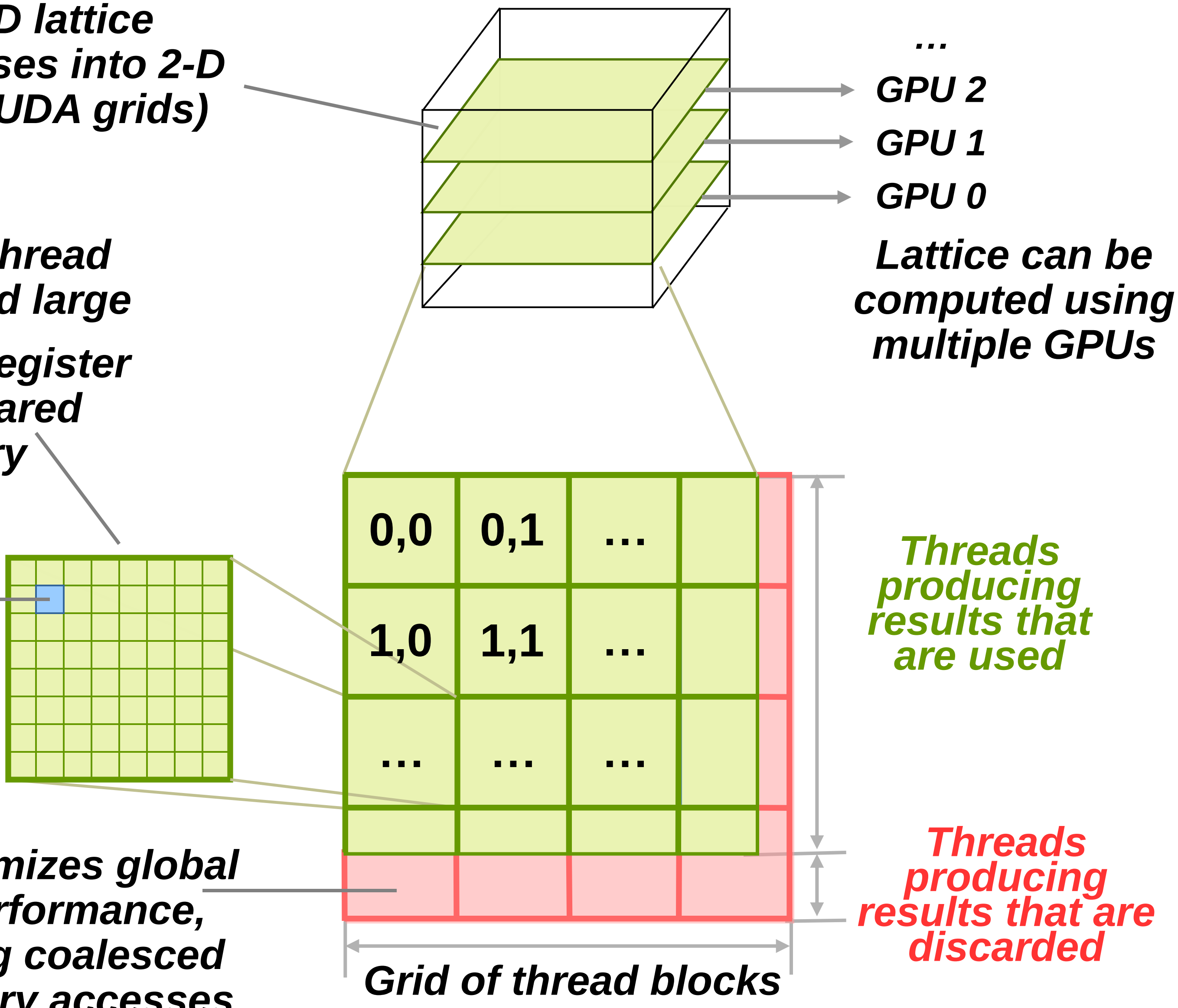
MO 3-D lattice decomposes into 2-D slices (CUDA grids)

