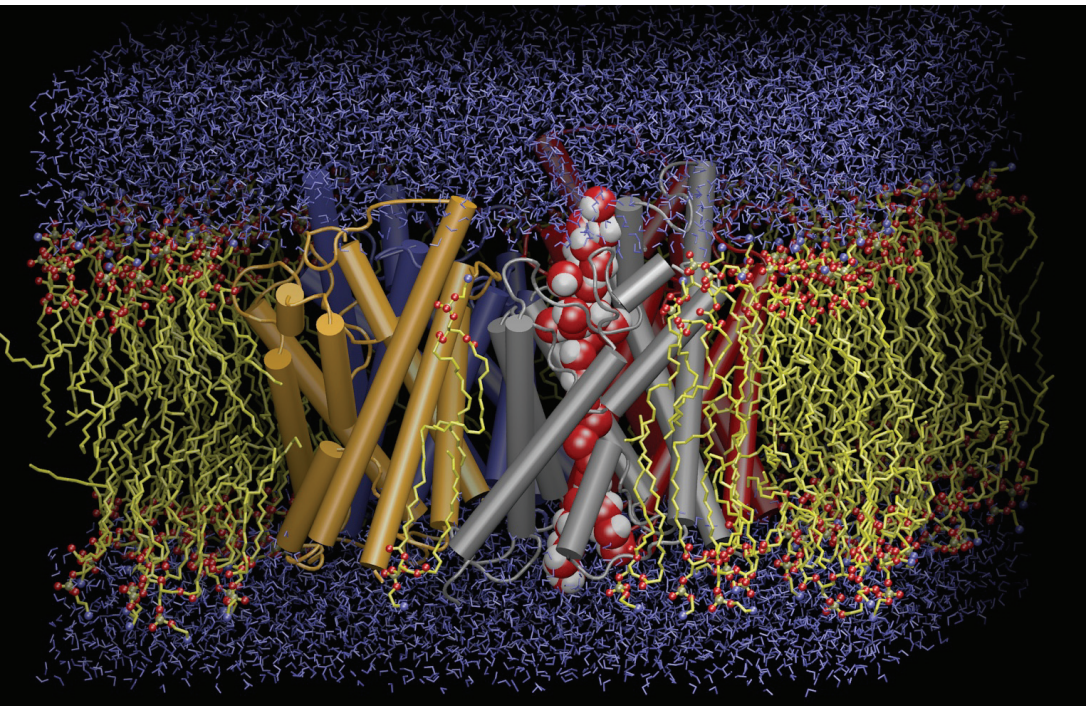
Single-node Multi-GPU Algorithms: Molecular Orbitals

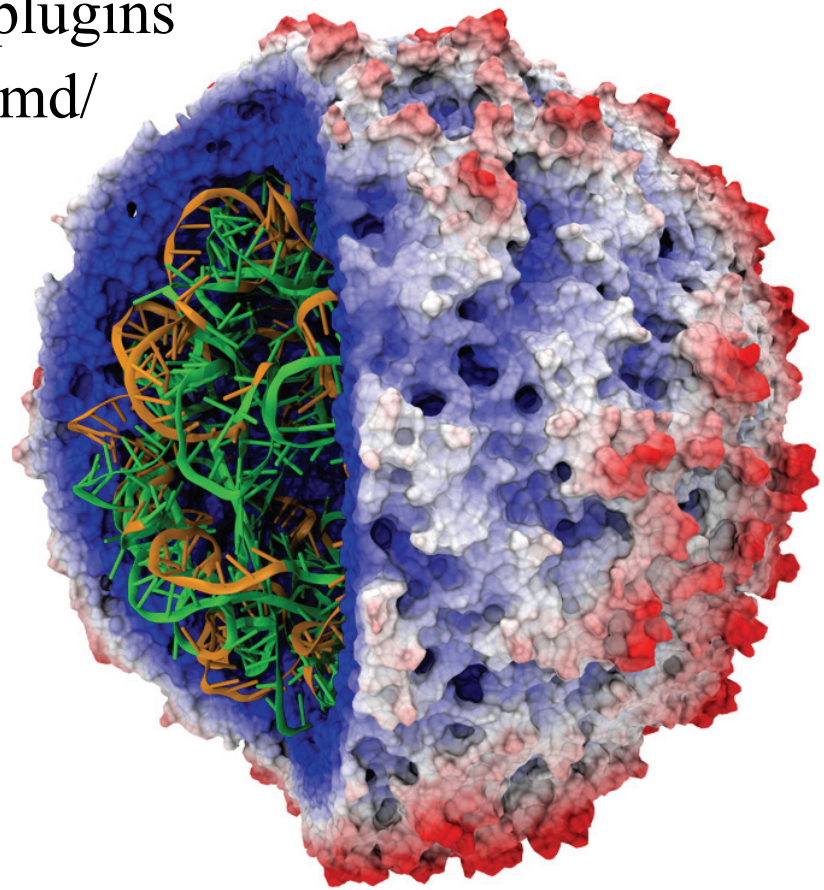
Theoretical and Computational Biophysics Group
Beckman Institute for Advanced Science and Technology
University of Illinois at Urbana-Champaign
http://www.ks.uiuc.edu/Research/gpu/

Workshop on GPU Programming for Molecular Modeling , Beckman Institute for Advanced Science and Technology, University of Illinois at Urbana-Champaign, August 8, 2010

VMD – "Visual Molecular Dynamics"

