Accelerating Molecular Modeling Applications with GPU Computing

John 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/

Second Sharcnet Symposium on GPU and Cell Computing

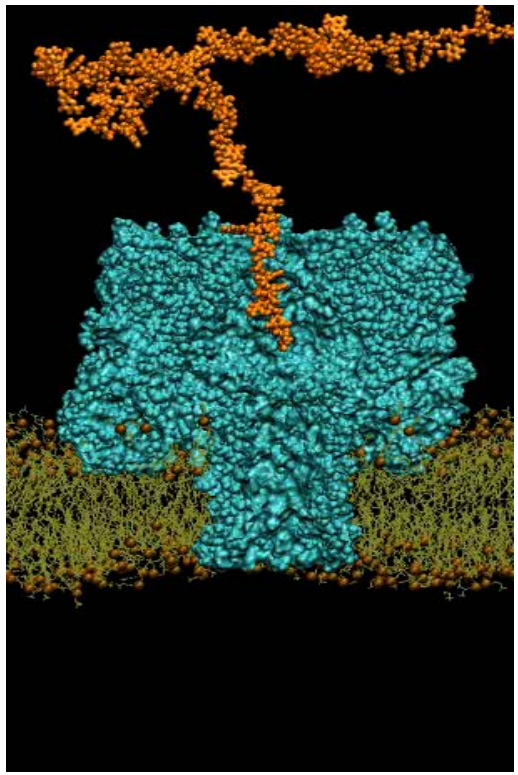
University of Waterloo, Canada, May 20, 2009

Outline

- Evolution of GPU hardware and software toward programmability and general purpose use
- Accelerating molecular modeling applications with GPUs:
  - CUDA overview (brief)
  - General GPU programming techniques
  - VMD molecular visualization and analysis

Computational Biology's Insatiable Demand for Processing Power

- Simulations still fall short of biological timescales
- Large simulations extremely difficult to prepare, analyze
- Order of magnitude increase in performance would allow use of more sophisticated models



Programmable Graphics Hardware

Groundbreaking research systems:

AT&T Pixel Machine (1989):

82 x DSP32 processors

UNC PixelFlow (1992-98):

64 x (PA-8000 +

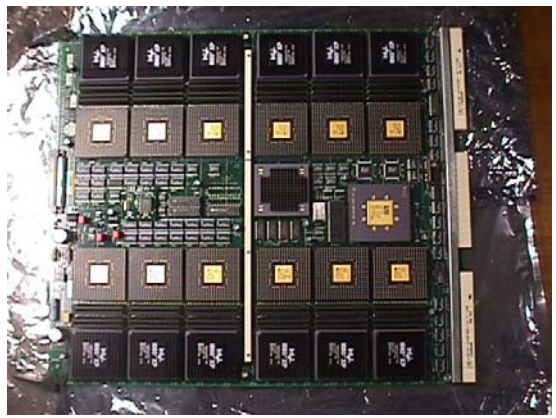
8,192 bit-serial SIMD)

SGI RealityEngine (1990s):

Up to 12 i860-XP processors perform vertex operations (*u*code), fixed-func. fragment hardware

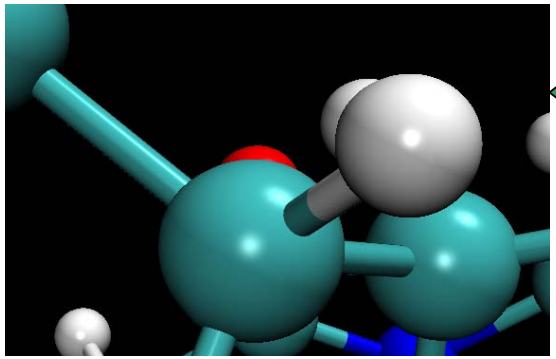
All mainstream GPUs now incorporate fully programmable processors

UNC PixelFlow Rack



SGI Reality Engine i860 Vertex Processors

GLSL Sphere Fragment Shader



- Written in OpenGL Shading Language
- High-level C-like language with vector types and operations
- Compiled dynamically by the graphics driver at *runtime*
- Compiled machine code executes on GPU

```
//
// VMD Sphere Fragment Shader (not for normal geometry)
//
void main(void) {
  vec3 raydir = normalize(V);
  vec3 spheredir = spherepos - rayorigin;

  // Perform ray-sphere intersection tests based on the code in Tachyon
  float b = dot(raydir, spheredir);
  float temp = dot(spheredir, spheredir);
  float disc = b*b + sphereradsq - temp;

  // only calculate the nearest intersection, for speed
  if (disc <= 0.0)
    discard; // ray missed sphere entirely, discard fragment

  // calculate closest intersection
  float tnear = b - sqrt(disc);

  if (tnear < 0.0)
    discard;

  // calculate hit point and resulting surface normal
  vec3 pnt = rayorigin + tnear * raydir;
  vec3 N = normalize(pnt - spherepos);

  // Output the ray-sphere intersection point as the fragment depth
  // rather than the depth of the bounding box polygons.
  // The eye coordinate Z value must be transformed to normalized device
  // coordinates before being assigned as the final fragment depth.
  if (vmdprojectionmode == 1) {
    // perspective projection = 0.5 + (hfpn + (f * n / pnt.z)) / diff
    gl_FragDepth = 0.5 + (vmdprojparms[2] + (vmdprojparms[1] * vmdprojparms[
3];
  } else {
    // orthographic projection = 0.5 + (-hfpn - pnt.z) / diff
    gl_FragDepth = 0.5 + (-vmdprojparms[2] - pnt.z) / vmdprojparms[3];
  }

#ifdef TEXTURE
  // perform texturing operations for volumetric data
  // The only texturing mode that applies to the sphere shader
```

Origins of Computing on GPUs

- Widespread support for programmable shading led researchers to begin experimenting with the use of GPUs for general purpose computation, “GPGPU”
- Early GPGPU efforts used existing graphics APIs to express computation in terms of drawing
- As expected, expressing general computation problems in terms of triangles and pixels and “drawing the answer” is obfuscating and painful to debug…
- Soon researchers began creating dedicated GPU programming tools, starting with Brook and Sh, and ultimately leading to a variety of commercial tools such as RapidMind, CUDA, OpenCL, and others...

GPU Computing

- Commodity devices, omnipresent in modern computers (over a **million** sold per **week**)
- Massively parallel hardware, hundreds of processing units, throughput oriented architecture
- Standard integer and floating point types supported
- Programming tools allow software to be written in dialects of familiar C/C++ and integrated into legacy software
- GPU algorithms are often multicore friendly due to attention paid to data locality and data-parallel work decomposition

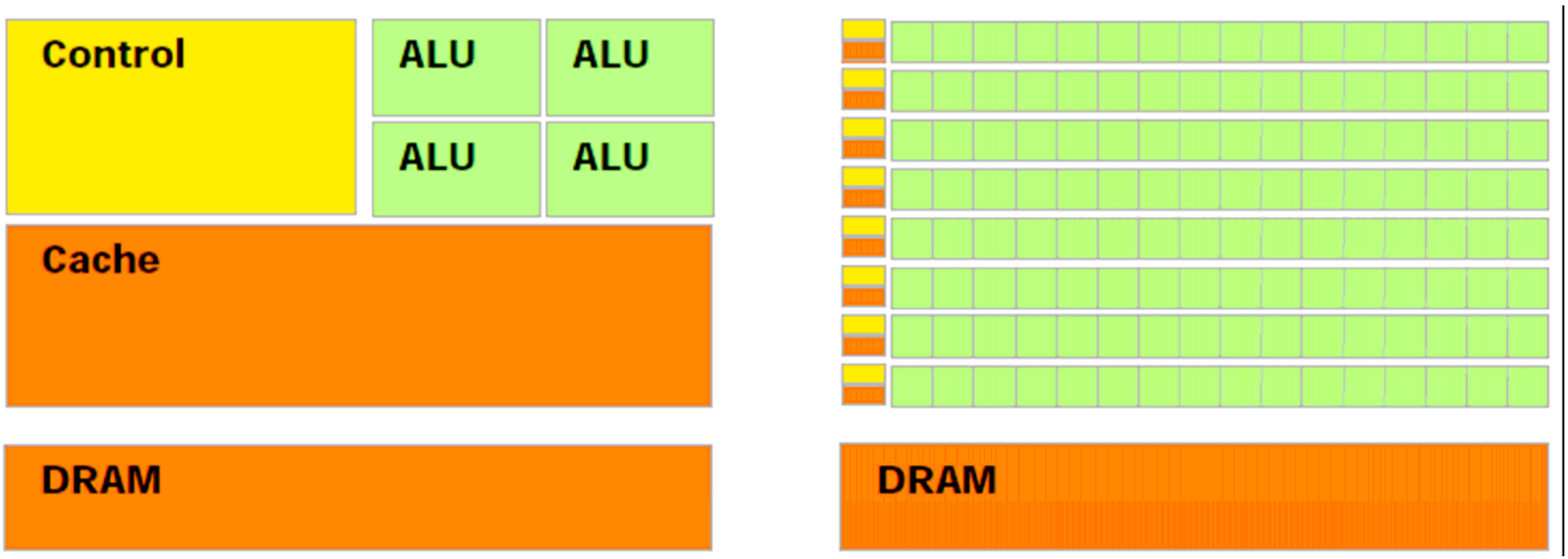
What Speedups Can GPUs Achieve?