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

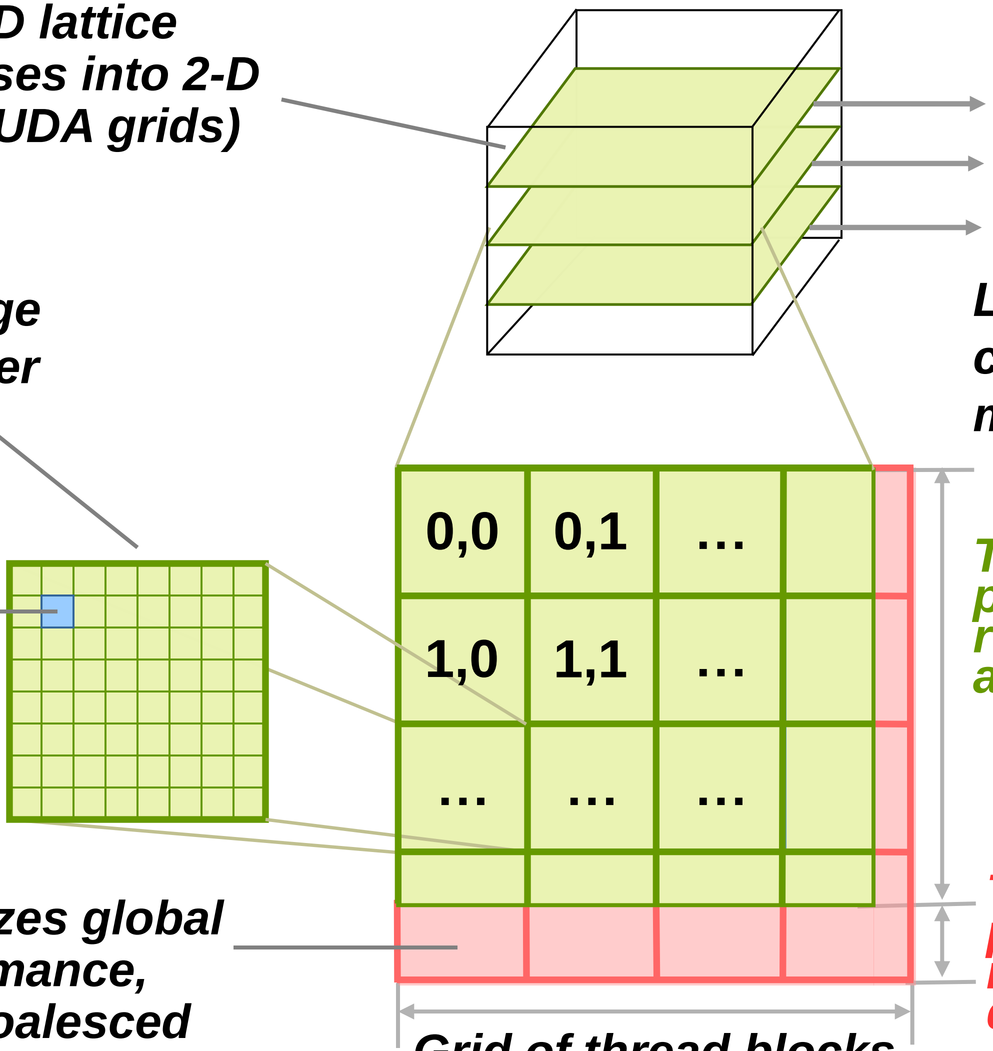
Each thread computes one MO lattice point.

Padding optimizes global memory performance, guaranteeing coalesced global memory accesses

Lattice can be computed using multiple GPUs


…
GPU 2
GPU 1
GPU 0
0,0
0,1
…
1,0
1,1
…
…
…
…
Threads producing results that are used
Threads producing results that are discarded
Grid of thread blocks

