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

John Stone, Jan Saam, David Hardy,

Kirby Vandivort, Wen-mei Hwu, Klaus Schulten

John Stone

Senior Research Programmer

Beckman Institute for Advanced Science and Technology

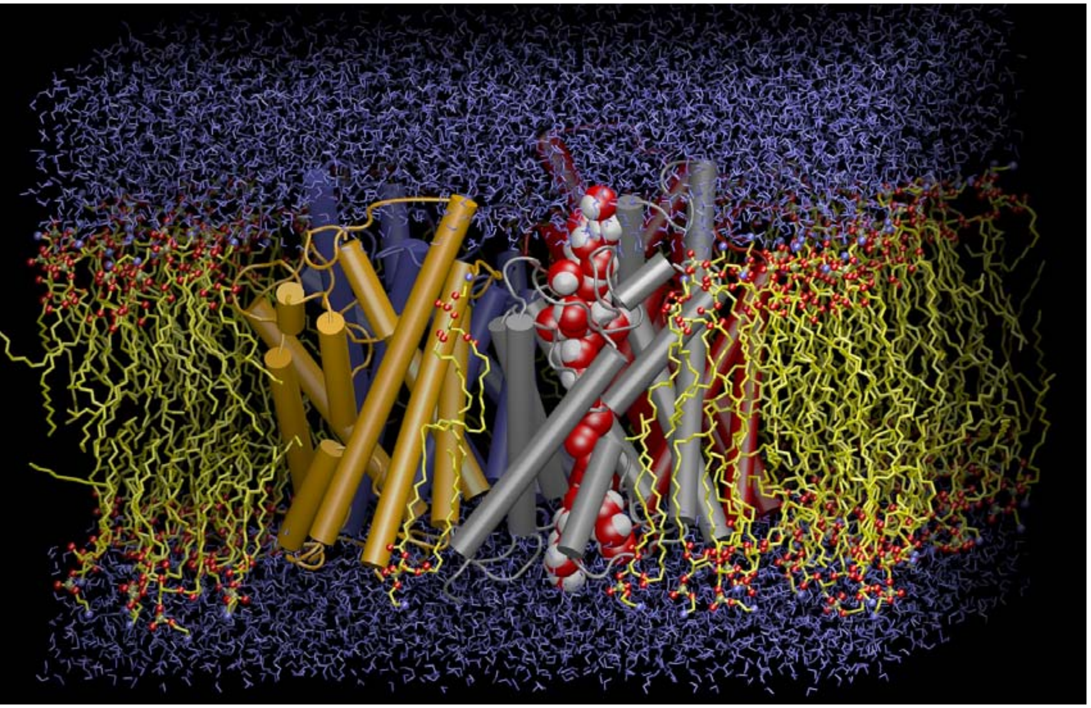
University of Illinois at Urbana-Champaign