- Single-GPU speedups of **10x** to **30x** vs. one CPU core are common
- Best speedups can reach **100x** or more, attained on codes dominated by floating point arithmetic, especially native GPU machine instructions, e.g. expf(), rsqrtf(), …
- Amdahl's Law can prevent legacy codes from achieving peak speedups with shallow GPU acceleration efforts

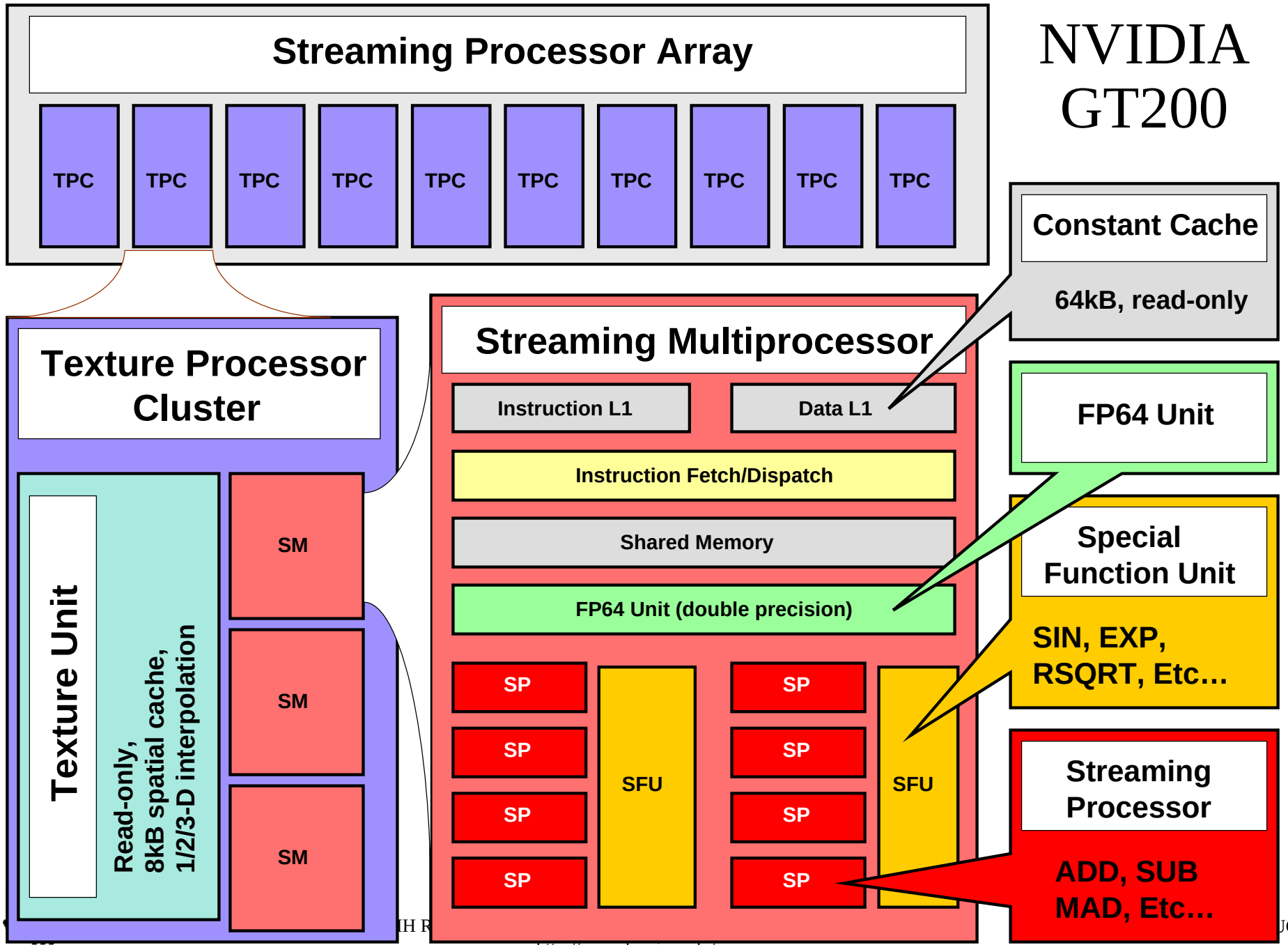
Comparison of CPU and GPU Hardware Architecture

CPU: Cache heavy, focused on individual thread performance

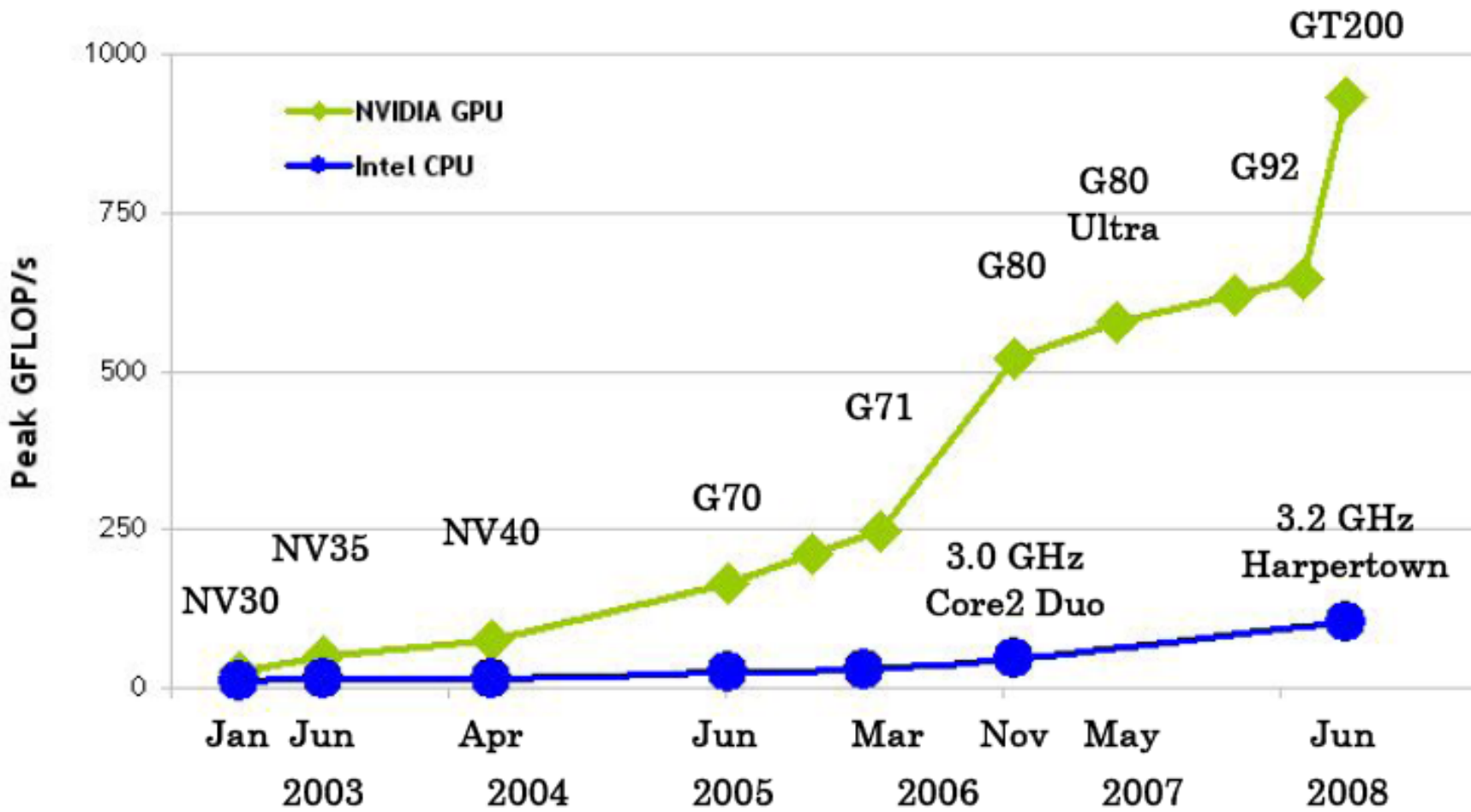
GPU: ALU heavy, massively parallel, throughput oriented