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

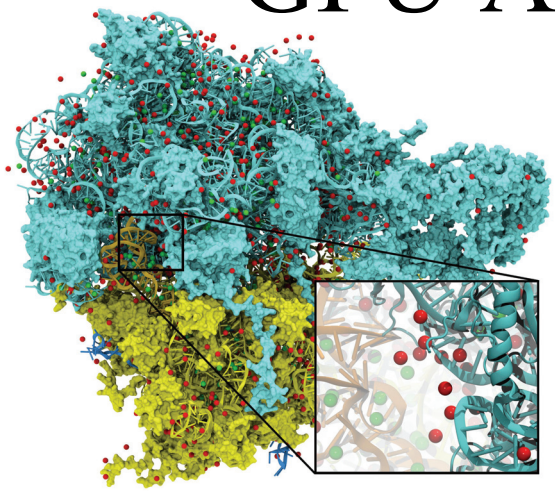




Motivation for GPU Acceleration in VMD

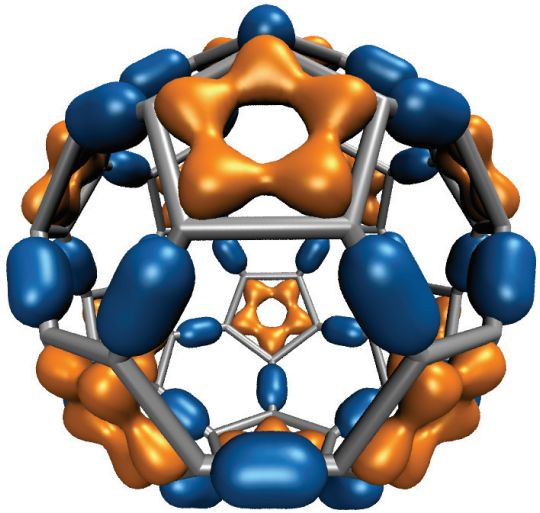
- Increases in supercomputing resources at NSF centers such as NCSA enable increased simulation complexity, fidelity, and longer time scales…
- Drives need for more visualization and analysis capability at the desktop and on clusters
- Desktop use is the most compute-limited scenario, where **GPUs can make a big impact**…
- GPU acceleration provides an opportunity to make some **slow, or batch** calculations capable of being run **interactively, or on-demand…**

GPU Acceleration in VMD 1.8.7



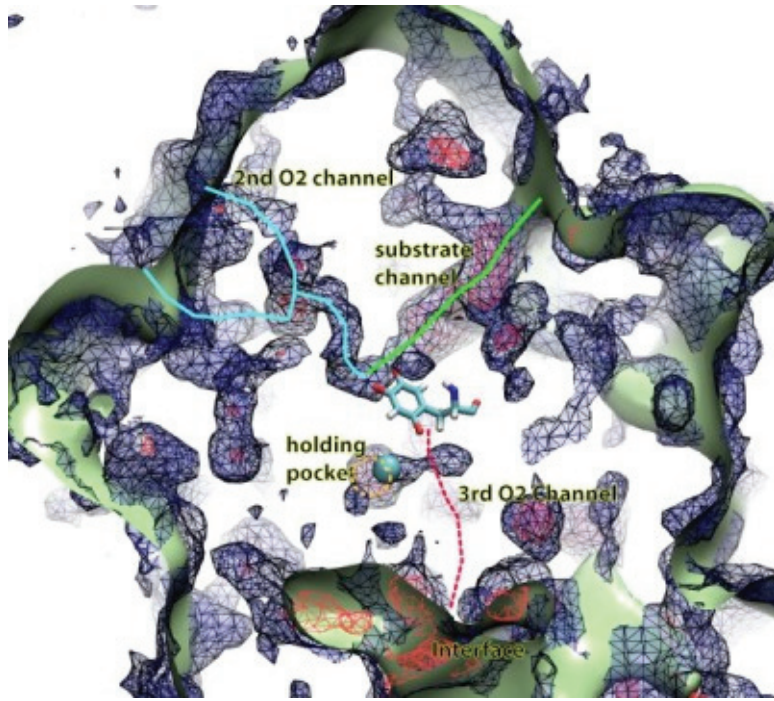
Electrostatic field calculation, ion placement

20x to 44x faster



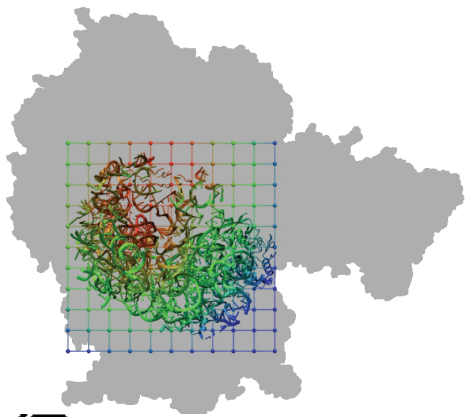
Molecular orbital calculation and display

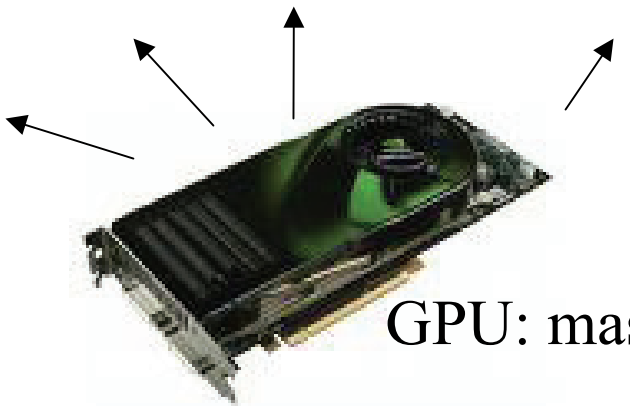
100x to 120x faster



Imaging of gas migration pathways in proteins with implicit ligand sampling

20x to 30x faster





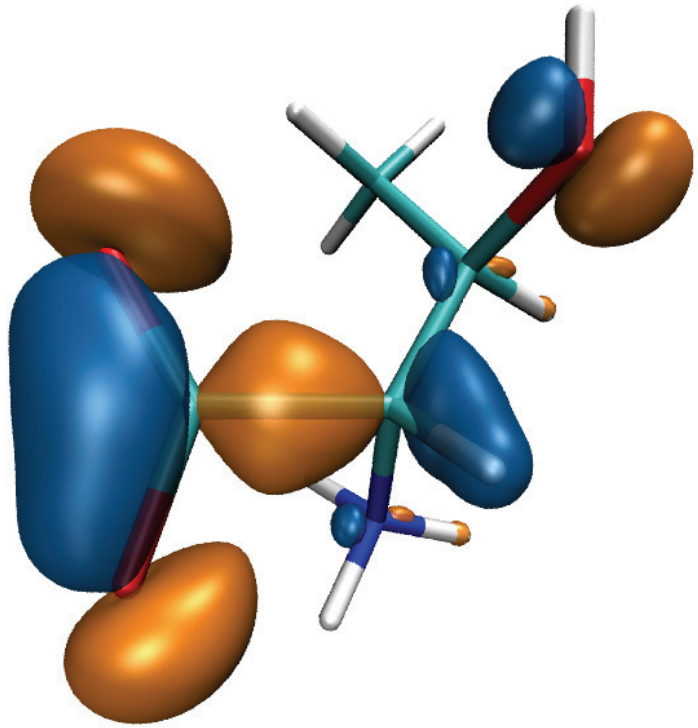
GPU: massively parallel co-processor

Ongoing VMD GPU Development

- Support for CUDA in MPI-enabled builds of VMD for analysis runs on GPU clusters
- Updating existing CUDA kernels to take advantage of new hardware features on the latest "Fermi" GPUs
- Adaptation of CUDA kernels to OpenCL, evaluation of JIT techniques with OpenCL
- Development of new CUDA kernels for computing radial distribution functions, histograms, faster surface renderings, common trajectory analysis tasks, …