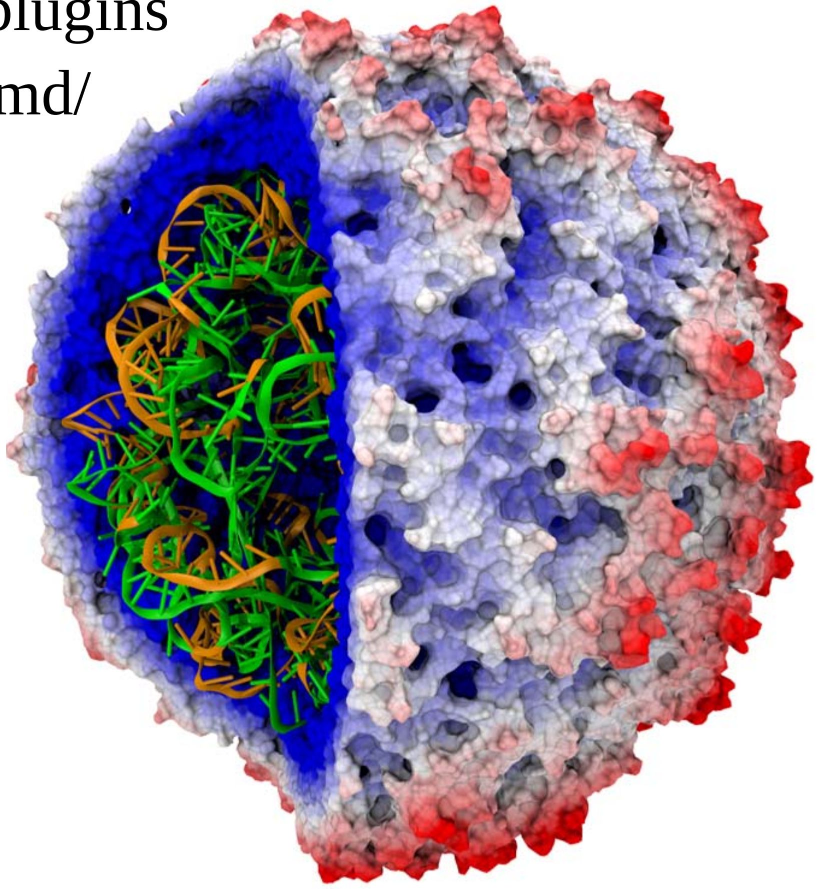
http://www.ks.uiuc.edu/Research/gpu/

Second GPGPU Workshop, March 8, 2009

VMD – "Visual Molecular Dynamics"

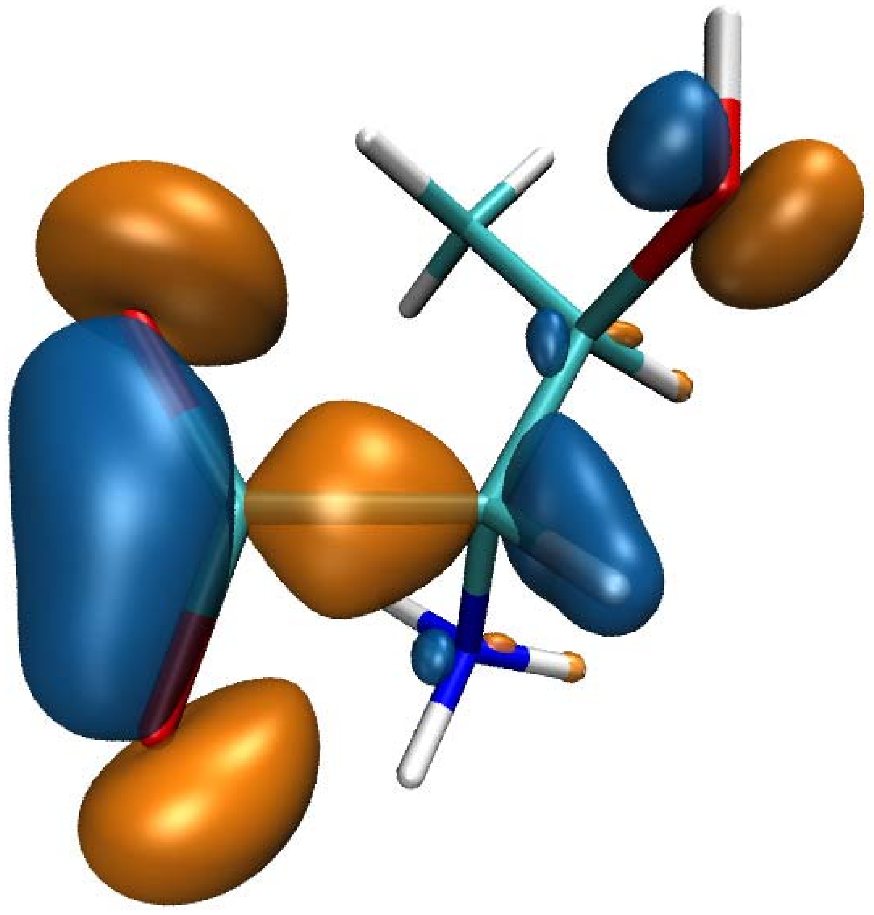
- Visualization and analysis of molecular dynamics simulations, sequence data, volumetric data, **quantum chemistry data**, particle systems
- User extensible with scripting and plugins
- http://www.ks.uiuc.edu/Research/vmd/