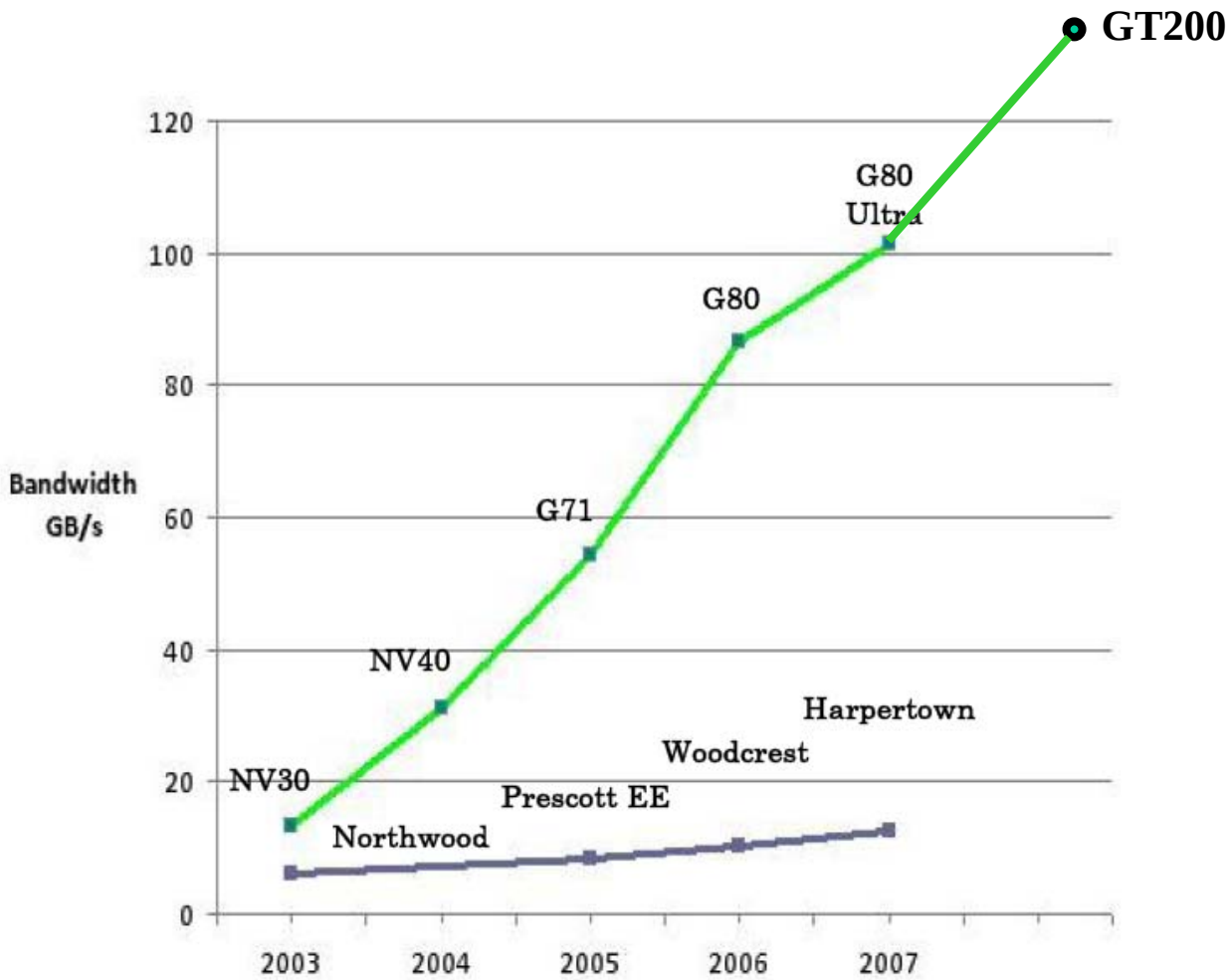
NVIDIA GT200



GPU Peak Single-Precision Performance: Exponential Trend



GPU Peak Memory Bandwidth: Linear Trend

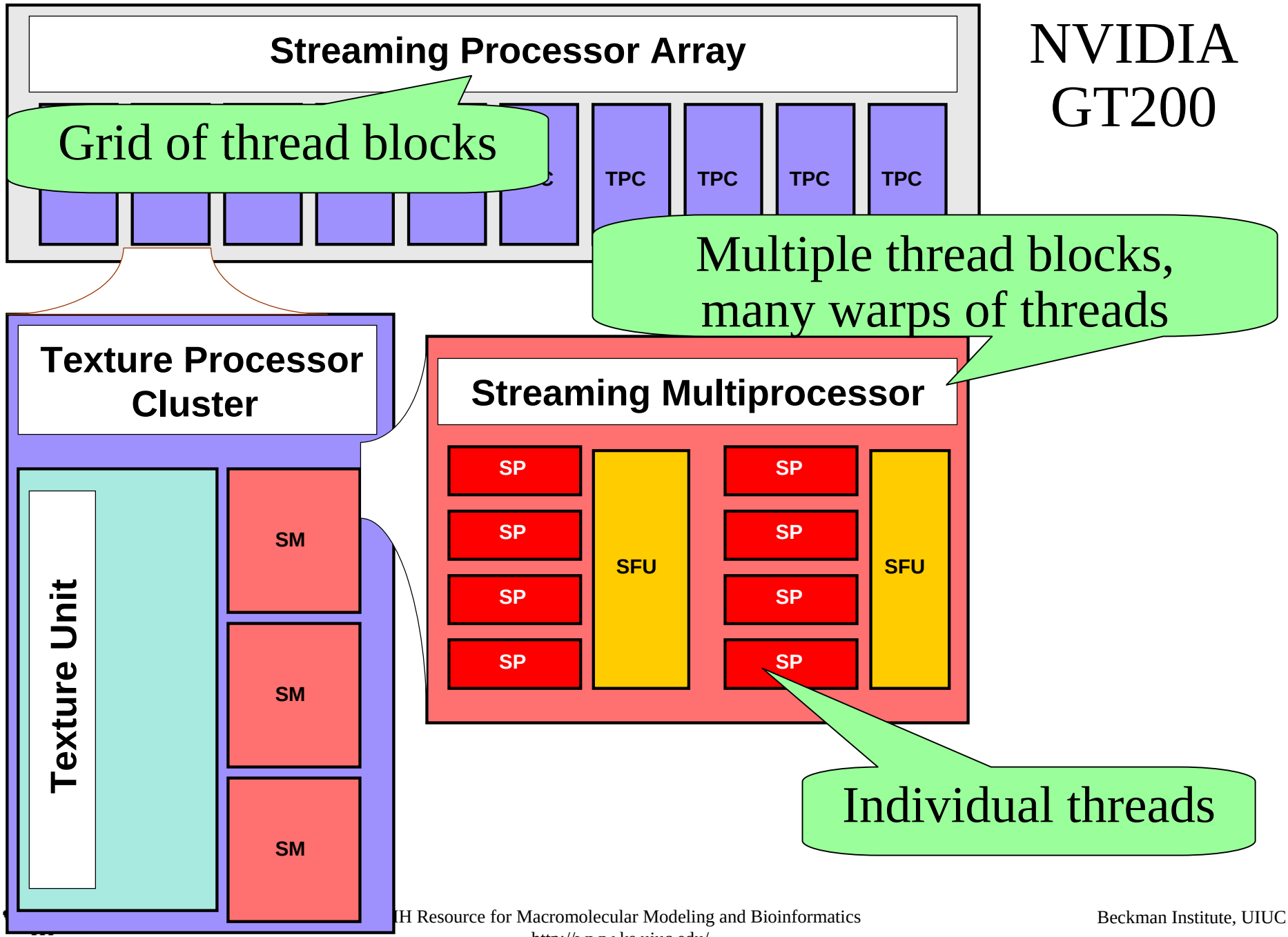


NVIDIA CUDA Overview

- Hardware and software architecture for GPU computing, foundation for building higher level programming libraries, toolkits
- C for CUDA, released in 2007:
  - Data-parallel programming model
  - Work is decomposed into “grids” of “blocks” containing “warps” of “threads”, multiplexed onto massively parallel GPU hardware
  - Light-weight, low level of abstraction, exposes many GPU architecture details/features enabling development of high performance GPU kernels

CUDA Threads, Blocks, Grids

- GPUs use hardware multithreading to hide latency and achieve high ALU utilization
- For high performance, a GPU must be **saturated** with concurrent work: **>10,000 threads**
- “Grids” of hundreds of “thread blocks” are scheduled onto a large array of SIMT cores
- Each core executes several thread blocks of 64-512 threads each, switching among them to hide latencies for slow memory accesses, etc…
- 32 thread “warps” execute in lock-step (e.g. in SIMD-like fashion)


Streaming Processor Array
TPC
TPC
TPC
TPC
Grid of thread blocks
NVIDIA
GT200
Multiple thread blocks,
many warps of threads
Texture Processor
Cluster
Texture Unit
SM
SM
SM
Streaming Multiprocessor
SP
SP
SP
SP
SFU
SP
SP
SP
SP
SFU
Individual threads

GPU Memory Accessible in CUDA

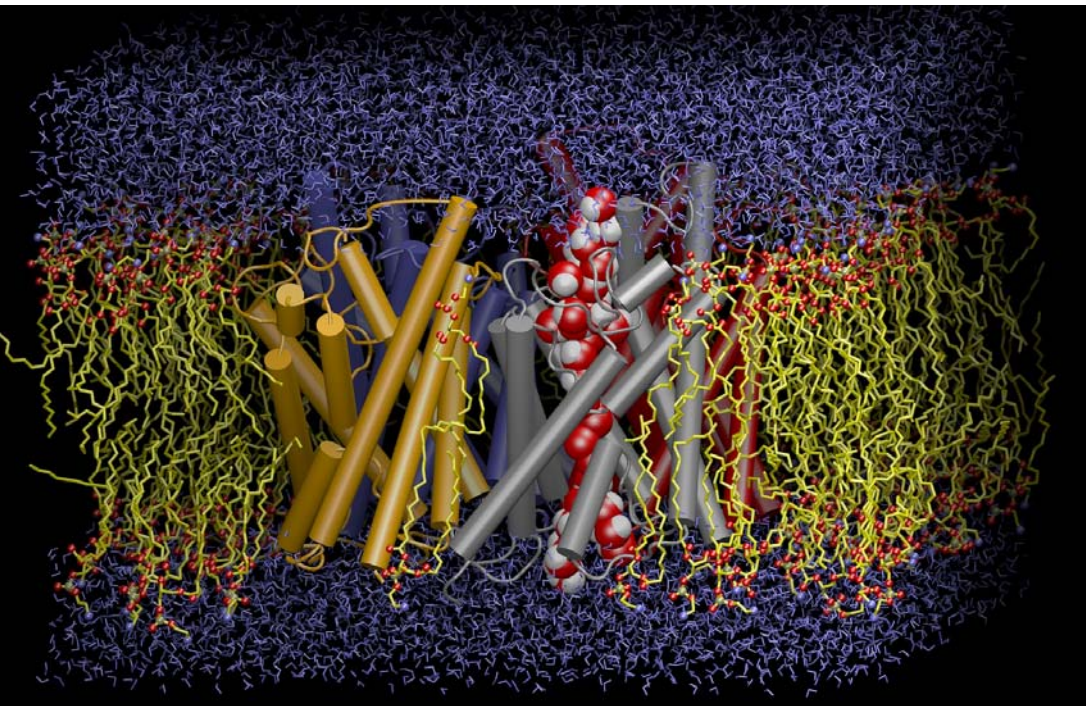
- Mapped host memory: up to 4GB, ~5.7GB/sec bandwidth (PCIe), accessible by multiple GPUs
- Global memory: up to 4GB, high latency (~600 clock cycles), 140GB/sec bandwidth, accessible by all threads, atomic operations (slow)
- Texture memory: read-only, cached, and interpolated/filtered access to global memory
- Constant memory: 64KB, read-only, cached, fast/low-latency if data elements are accessed in unison by peer threads
- Shared memory:16KB, low-latency, accessible among threads in the same block, fast if accessed without bank conflicts

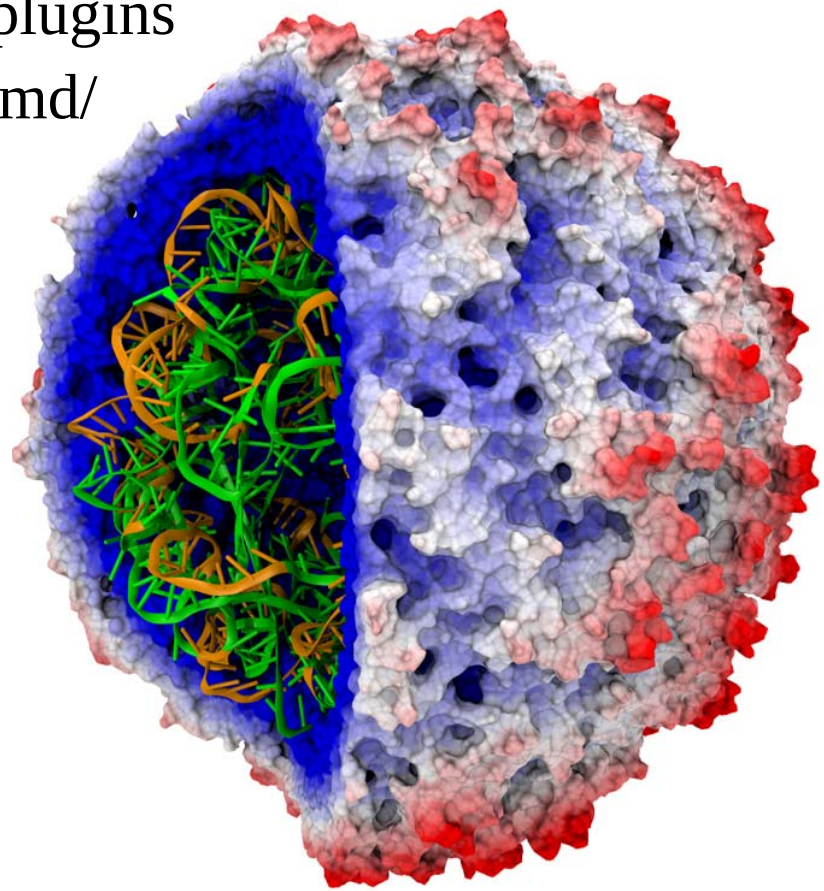
An Approach to Writing CUDA Kernels

- Find an algorithm that exposes substantial parallelism, thousands of independent threads…
- Identify appropriate GPU memory subsystems for storage of data used by kernel
- Are there trade-offs that can be made to exchange computation for more parallelism?
  - Though counterintuitive, past successes resulted from this strategy
  - “Brute force” methods that expose significant parallelism do surprisingly well on current GPUs
- Analyze the real-world use case for the problem and optimize the kernel for the problem size/characteristics that will be heavily used