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

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 the 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
```

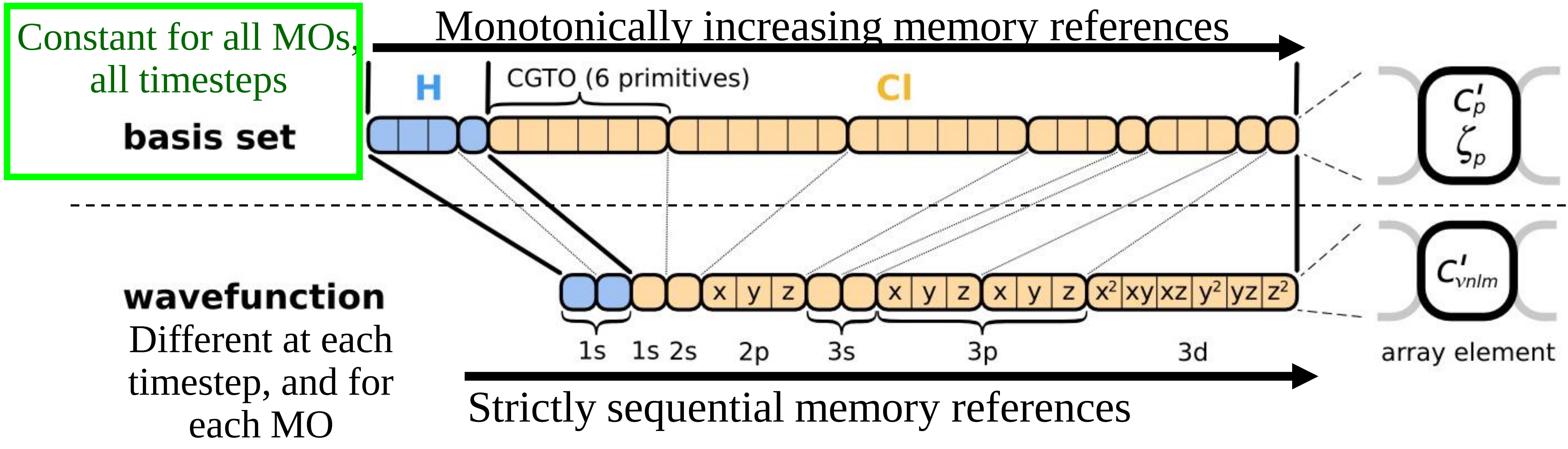
Loop unrolling:

- 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

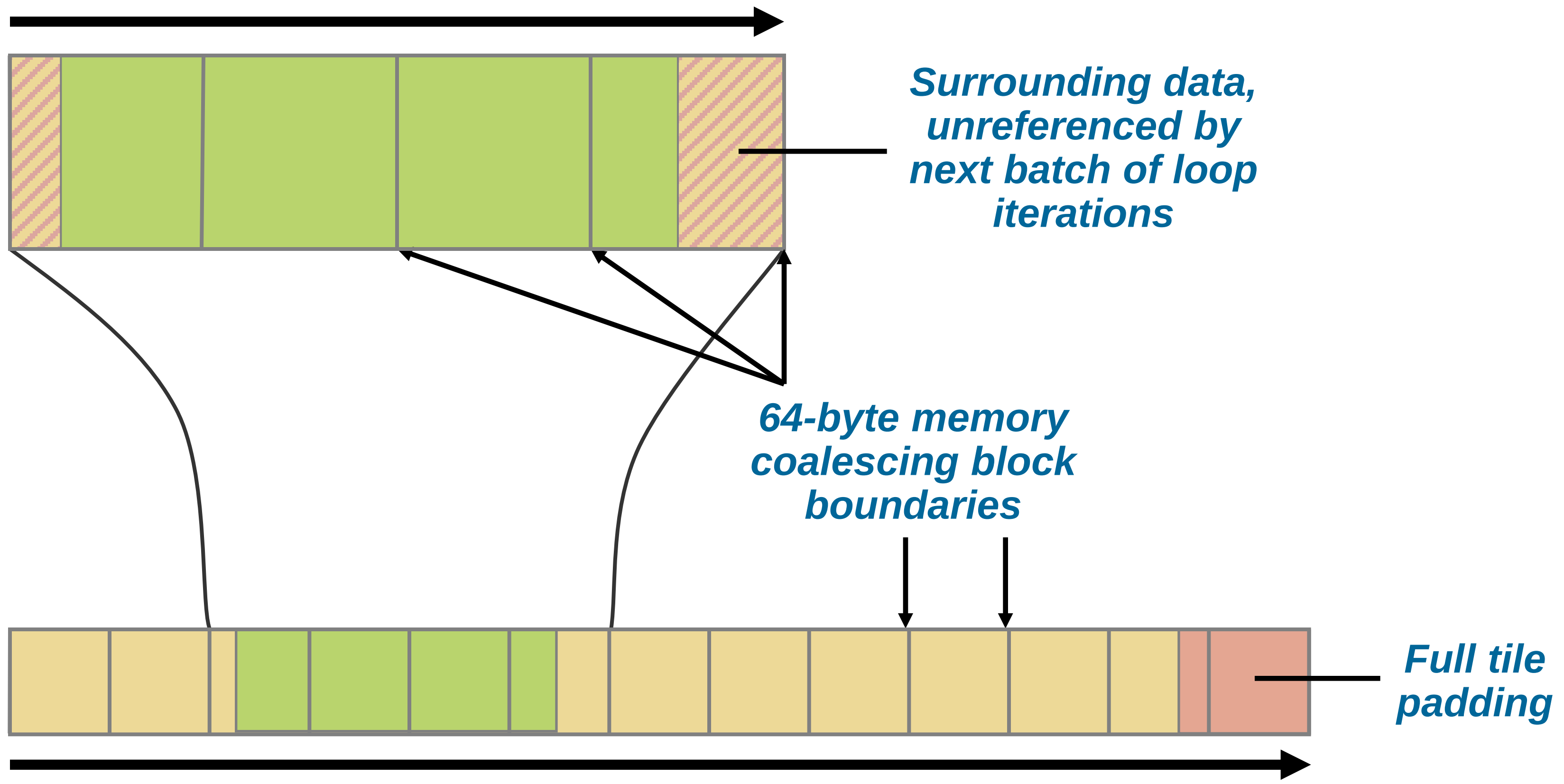

Constant for all MOs, all timesteps
basis set
Monotonically increasing memory references
H
CGTO (6 primitives)
Cl
C'_p
ζ_p
wavefunction
Different at each timestep, and for each MO
x y z x y z x y z
x^2 xy xz y^2 yz z^2
1s 1s 2s 2p 3s 3p 3d
$C'_{\nu nlm}$
array element
Strictly sequential memory references

- 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" and "Kepler" 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:

Fermi/Kepler kernel based on L1 cache