Molecular Orbitals

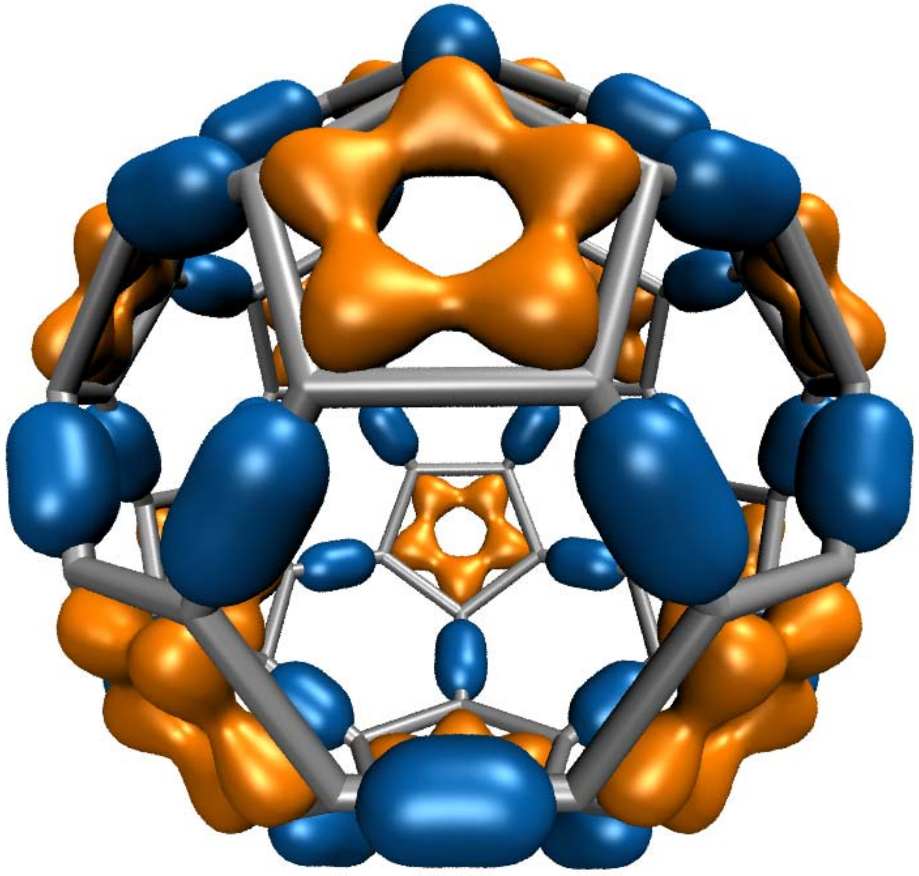
- Visualization of MOs aids in understanding the chemistry of molecular system
- MO spatial distribution is correlated with probability density for an electron