VMD – “Visual Molecular Dynamics”

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





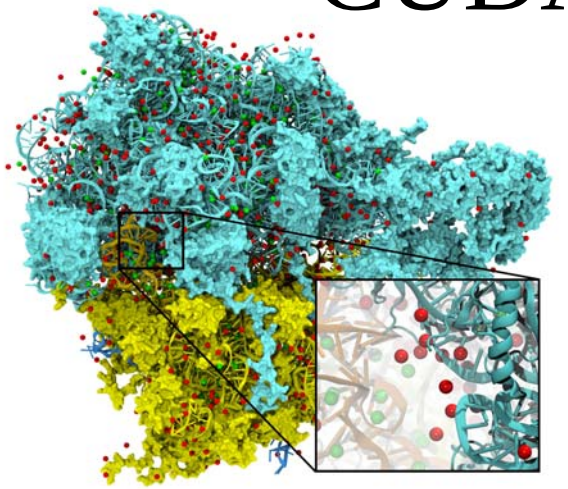
Range of VMD Usage Scenarios

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

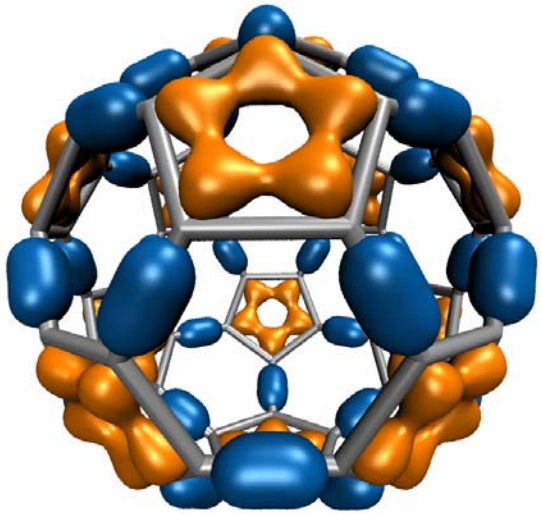
Need for Multi-GPU Acceleration in VMD

- Ongoing 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 running batch analysis jobs
- Desktop use is the most compute-resource-limited scenario, where **GPUs can make a big impact**…

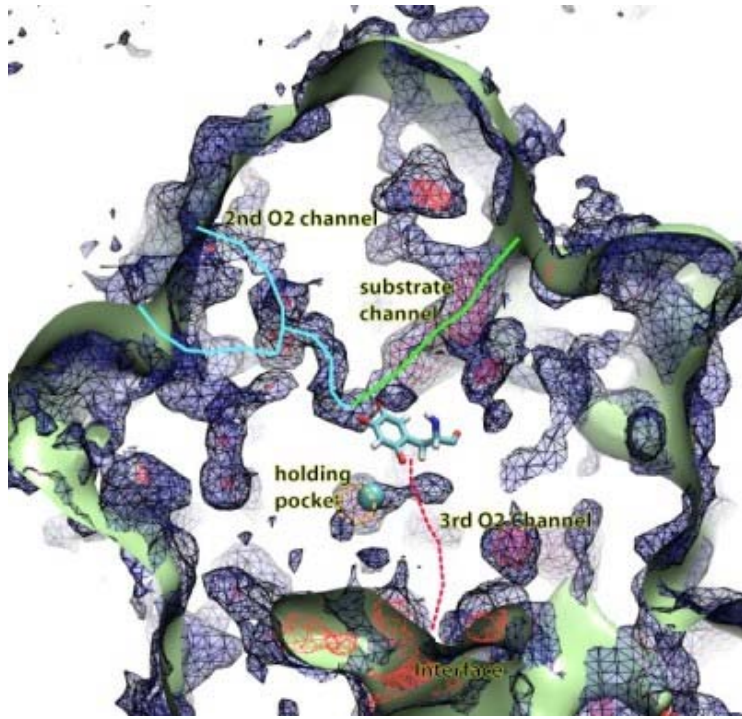
CUDA Acceleration in VMD



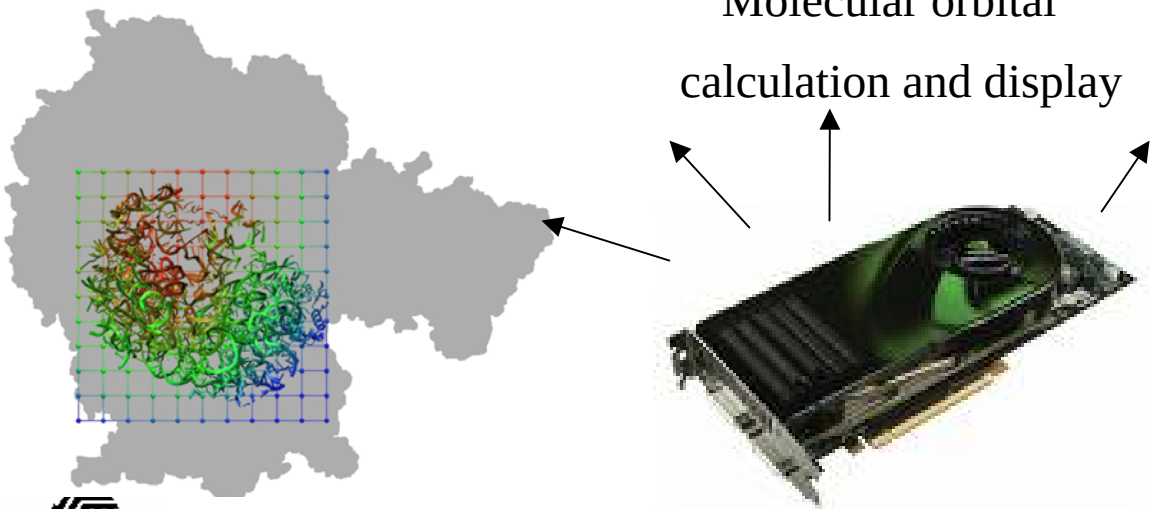
Electrostatic field calculation, ion placement



Molecular orbital calculation and display



Imaging of gas migration pathways in proteins with implicit ligand sampling

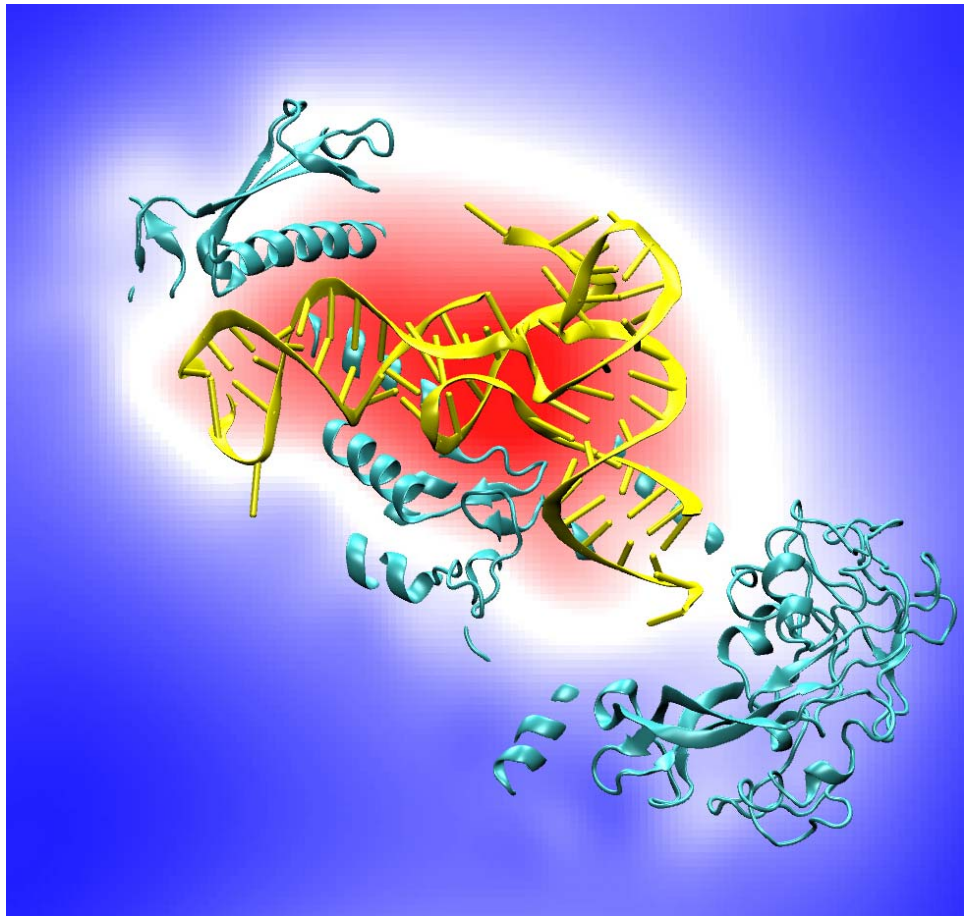


Electrostatic Potential Maps

- Electrostatic potentials evaluated on 3-D lattice:

$$V_i = \sum_j \frac{q_j}{4\pi\epsilon_0 |\mathbf{r}_j - \mathbf{r}_i|}$$

- Applications include:
  - Ion placement for structure building
  - Time-averaged potentials for simulation
  - Visualization and analysis