Computing Molecular Orbitals

- Visualization of MOs aids in understanding the chemistry of molecular system
- Calculation of high resolution MO grids for display can require tens to hundreds of seconds on multi-core CPUs, even with the use of hand-coded SSE



MO 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 CPU	1*	1.0	1.0	1.0	1.0	1.0	1.0
MacMolPlt CPU	4	2.4	2.6	2.1	2.4	4.3	4.5
VMD CPU SSE	4	16.8	17.2	13.9	12.6	17.3	21.5
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}

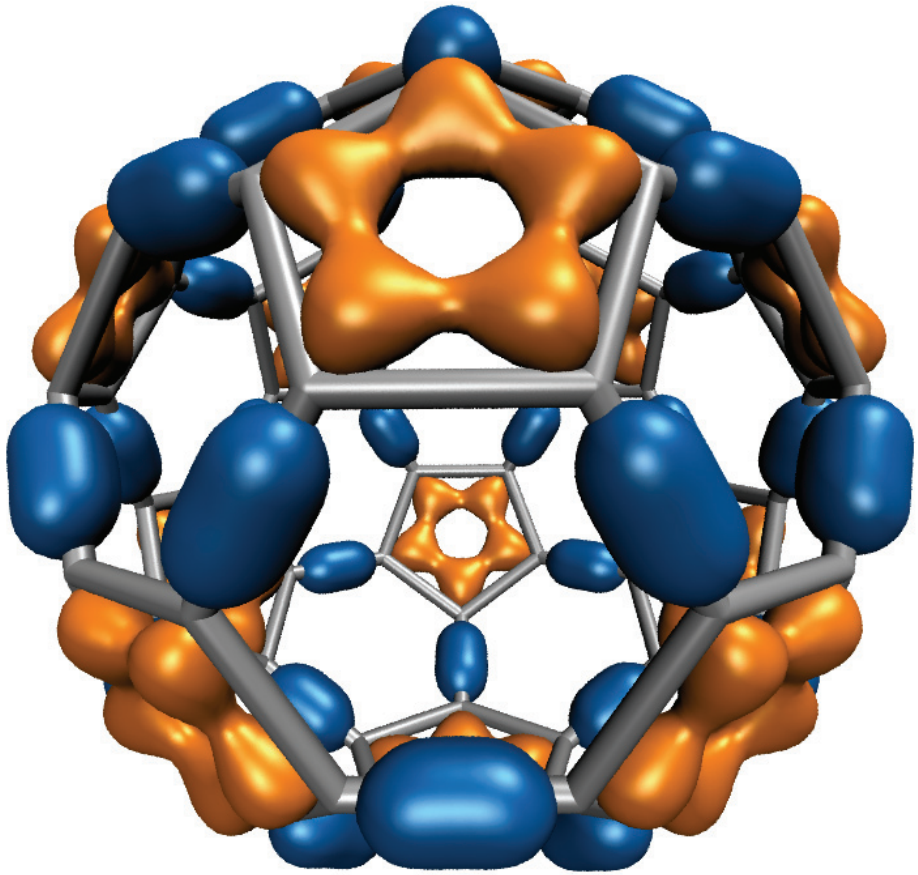
Intel X5550 CPU, 4x Tesla C1060 GPUs,

Kernel	Cores/GPUs	Runtime (s)	Speedup
CPU-ICC-SSE	1	30.64	1.0
CPU-ICC-SSE	8	4.13	7.4
CUDA-const-cache	1	0.381	80.4
CUDA-const-cache	2	0.199	154
CUDA-const-cache	3	0.143	214
CUDA-const-cache	4	0.111	276