```
[… 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

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)

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)

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 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 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.
CUDA-const-cache-JIT*	**1**	**0.27**	**173.** **(JIT 40% faster)**

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.

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

**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 L1-cache (16KB)	**1**	**0.228**	**134**
CUDA const-cache	1	0.104	292
CUDA const-cache, zero-copy	1	0.0938	326

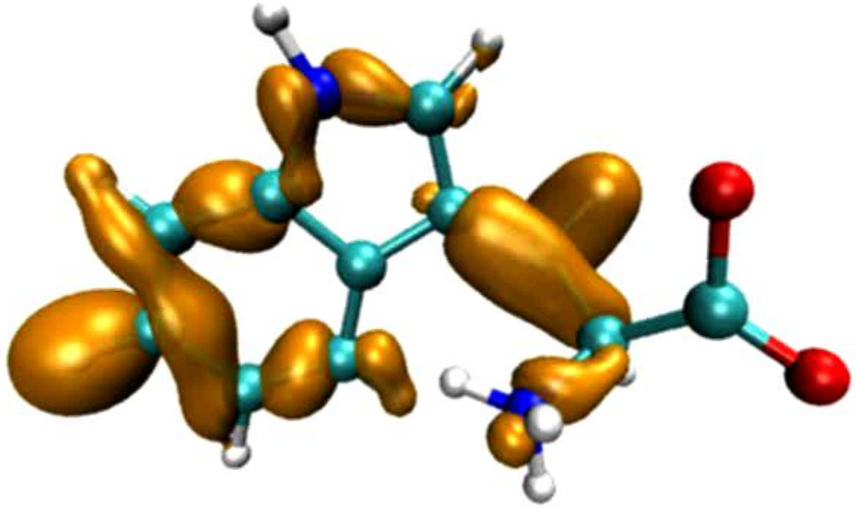
Kepler GK104 (GeForce 680) seems to strongly prefer the constant cache kernels vs. the others.