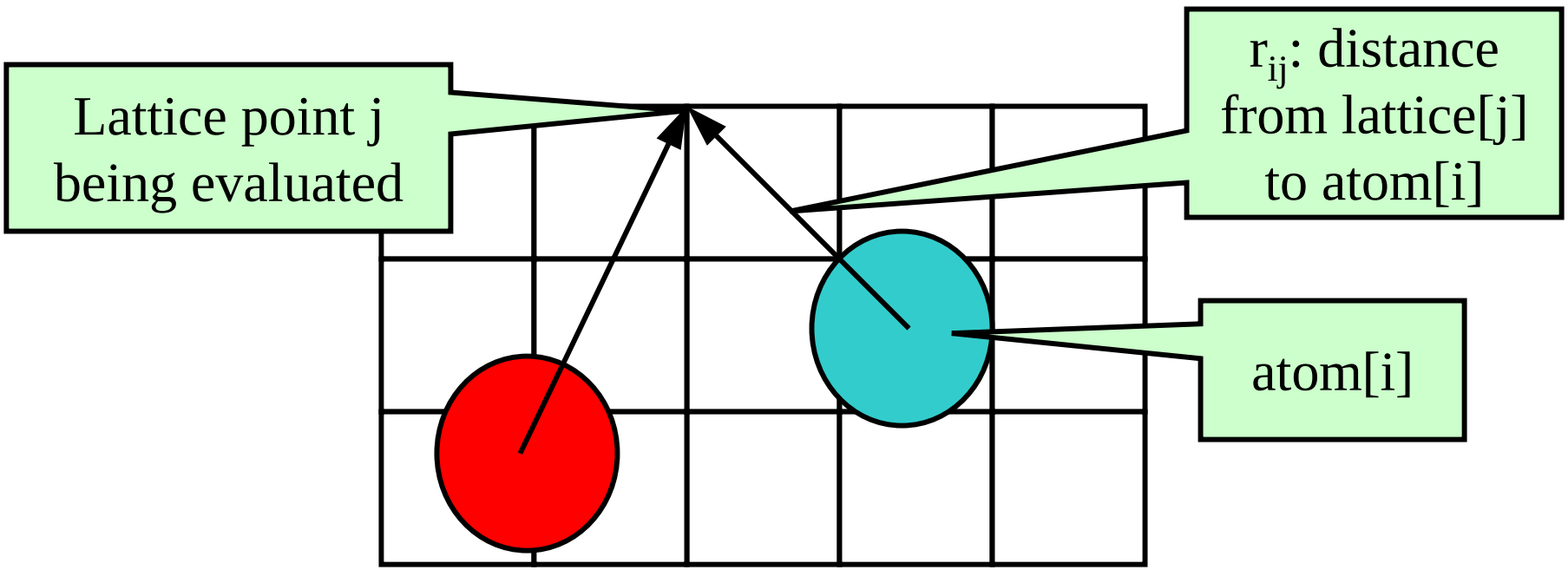


Isoleucine tRNA synthetase

Direct Coulomb Summation

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

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

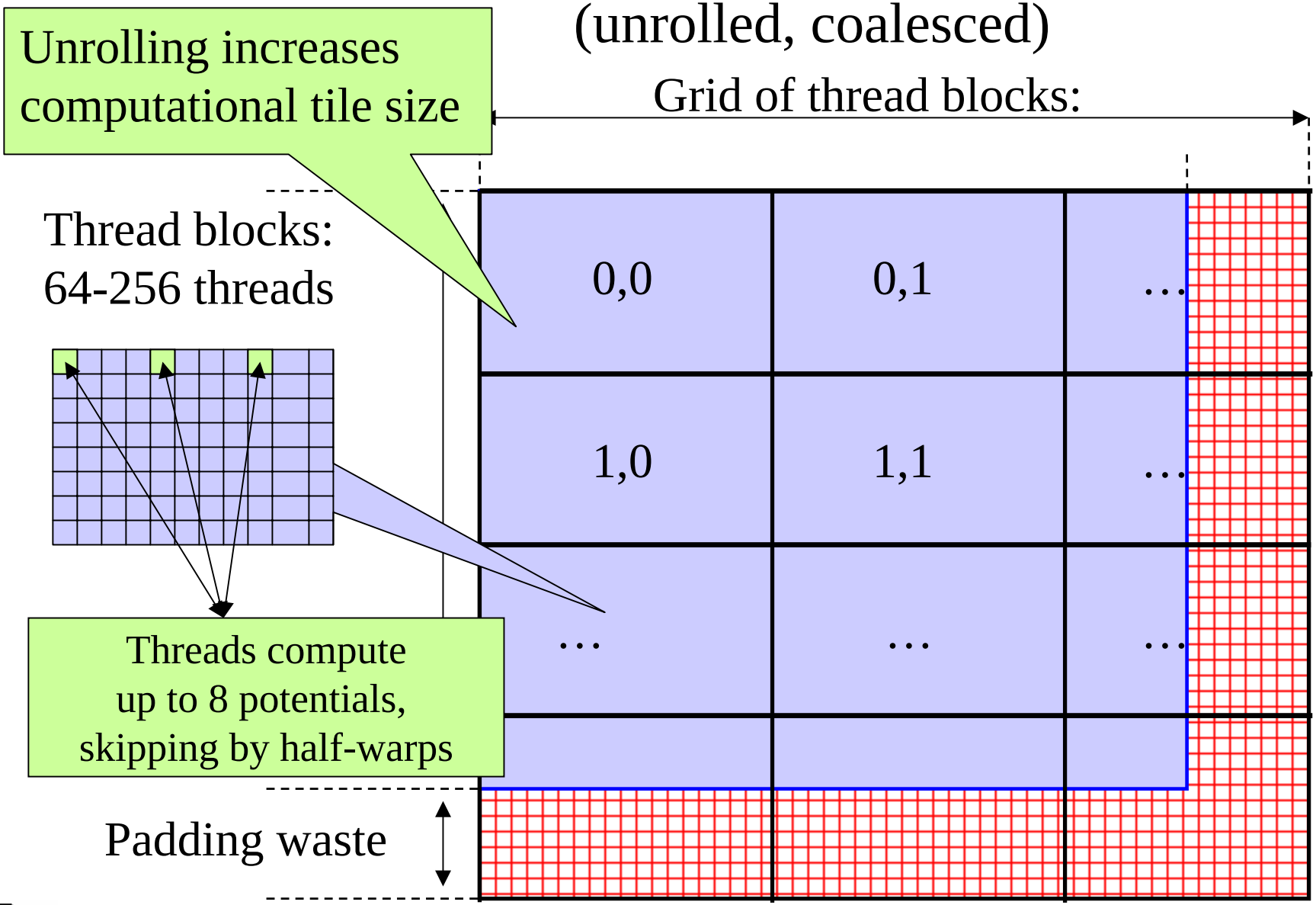


Direct Coulomb Summation on the GPU

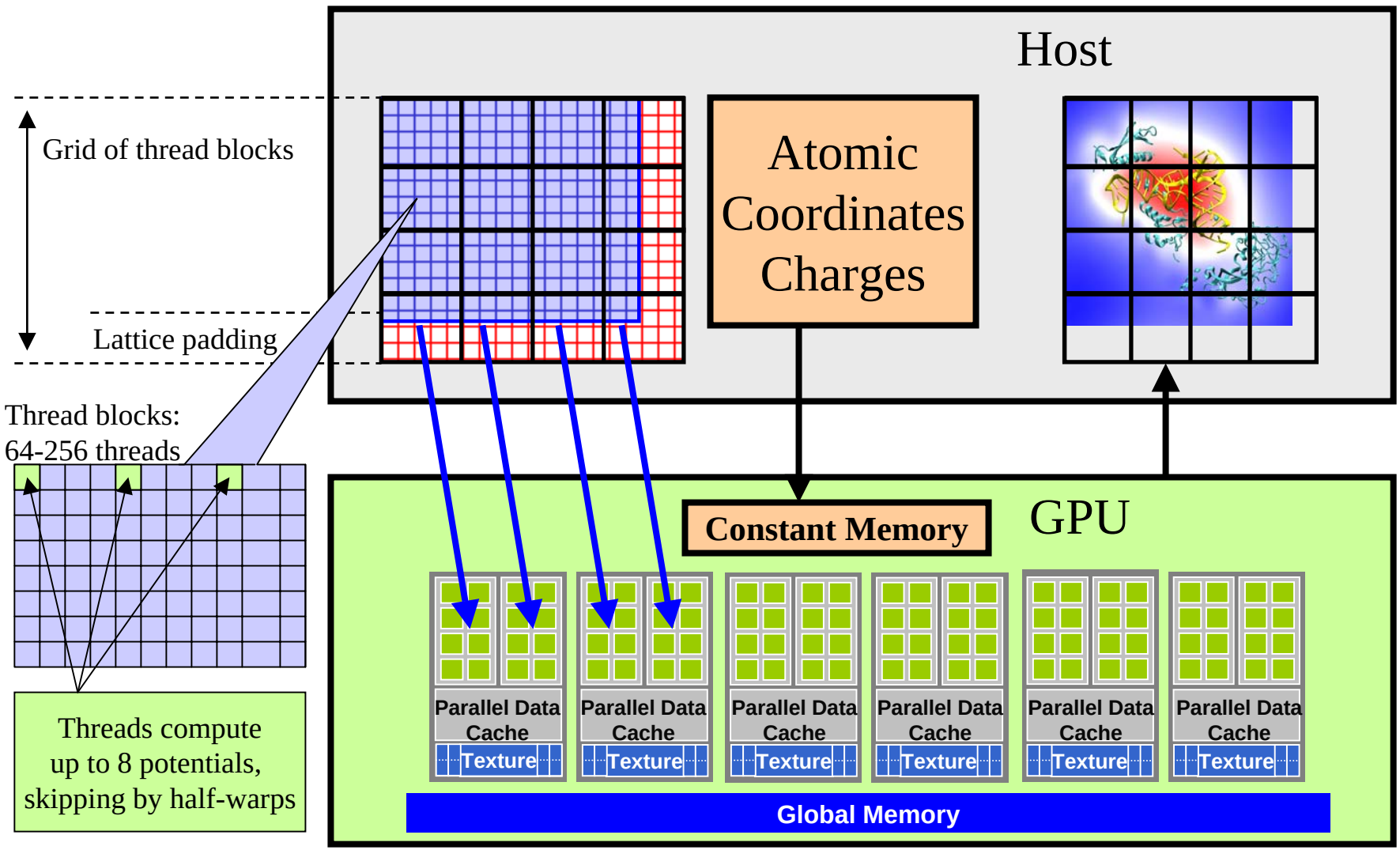
- GPU outruns a CPU core by 44x
- Work is decomposed into tens of thousands of independent threads, multiplexed onto hundreds of GPU processing units
- Single-precision FP arithmetic is adequate for intended application
- Numerical accuracy can be improved by compensated summation, spatially ordered summation groupings, or accumulation of potential in double-precision
- Starting point for more sophisticated linear-time algorithms like multilevel summation

DCS CUDA Block/Grid Decomposition

(unrolled, coalesced)