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

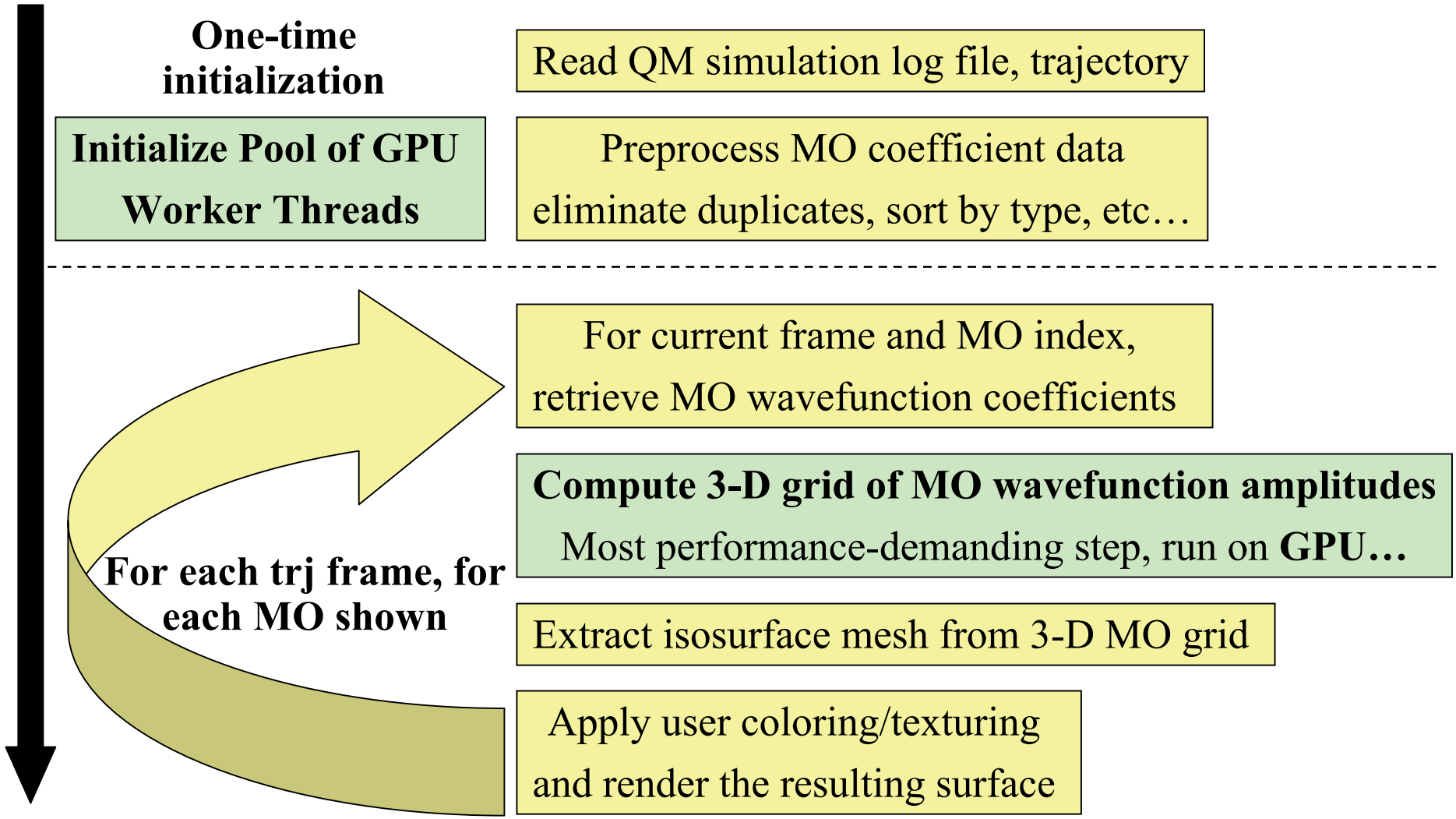
Animating Molecular Orbitals

- 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!