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**

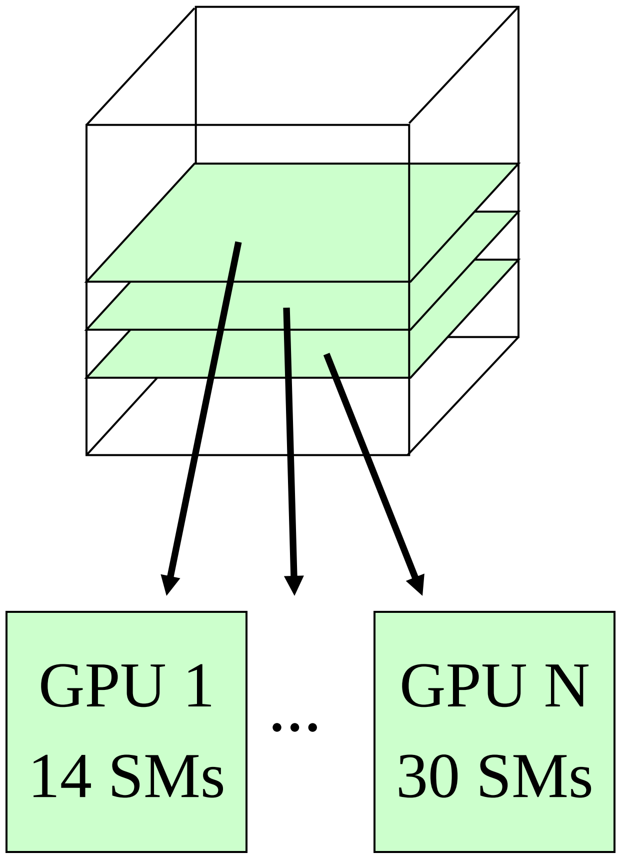

threonine

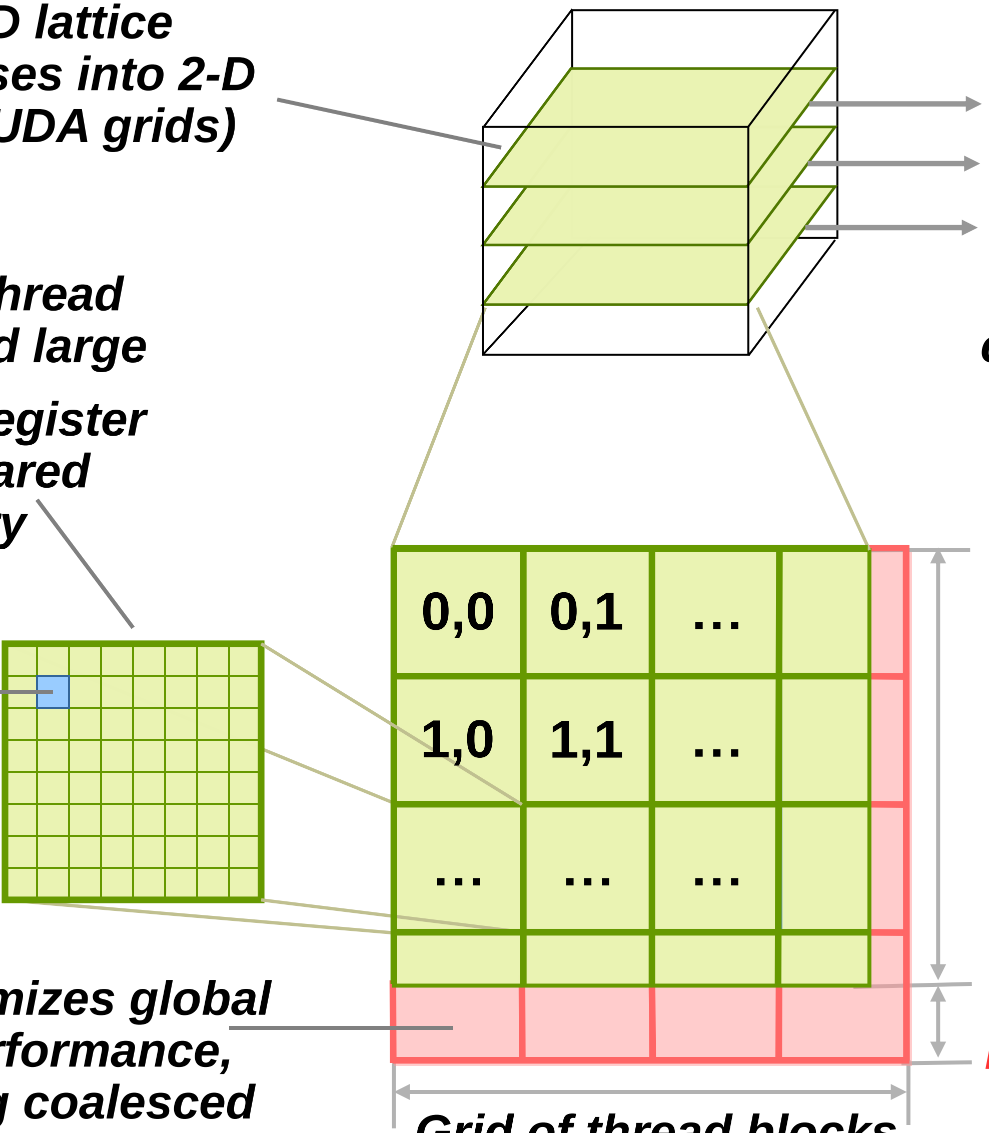
With GPU speedups over **100x**, previously insignificant CPU surface gen, gradient calc, and rendering are now **66%** of runtime.