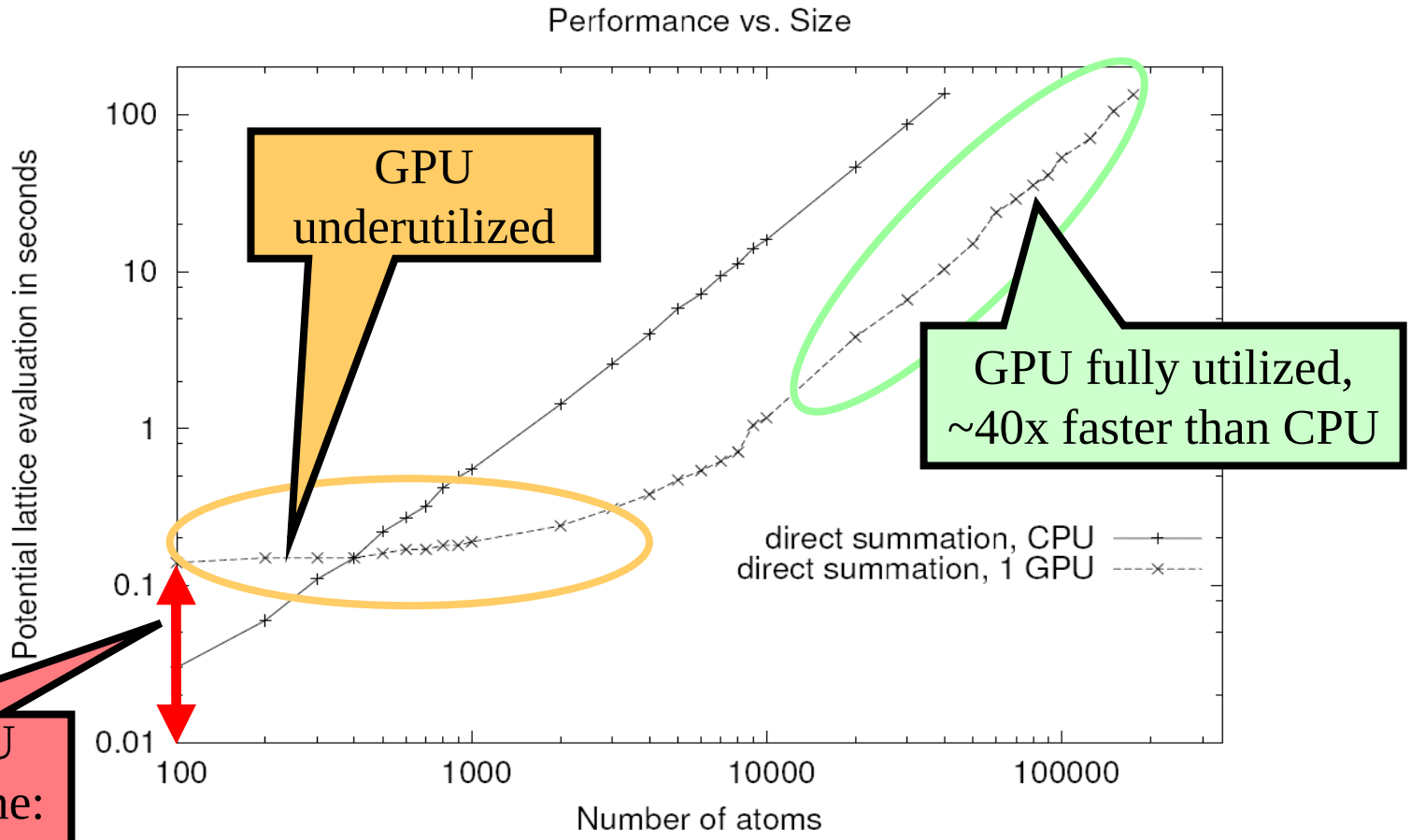


Direct Coulomb Summation on the GPU



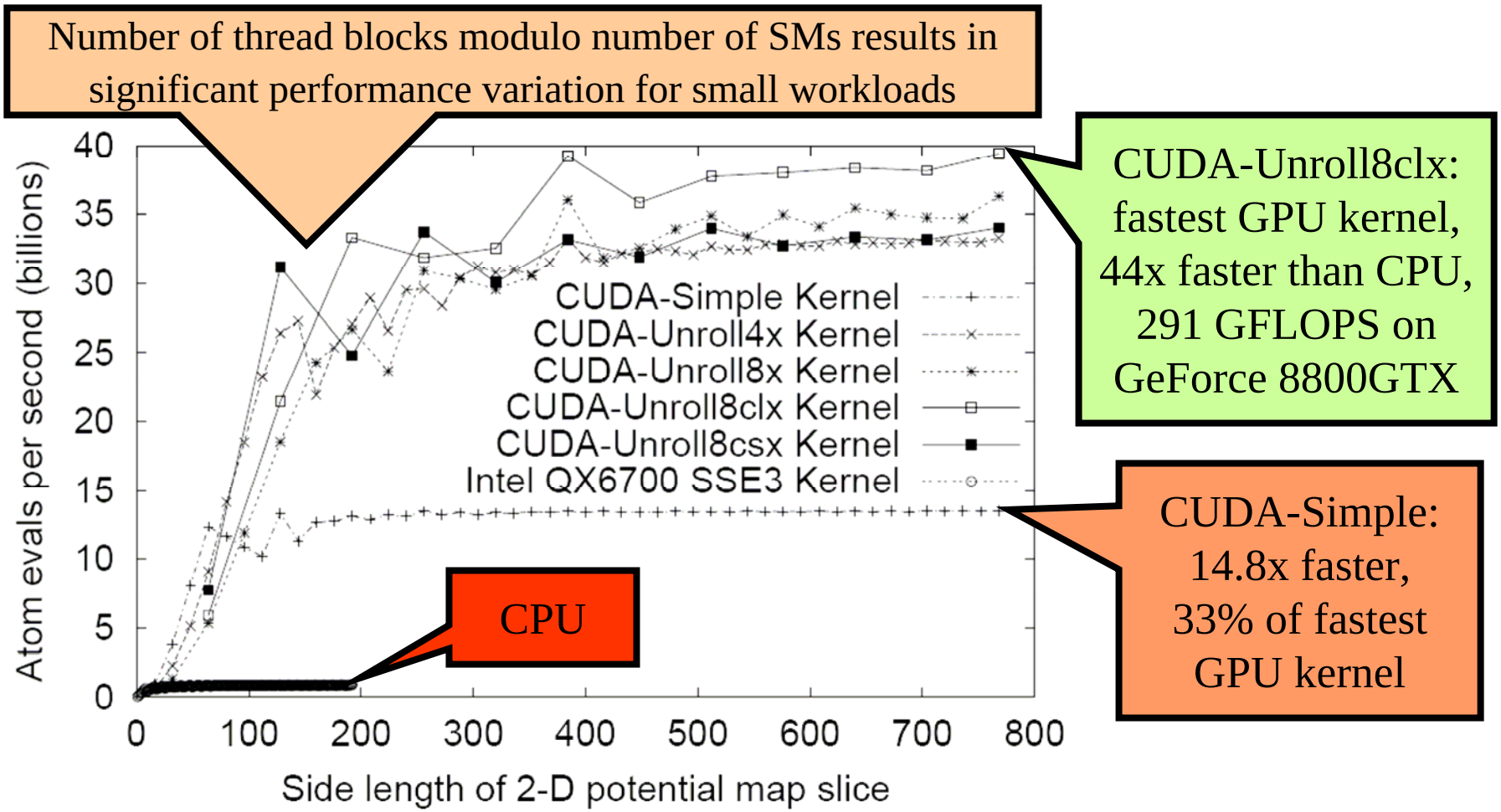
Direct Coulomb Summation Runtime

Lower is better



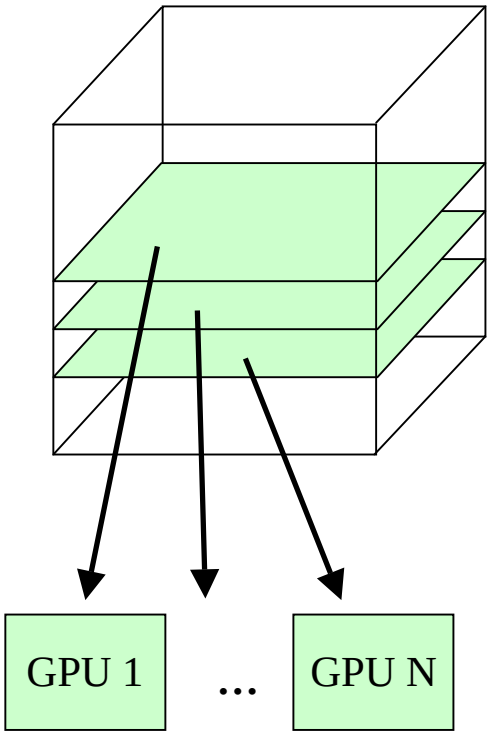
Accelerating molecular modeling applications with graphics processors. J. Stone, J. Phillips, P. Freddolino, D. Hardy, L. Trabuco, K. Schulten. *J. Comp. Chem.*, 28:2618-2640, 2007.

Direct Coulomb Summation Performance



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

Multi-GPU Direct Coulomb Summation



NCSA GPU Cluster
http://www.ncsa.uiuc.edu/Projects/GPUcluster/

	Evals/sec	TFLOPS	Speedup*
4-GPU (2 Quadroplex) Opteron node at NCSA	157 billion	1.16	176
4-GPU GTX 280 (GT200)	241 billion	1.78	271

*Speedups relative to Intel QX6700 CPU core w/ SSE

Infinite vs. Cutoff Potentials

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

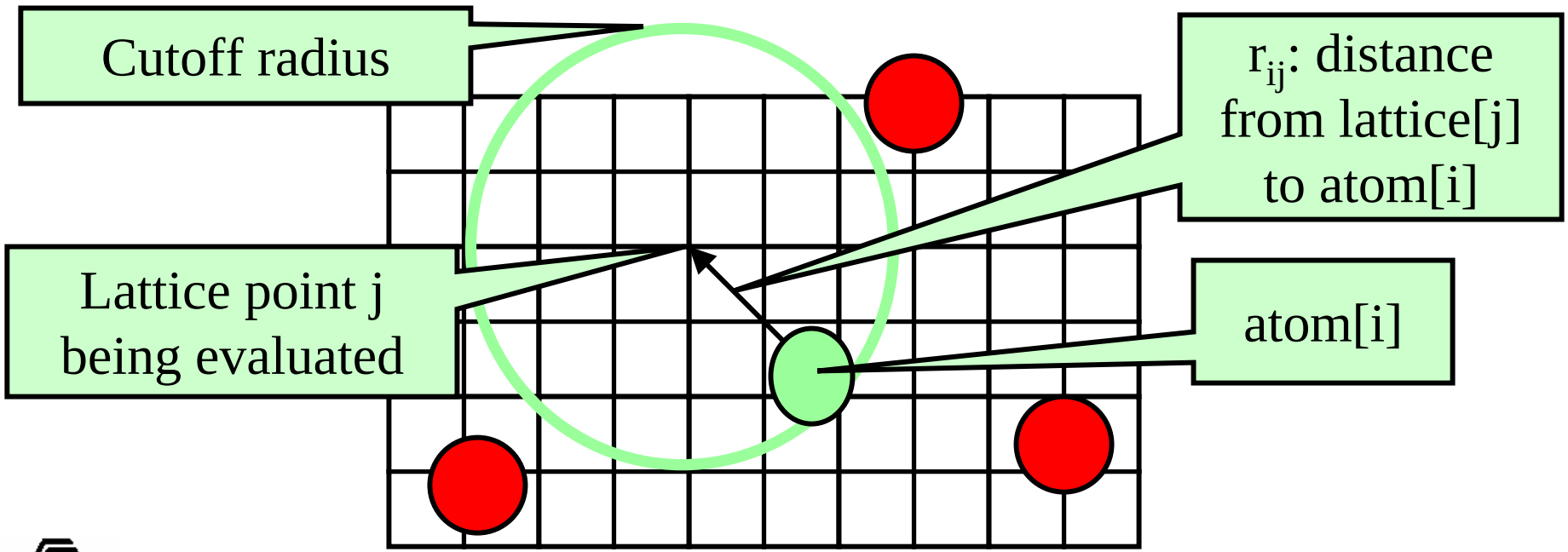
Cutoff Summation

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

  if (r_{ij} < cutoff)