threonine

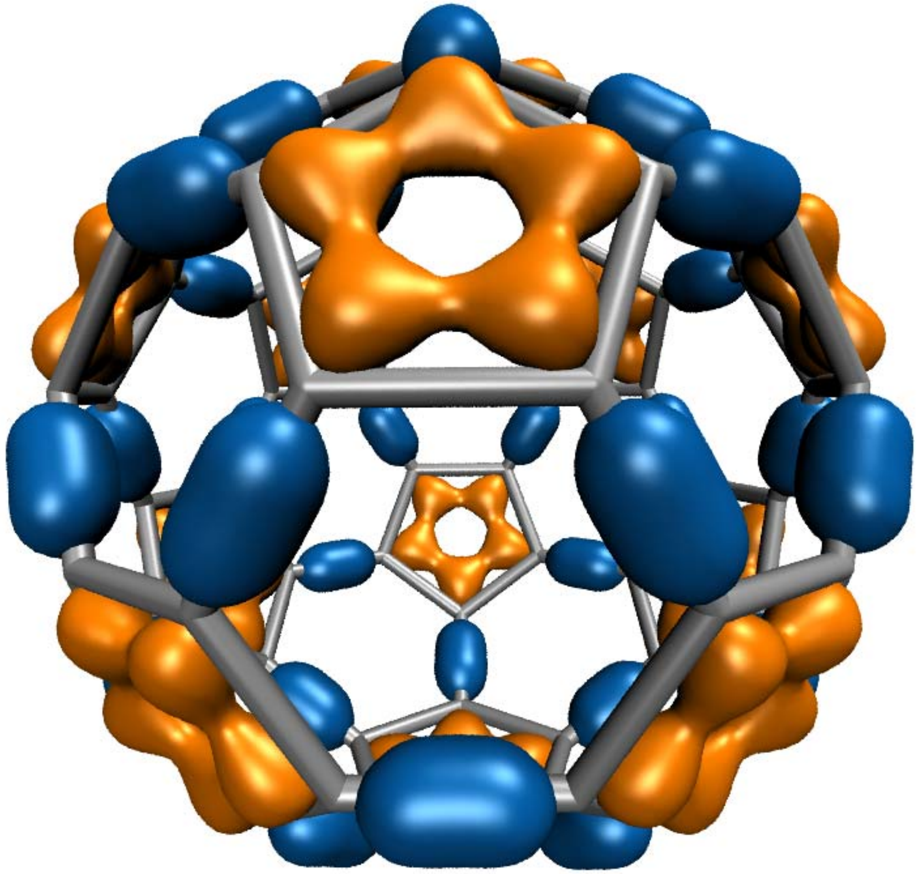
Computing Molecular Orbitals

- 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}

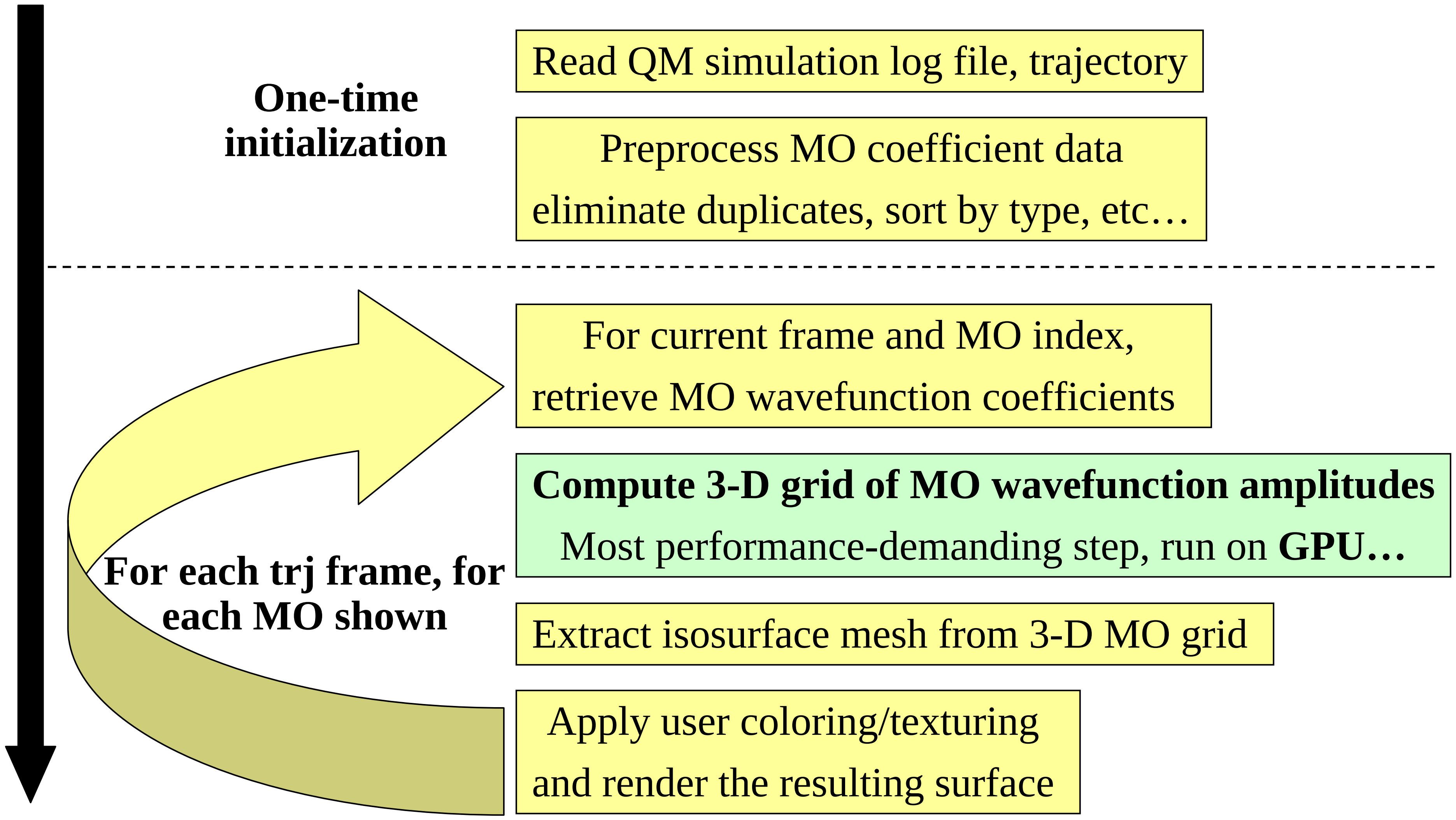
Animating Molecular Orbitals

- 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

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

For each trj frame, for each MO shown

For current frame and MO index,
retrieve MO wavefunction coefficients

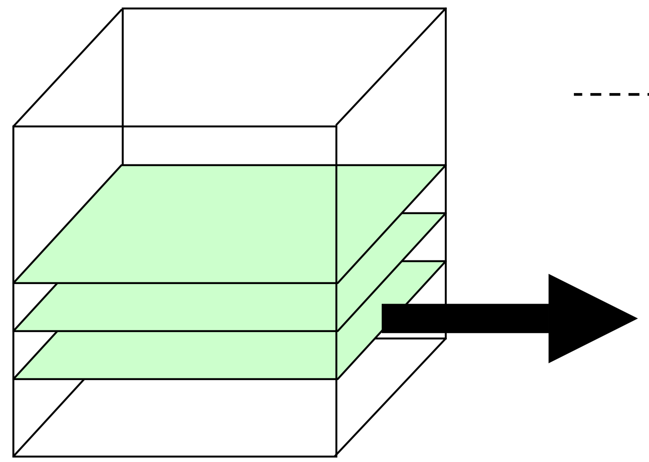
Compute 3-D grid of MO wavefunction amplitudes
Most performance-demanding step, run on **GPU…**