Need GPU-accelerated surface gen next…

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 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.

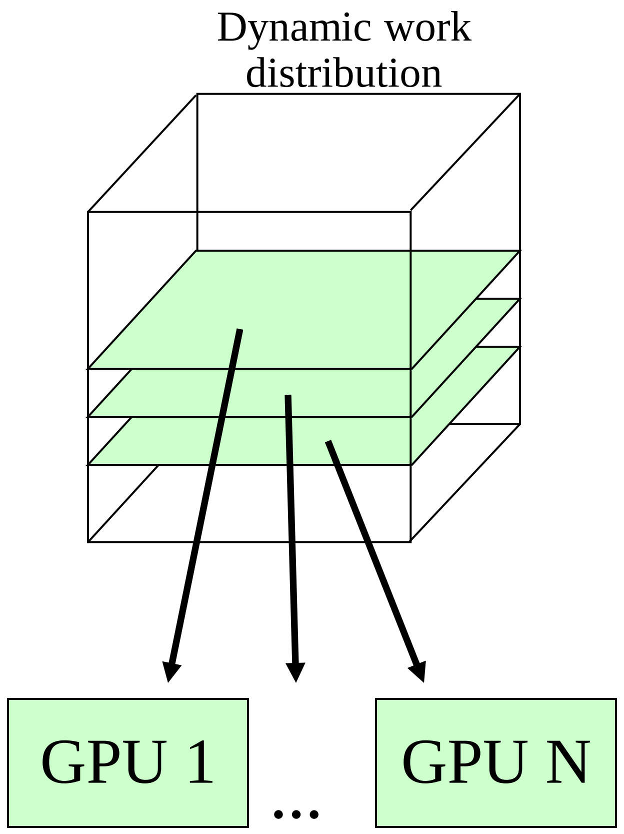
Threads producing results that are used

Threads producing results that are discarded

Padding optimizes global memory performance, guaranteeing coalesced global memory accesses