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

- Smoothing function s(r) is algorithm dependent



Cutoff Summation on the GPU

Atoms are spatially hashed into fixed-size bins

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

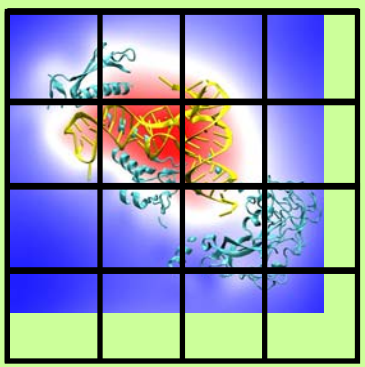
GPU thread block calculates corresponding region of potential map,

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

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

Shared memory

atom bin



Global memory

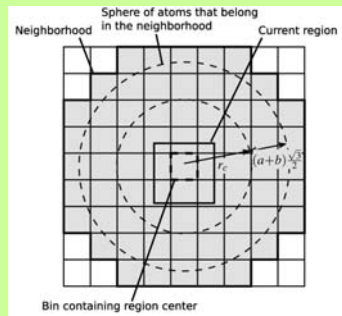
Potential map regions

Bins of atoms

Constant memory

Offsets for bin neighborhood

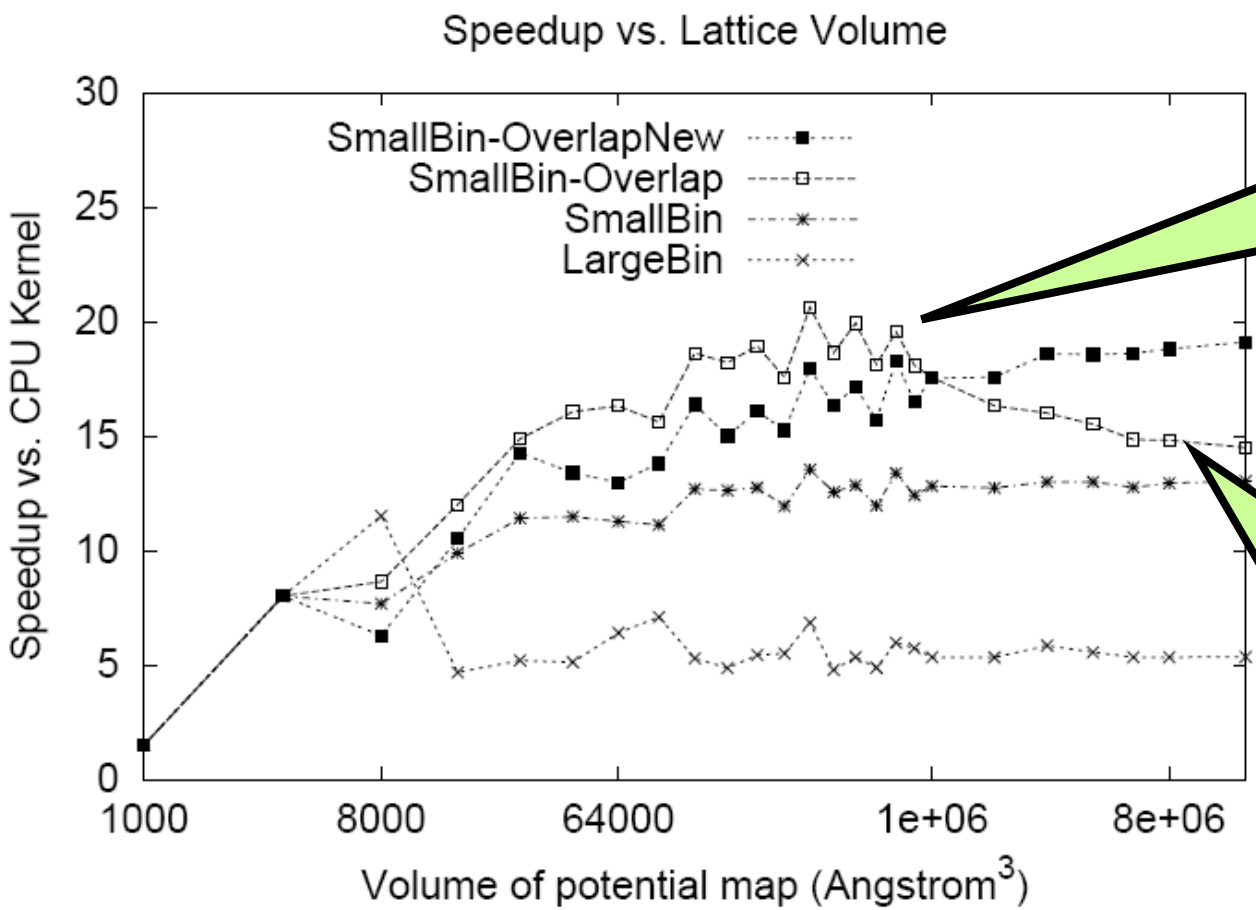
Look-up table encodes "logic" of spatial geometry



Using the CPU to Improve GPU Performance

- GPU performs best when the work evenly divides into the number of threads/processing units
- Optimization strategy:
  - Use the CPU to “regularize” the GPU workload
  - Use fixed size bin data structures, with “empty” slots skipped or producing zeroed out results
  - Handle exceptional or irregular work units on the CPU while the GPU processes the bulk of the work
  - On average, the GPU is kept highly occupied, attaining a much higher fraction of peak performance

Cutoff Summation Runtime



GPU cutoff with CPU overlap: 17x-21x faster than CPU core

If asynchronous stream blocks due to queue filling, performance will degrade from peak…

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

Cutoff Summation Observations

- Use of CPU to handle overflowed bins is very effective, overlaps completely with GPU work
- Caveat: avoid overfilling the asynchronous stream queue with work, doing so can trigger blocking behavior (improved in current drivers)
- The use of compensated summation (all GPUs) or double-precision (GT200 only) for potential accumulation resulted in only a **~10%** performance penalty vs. pure single-precision arithmetic, while reducing the effects of floating point truncation

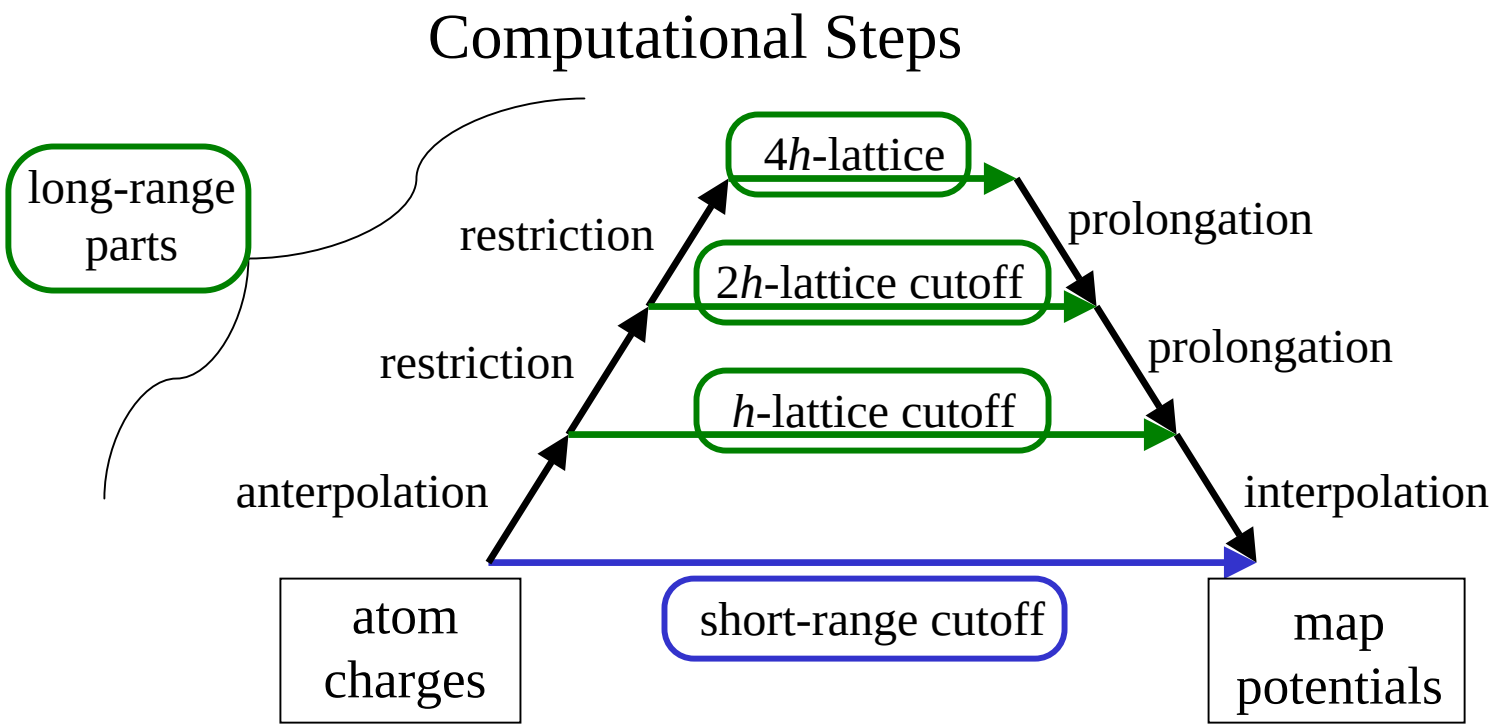
Multilevel Summation

- Approximates full electrostatic potential
- Calculates sum of smoothed pairwise potentials interpolated from a hierarchy of lattices
- Advantages over PME and/or FMM:
  - Algorithm has **linear time complexity**
  - Permits non-periodic and periodic boundaries
  - Produces continuous forces for dynamics (advantage over FMM)
  - Avoids 3-D FFTs for better parallel scaling (advantage over PME)
  - Spatial separation allows use of multiple time steps
  - Can be extended to other pairwise interactions