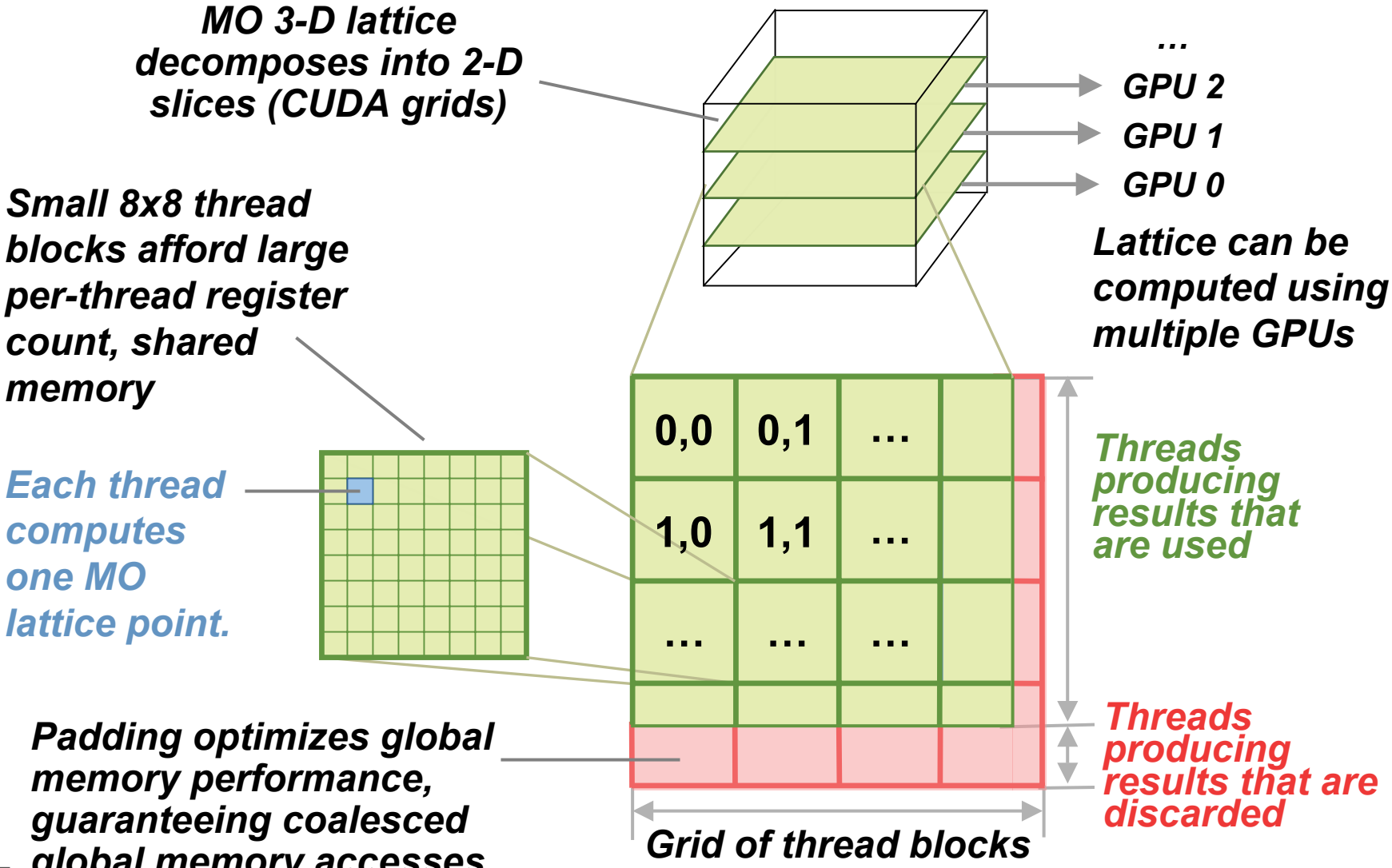


C_{60}

Molecular Orbital Computation and Display Process

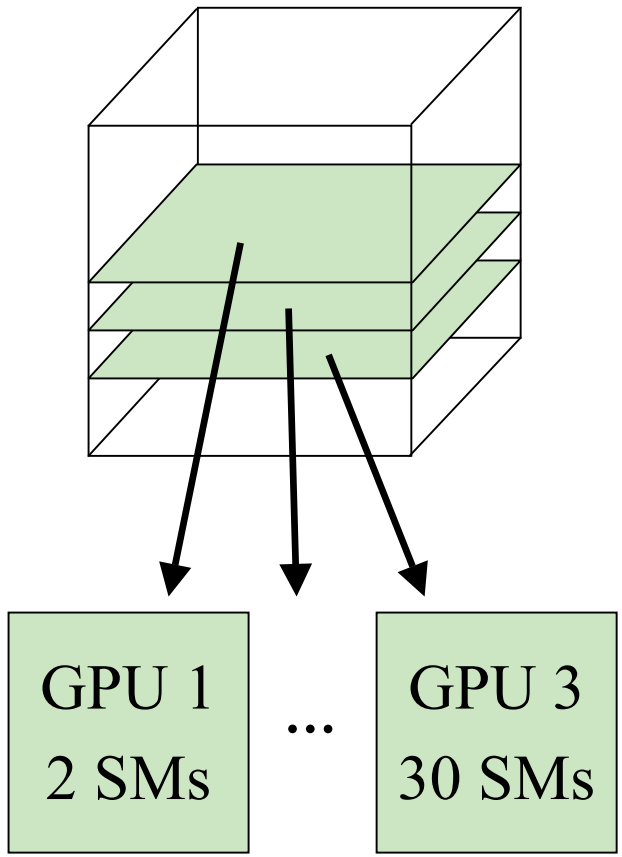


MO GPU Parallel Decomposition



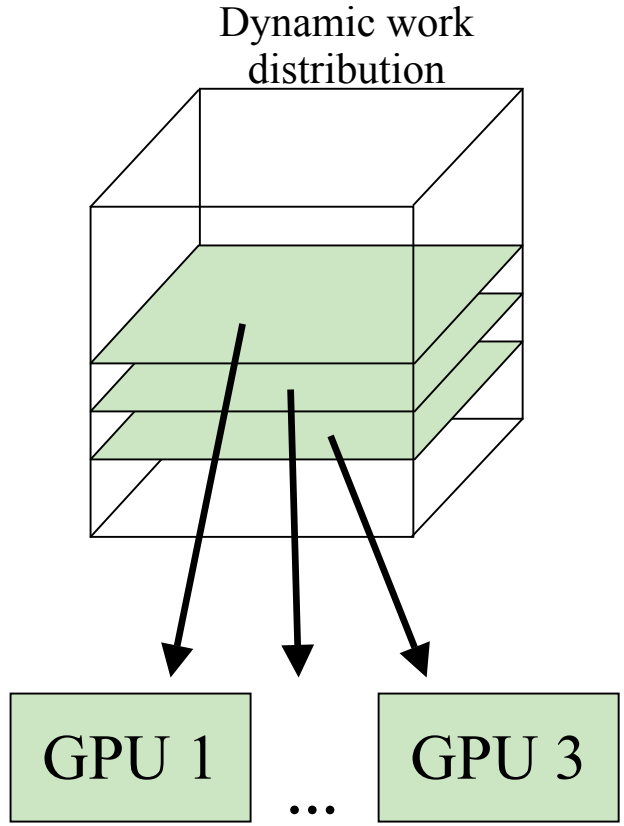
Multi-GPU Load Balance

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



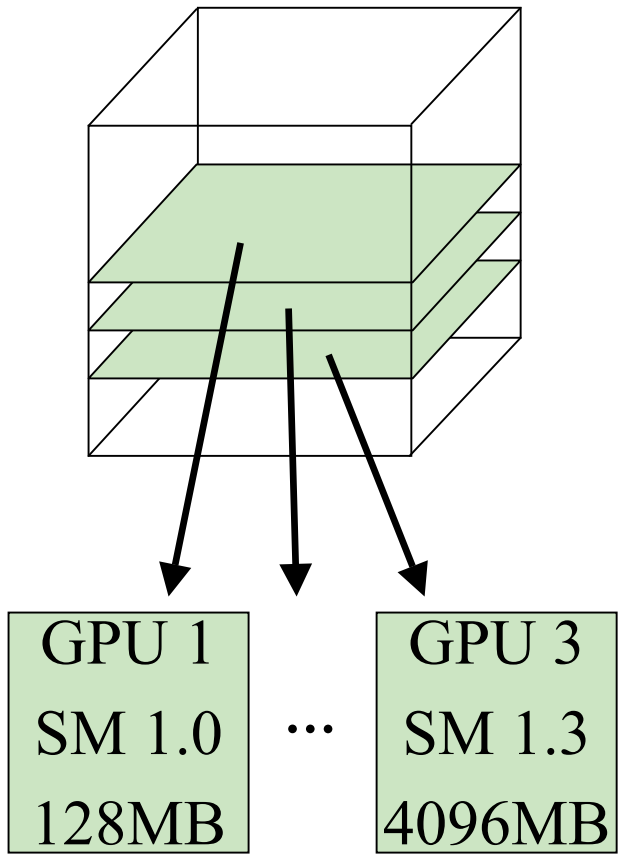
Multi-GPU Dynamic Work Distribution

```
// Each GPU worker thread loops over
// subset 2-D planes in a 3-D cube…
while (!threadpool_next_tile(&parms,
    tilesize, &tile)){
  // Process one plane of work…
  // Launch one CUDA kernel for each
  //   loop iteration taken…
  // Shared iterator automatically
  //   balances load on GPUs
}
```



Multi-GPU Runtime Error/Exception Handling

- Competition for resources from other applications or the windowing system can cause runtime failures (e.g. GPU out of memory half way through an algorithm)
- Handling of algorithm exceptions (e.g. convergence failure, NaN result, etc)
- Need to handle and/or reschedule failed tiles of work



Some Example Multi-GPU Latencies Relevant to Interactive Sci-Viz Apps

8.4us	CUDA empty kernel (immediate return)
10.0us	Sleeping barrier primitive (non-spinning barrier that uses POSIX condition variables to prevent idle CPU consumption while workers wait at the barrier)
20.3us	pool wake / exec / sleep cycle (no CUDA)
21.4us	pool wake / 1 x (tile fetch) / sleep cycle (no CUDA)
30.0us	pool wake / 1 x (tile fetch / CUDA nop kernel) / sleep cycle, test CUDA kernel computes an output address from its thread index, but does no output
1441.0us	pool wake / 100 x (tile fetch / CUDA nop kernel) / sleep cycle test CUDA kernel computes an output address from its thread index, but does no output

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)