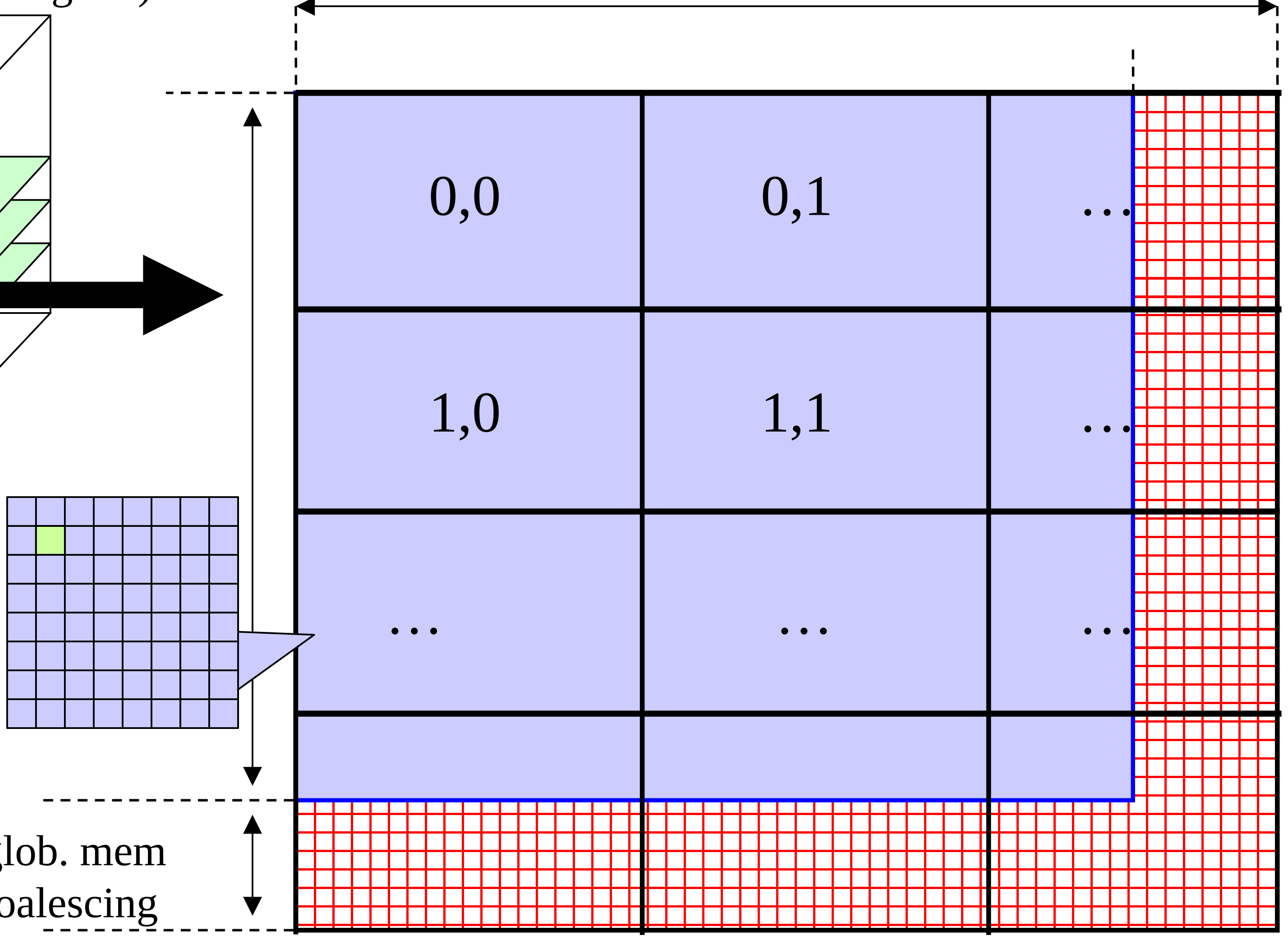
Extract isosurface mesh from 3-D MO grid