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
}
```



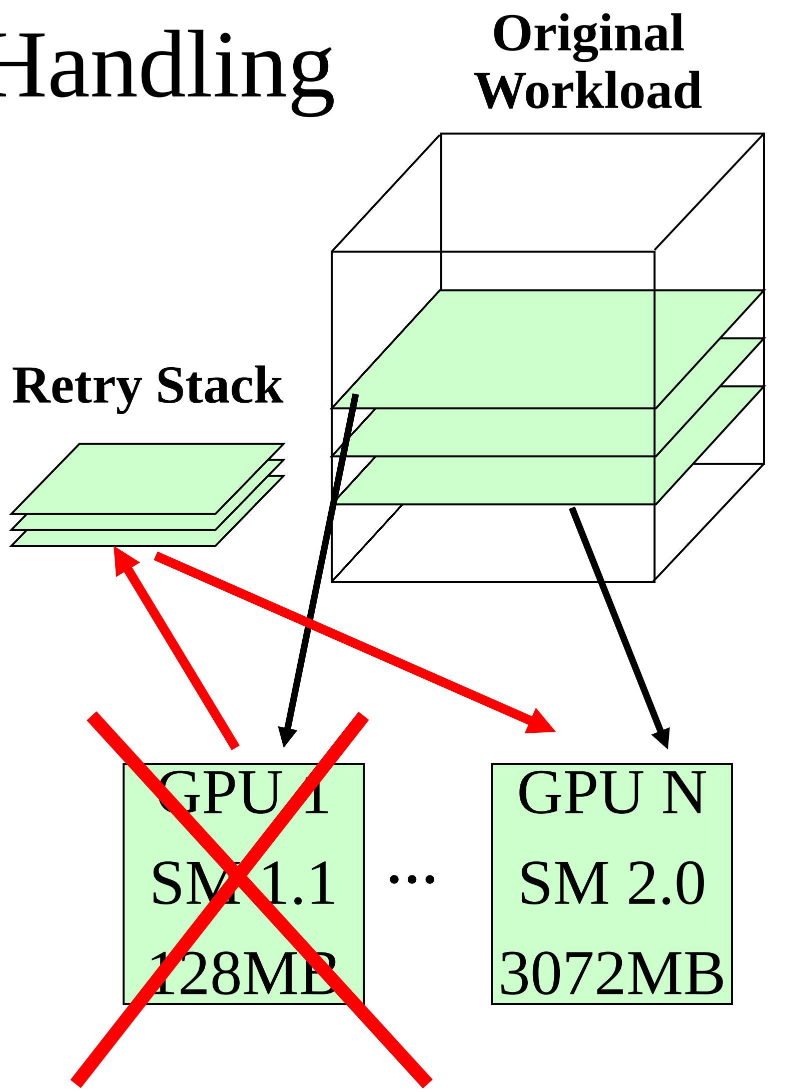
Example Multi-GPU Latencies

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

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 + Tesla C2050 + Quadro 5800	3	0.114	268 (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) {
        }
        Loop over angular momenta for this shell type (1 to ~15) {}
    }
}
```

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**

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

General loop-based CUDA kernel

```
…..
// 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;
```

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;
```

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

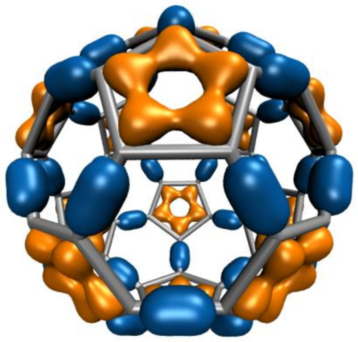
Experiments Porting VMD CUDA Kernels to OpenCL

- 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!)

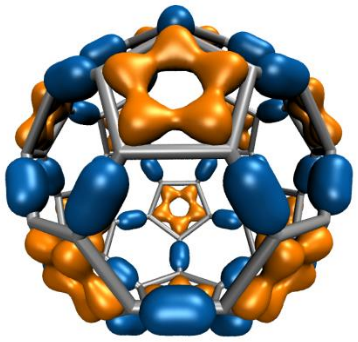
```
for (shell=0; shell < maxshell; shell++) {
   __m128 Cgto = _mm_setzero_ps();
   for (prim=0; prim<num_prim_per_shell[shell_counter]; prim++) {
      float exponent       = -basis_array[prim_counter    ];
      float contract_coeff =  basis_array[prim_counter + 1];
      __m128 expval = _mm_mul_ps(_mm_load_ps1(&exponent), 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...

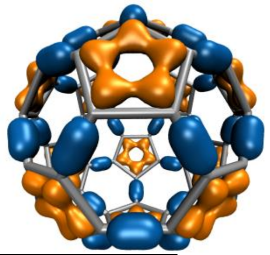
Molecular Orbital Inner Loop, OpenCL Vec4 Ahhh, much easier to read!!!

```
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.

Acknowledgements

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

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

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

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

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

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

- **GPU Clusters for High Performance Computing**. V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, W. Hwu. *Workshop on Parallel Programming on Accelerator Clusters (PPAC),* In Proceedings IEEE Cluster 2009, pp. 1-8, Aug. 2009.
- **Long time-scale simulations of in vivo diffusion using GPU hardware**. E. Roberts, J. Stone, L. Sepulveda, W. Hwu, Z. Luthey-Schulten. In *IPDPS'09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Computing*, pp. 1-8, 2009.
- **High Performance Computation and Interactive Display of Molecular Orbitals on GPUs and Multi-core CPUs**. J. Stone, J. Saam, D. Hardy, K. Vandivort, W. Hwu, K. Schulten, *2nd Workshop on General-Purpose Computation on Graphics 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.