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
```

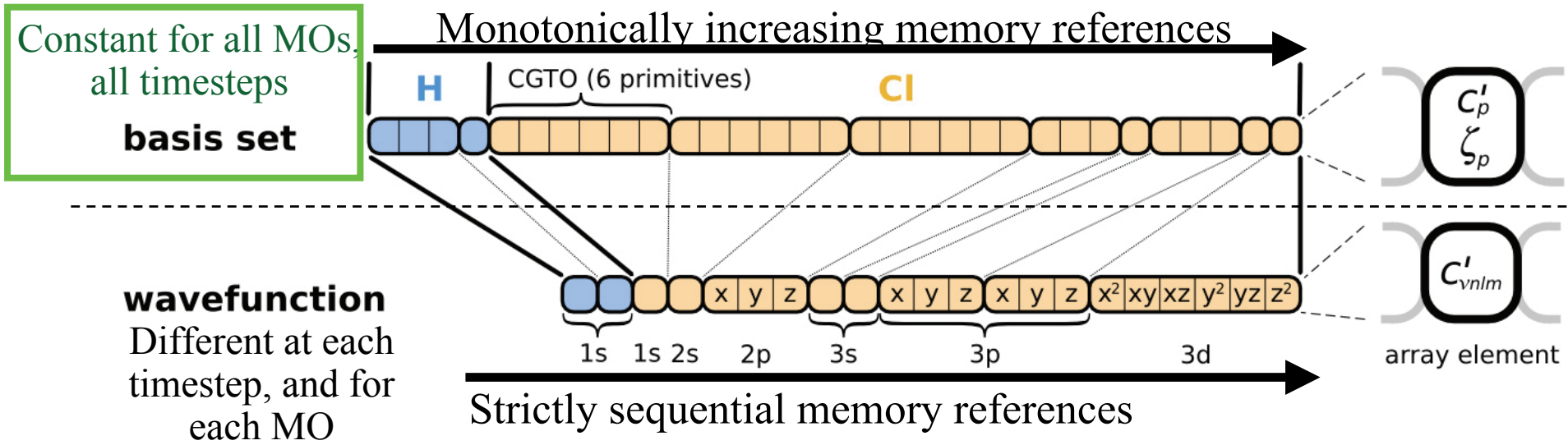
Loop unrolling:

- 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

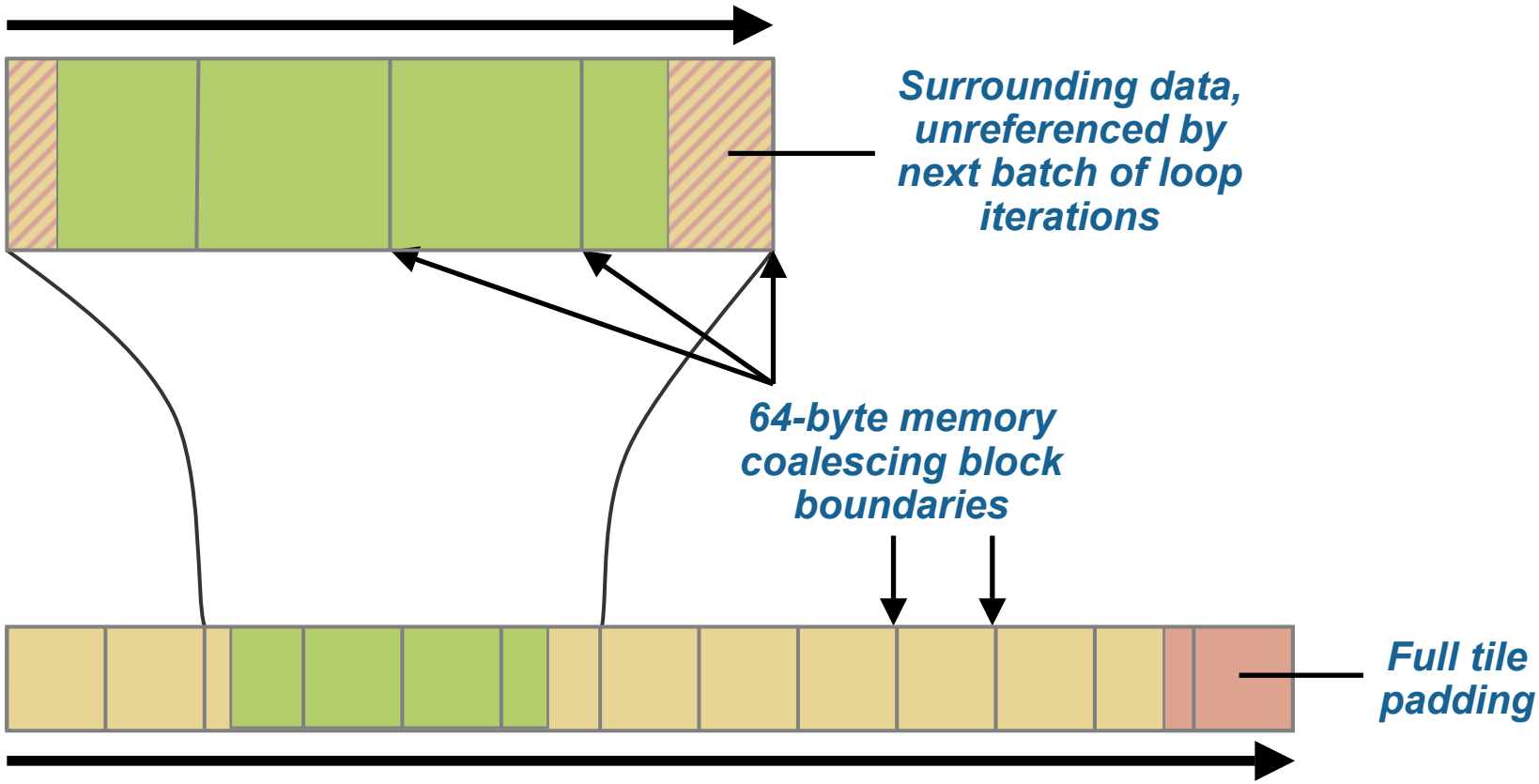


- Loop iterations always access same or consecutive array elements for all threads in a thread block:
  - Yields good constant memory and L1 cache performance
  - Increases shared memory tile reuse

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 (GT200)
- "Fermi" GPUs provide 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.



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

"Fermi" Brings Opportunities for Higher Performance and Easier Programming

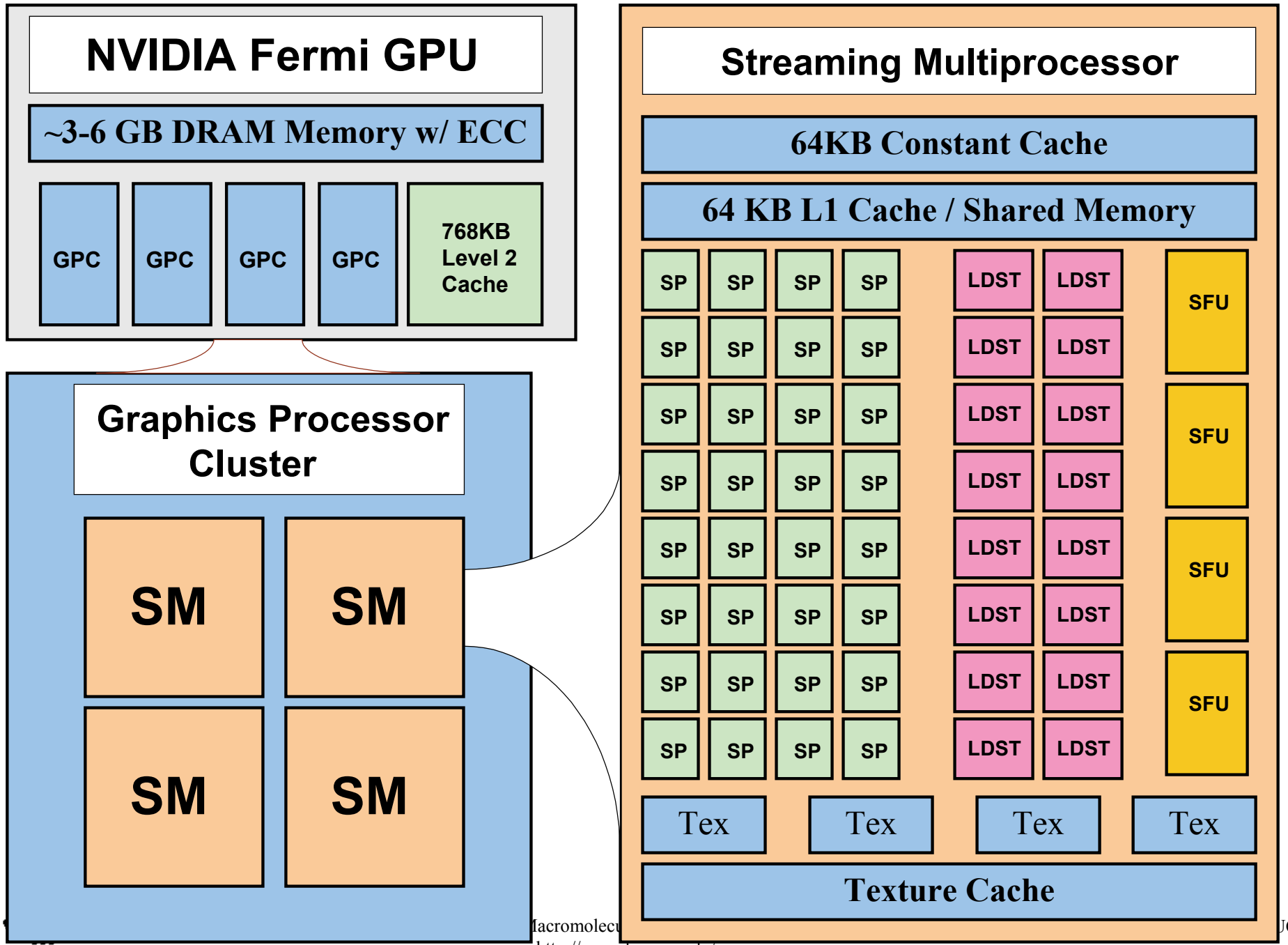
- NVIDIA's latest "Fermi" GPUs bring:
  - Greatly increased peak single- and double-precision arithmetic rates
  - Moderately increased global memory bandwidth
  - Increased capacity on-chip memory partitioned into shared memory and an L1 cache for global memory
  - Concurrent kernel execution
  - Bidirectional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

VMD MO GPU Kernel Snippet:
Fermi kernel based on L1 cache