Apply user coloring/texturing
and render the resulting surface

CUDA Block/Grid Decomposition

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





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

Padding optimizes glob. mem perf, guaranteeing coalescing

MO Kernel for One Grid Point (Naive C)

```
…
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

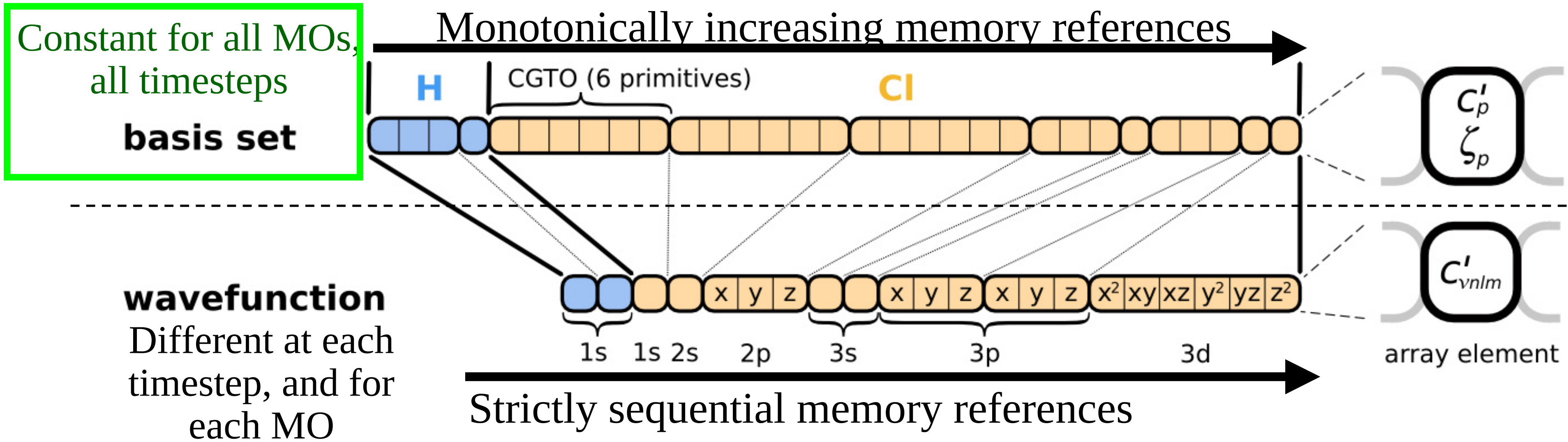
(unrolled in real code)

Preprocessing of Atoms, Basis Set, and Wavefunction Coefficients

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

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

Constant for all MOs, all timesteps


Monotonically increasing memory references
basis set
H
CGTO (6 primitives)
Cl
c'_p
ζ_p
wavefunction
x y z x y z x y z x² xy xz y² yz z²
1s 1s 2s 2p 3s 3p 3d
$c'_{\nu nlm}$
array element
Strictly sequential memory references

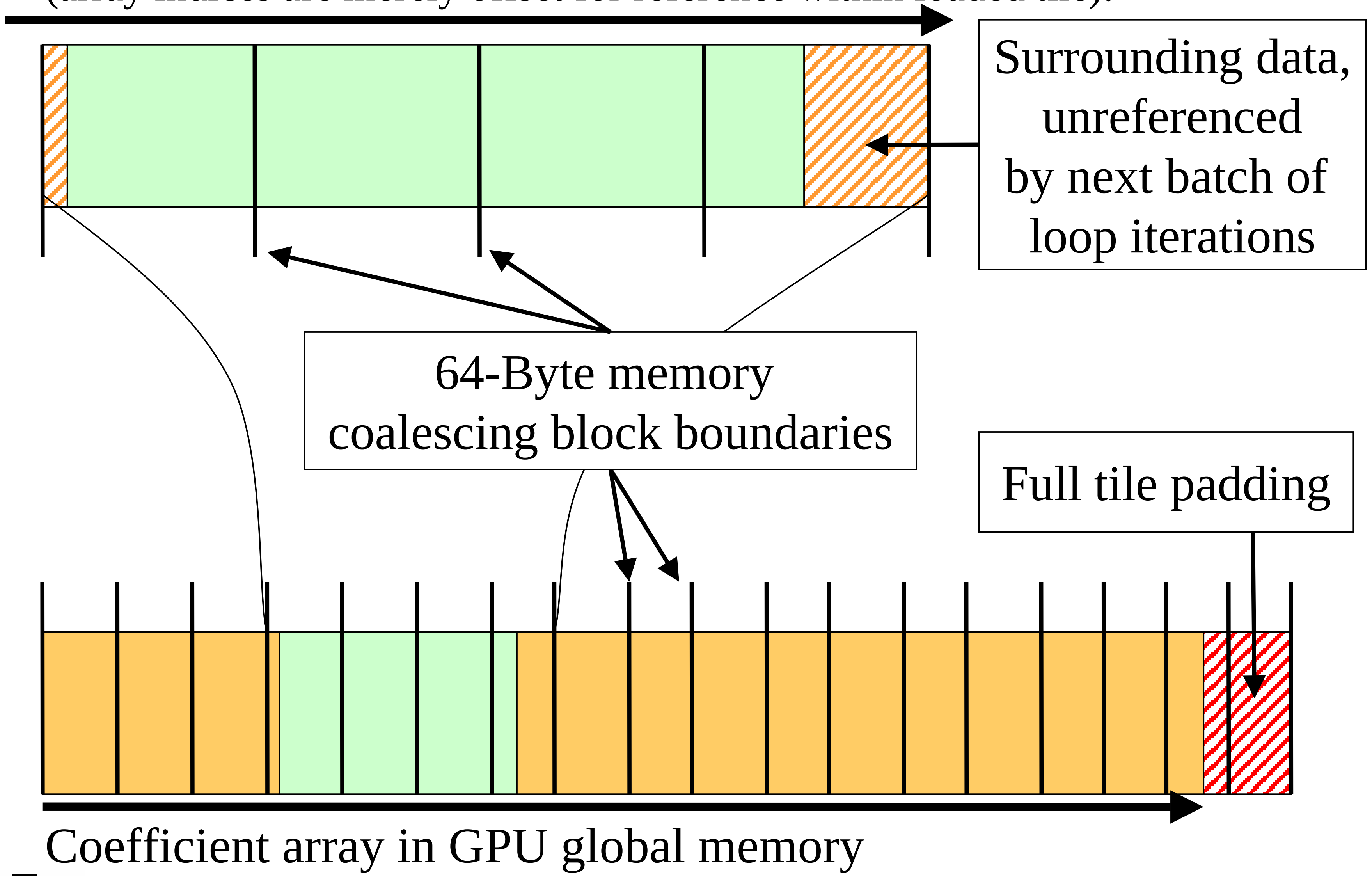
Different at each timestep, and for each MO

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

Use of GPU On-chip Memory

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

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


Surrounding data, unreferenced by next batch of loop iterations
64-Byte memory coalescing block boundaries
Full tile padding
Coefficient array in GPU global memory

VMD MO Performance Results for C_{60}

Sun Ultra 24: Intel Q6600, NVIDIA GTX 280

Kernel	Cores/GPUs	Runtime (s)	Speedup
CPU ICC-SSE	1	46.58	1.00
CPU ICC-SSE	4	11.74	3.97
CPU ICC-SSE-approx**	4	3.76	12.4
CUDA-tiled-shared	1	0.46	100.
CUDA-const-cache	1	0.37	126.
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

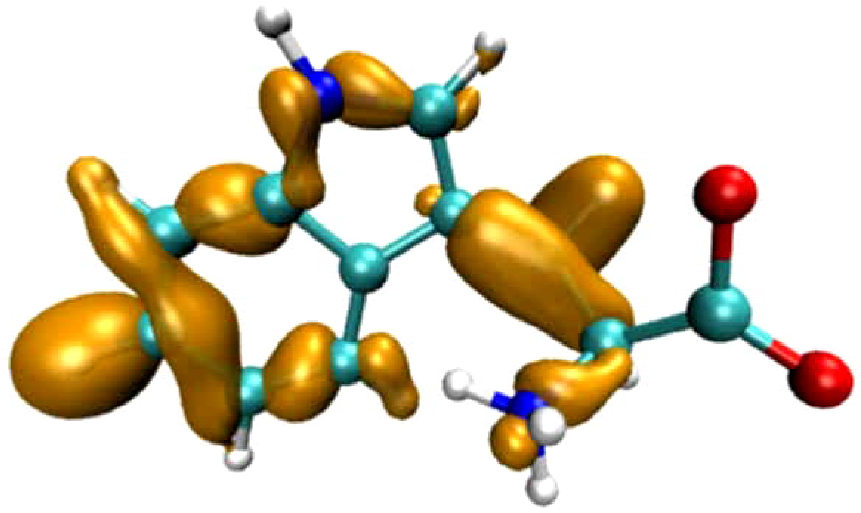
**Reduced-accuracy approximation of expf(), cannot be used for zero-valued MO isosurfaces

VMD 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…

MO Kernel Structure, Opportunity for JIT…

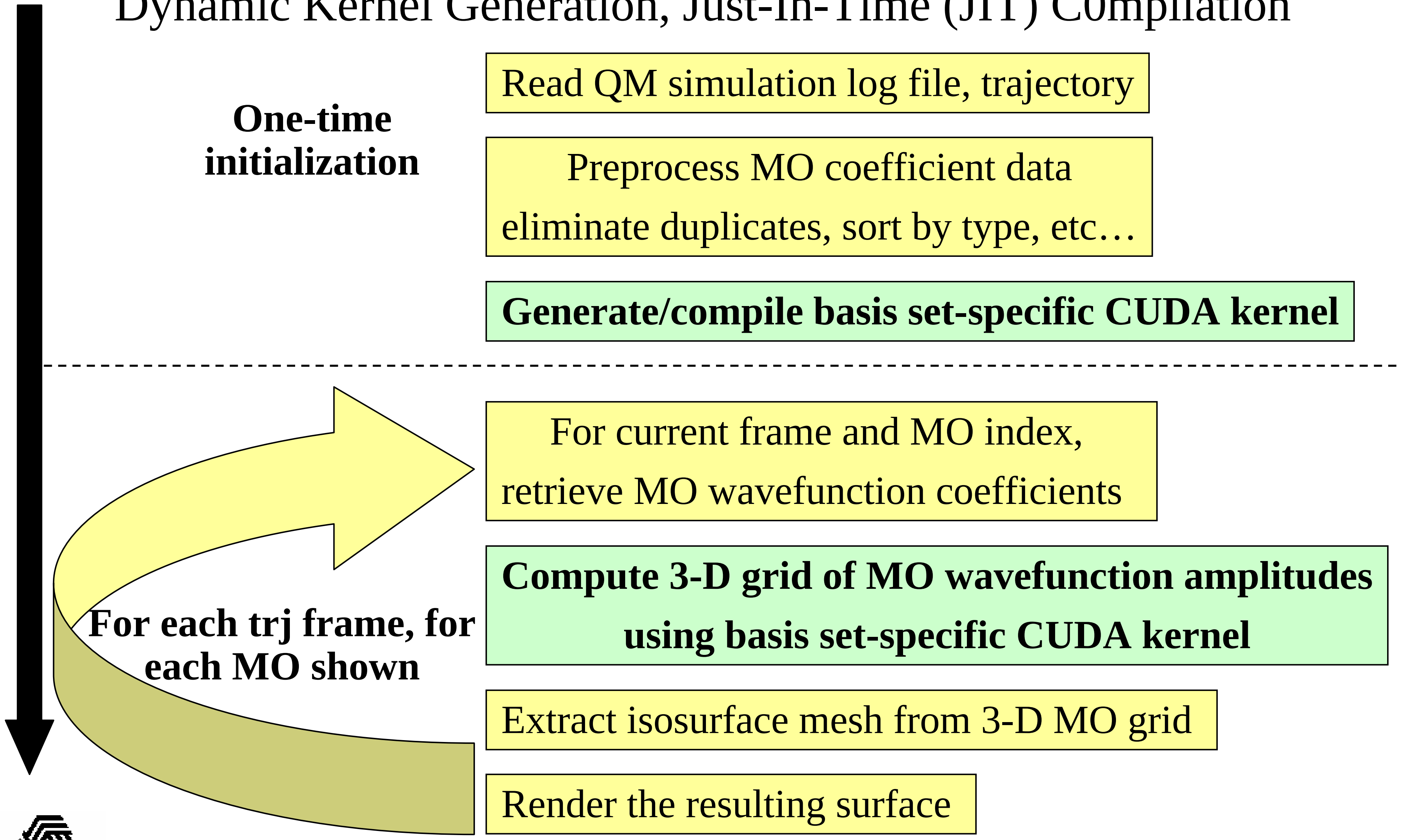
Data-driven, but representative loop trip counts in (…)

```
Loop over atoms (1 to ~200) {
    Loop over electron shells for this atom type (1 to ~6) {
        Loop over primitive functions for this shell type (1 to ~6) {
            Unpredictable (at compile-time, since data-driven ) but
            small loop trip counts result in significant loop overhead.
            Dynamic kernel generation and JIT compilation can
            eliminate this entirely, resulting in 40% speed boost
        }
        Loop over angular momenta for this shell type (1 to ~15) {}
    }
}
```

Dynamic kernel generation and JIT compilation can eliminate this 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…

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;
```

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

Future Work

- Tune Multi-GPU implementation to workaround small kernel launch delays that adversely impact animation speed
- Further development of runtime-generated MO kernels using new CUDA JIT compilation APIs
- Multi-pass approach with spatial decomposition and distance-based cutoff to truncate rapidly decaying exponentials (CPU+GPU cooperation)

Acknowledgements

- Theoretical and Computational Biophysics Group, IMPACT group, University of Illinois at Urbana-Champaign
- CUDA team at NVIDIA
- NIH support: P41-RR05969