```
[… outer loop over atoms …]
  // loop over the shells belonging to this atom (or basis function)
  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


NVIDIA Fermi GPU
~3-6 GB DRAM Memory w/ ECC
GPC
GPC
GPC
GPC
768KB Level 2 Cache
Graphics Processor Cluster
SM
SM
SM
SM
Streaming Multiprocessor
64KB Constant Cache
64 KB L1 Cache / Shared Memory
SP
LDST
SFU
Tex
Texture Cache

VMD MO Performance Results for C_{60}
2.6GHz Intel X5550 vs. NVIDIA GTX 480

Kernel	Cores/GPUs	Runtime (s)	Speedup
CPU ICC-SSE	1	30.64	1.0
CPU ICC-SSE	8	4.13	7.4
CUDA-tiled-shared	1	0.37	83
CUDA-Fermi-L1-cache (16kB)	1	0.27	113
CUDA-const-cache	1	0.26	117

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

Fermi L1 cache supports arbitrary sized problems, at near peak performance, with much simpler kernel design…

MO Kernel Structure, Opportunity for JIT…

Data-driven. 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 can entirely unroll inner loops, resulting in 40-70% speed boost**

}

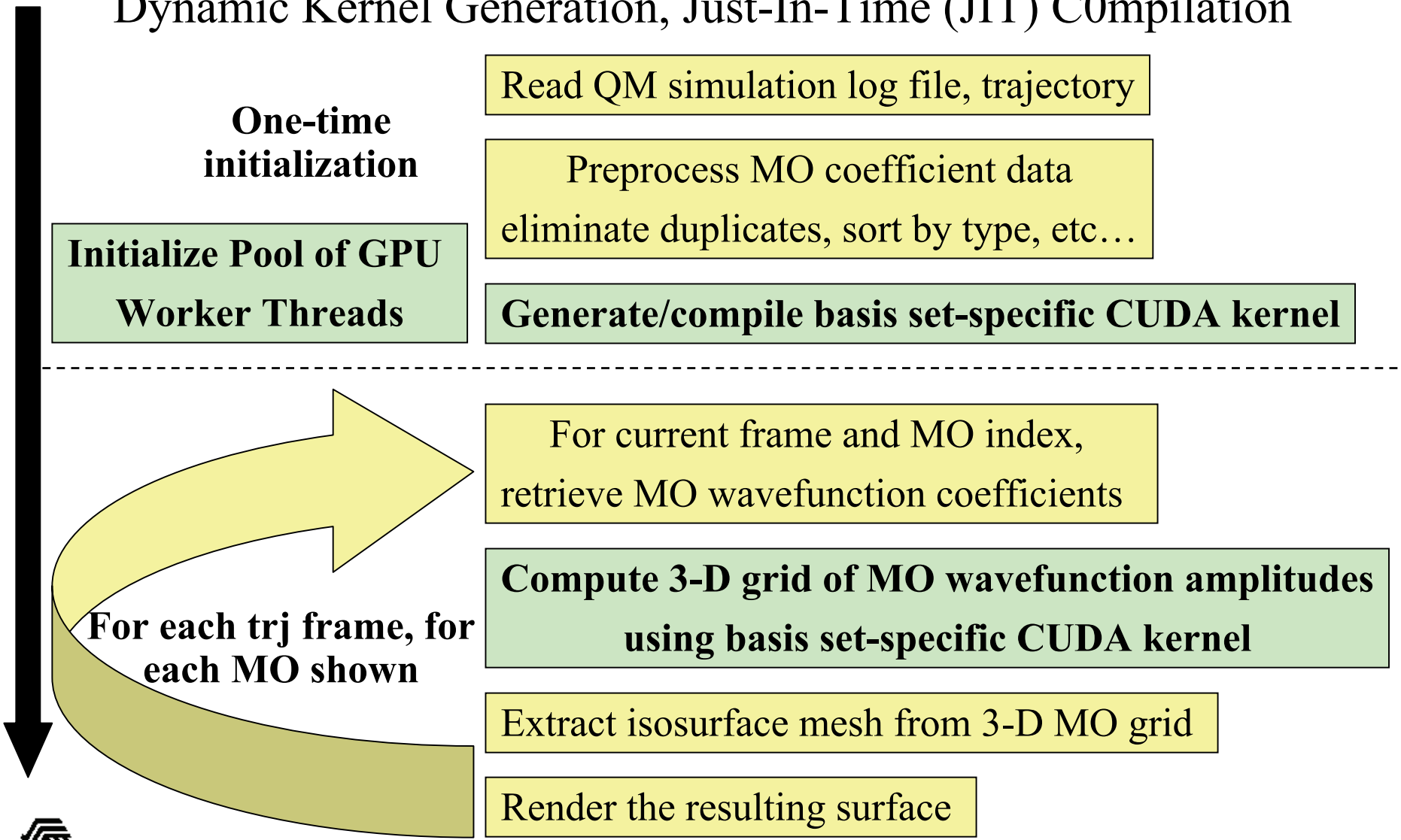
Loop over angular momenta for this shell type (1 to ~15) {}

}

}

Molecular Orbital Computation and Display Process

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



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

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

  /* multiply with the appropriate wavefunction coefficient */
  float tmpshell=0;
  switch (shell_type) {
   case S_SHELL:
    value += const_wave_f[ifunc++] * contracted_gto;
    break;
[…..]
   case D_SHELL:
    tmpshell += const_wave_f[ifunc++] * xdist2;
    tmpshell += const_wave_f[ifunc++] * ydist2;
    tmpshell += const_wave_f[ifunc++] * zdist2;
    tmpshell += const_wave_f[ifunc++] * xdist * ydist;
    tmpshell += const_wave_f[ifunc++] * xdist * zdist;
    tmpshell += const_wave_f[ifunc++] * ydist * zdist;
    value += tmpshell * contracted_gto;
    break;
```

General loop-based CUDA/OpenCL kernel

Dynamically-generated basis set-specific CUDA/OpenCL kernel (JIT)

JIT kernels 40% to 70% 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;
```

VMD MO 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 $1/8^{th}$ the grid points.

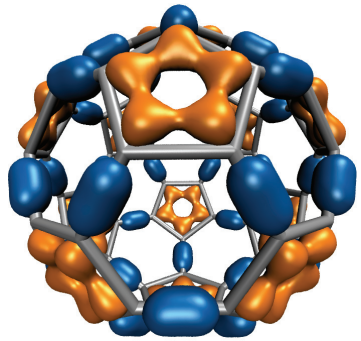
JIT kernels eliminate overhead for low trip count for loops, replace dynamic table lookups with constants, and increase floating point arithmetic intensity

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!)

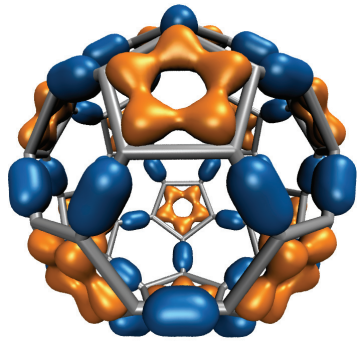
```
for (shell=0; shell < maxshell; shell++) {
  __m128 Cgto = _mm_setzero_ps();
  for (prim=0; prim<num_prim_per_shell[shell_counter]; prim++) {
    float exponent       = -basis_array[prim_counter    ];
    float contract_coeff =  basis_array[prim_counter + 1];
    __m128 expval = _mm_mul_ps(_mm_load_ps1(&exponent),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

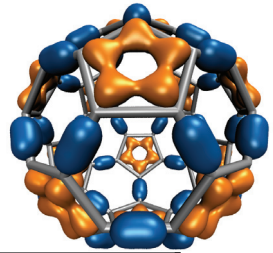
Molecular Orbital Inner Loop, OpenCL Vec4 Ahhh, much easier to read!!!

```
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

- Jan Saam, David Hardy, Kirby Vandivort
- Theoretical and Computational Biophysics Group, University of Illinois at Urbana-Champaign
- Wen-mei Hwu and the IMPACT group at University of Illinois at Urbana-Champaign
- NVIDIA CUDA Center of Excellence, University of Illinois at Urbana-Champaign
- NCSA Innovative Systems Lab
- The CUDA team at NVIDIA
- NIH support: P41-RR05969