- Skeel, Tezcan, Hardy, *J Comp Chem*, 2002 — Computing forces for molecular dynamics
- Hardy, Stone, Schulten, *J Paral Comp*, 2009 — GPU-acceleration of potential map calculation

Multilevel Summation Calculation

map potential	=	**exact short-range interactions**	+	**interpolated long-range interactions**

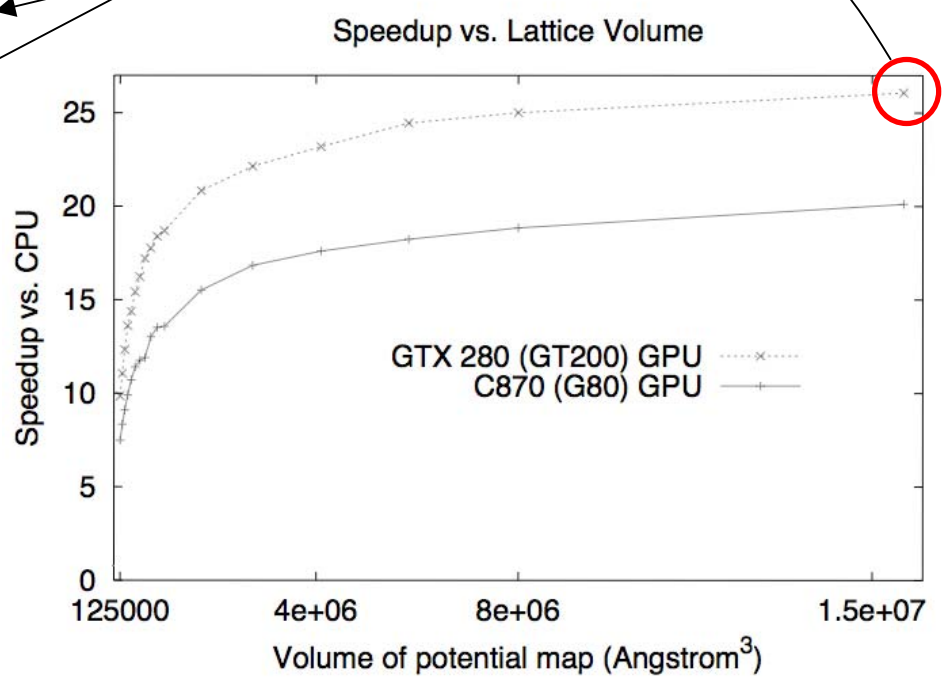


Multilevel Summation on the GPU

Accelerate **short-range cutoff** and **lattice cutoff** parts

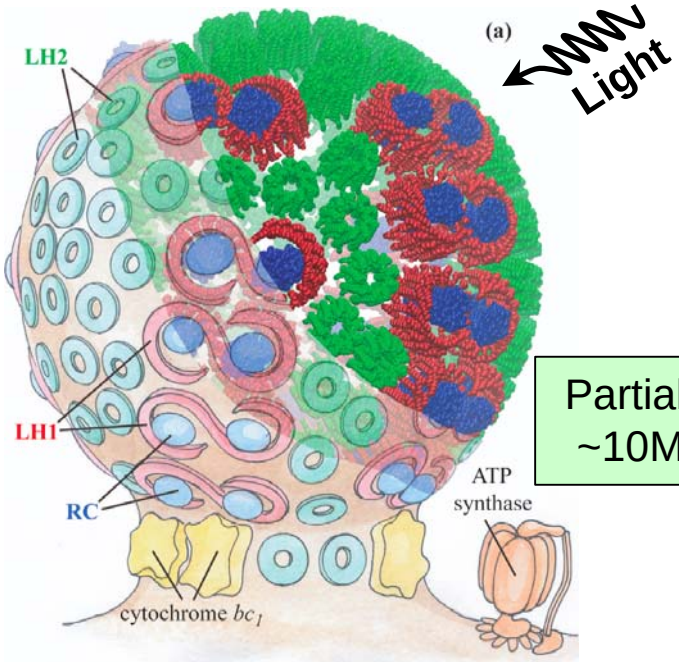
Performance profile for 0.5 Å map of potential for 1.5 M atoms.
Hardware platform is Intel QX6700 CPU and NVIDIA GTX 280.

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

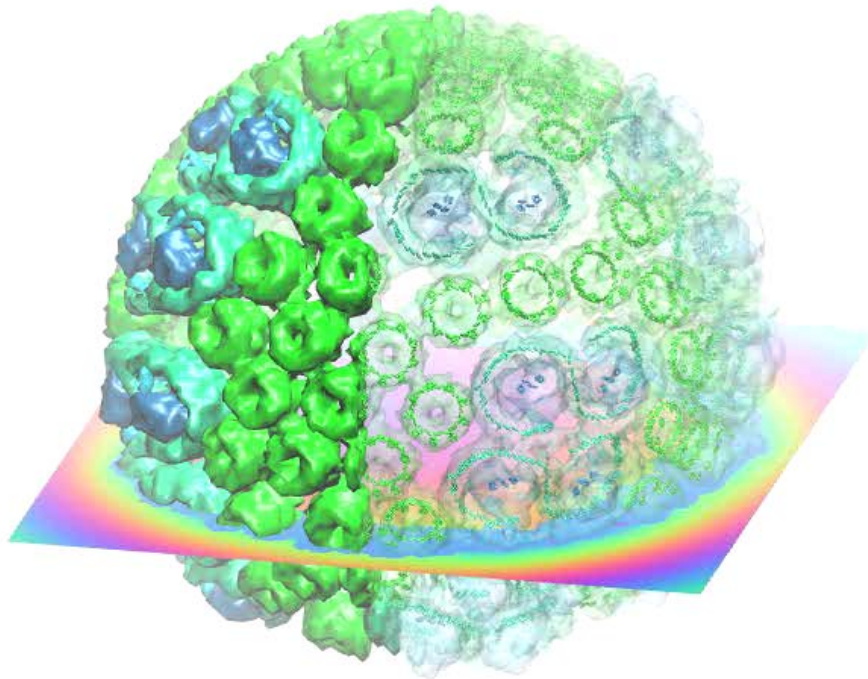


Photobiology of Vision and Photosynthesis

Investigations of the chromatophore, a photosynthetic organelle



Partial model:
~10M atoms



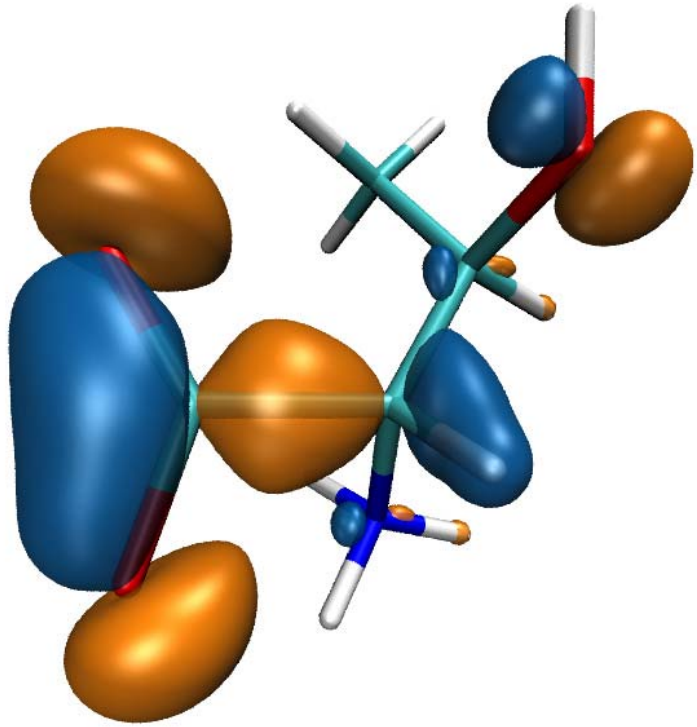
Electrostatics needed to build full structural model, place ions, study macroscopic properties

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

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

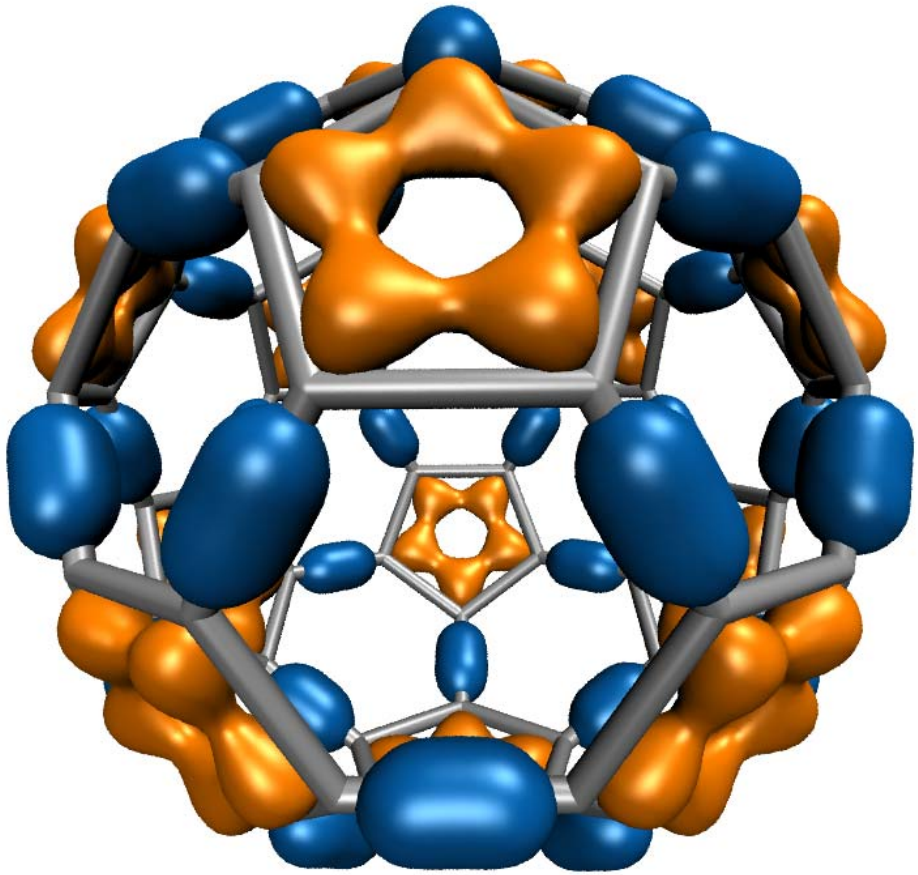
Molecular Orbitals

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



Computing Molecular Orbitals

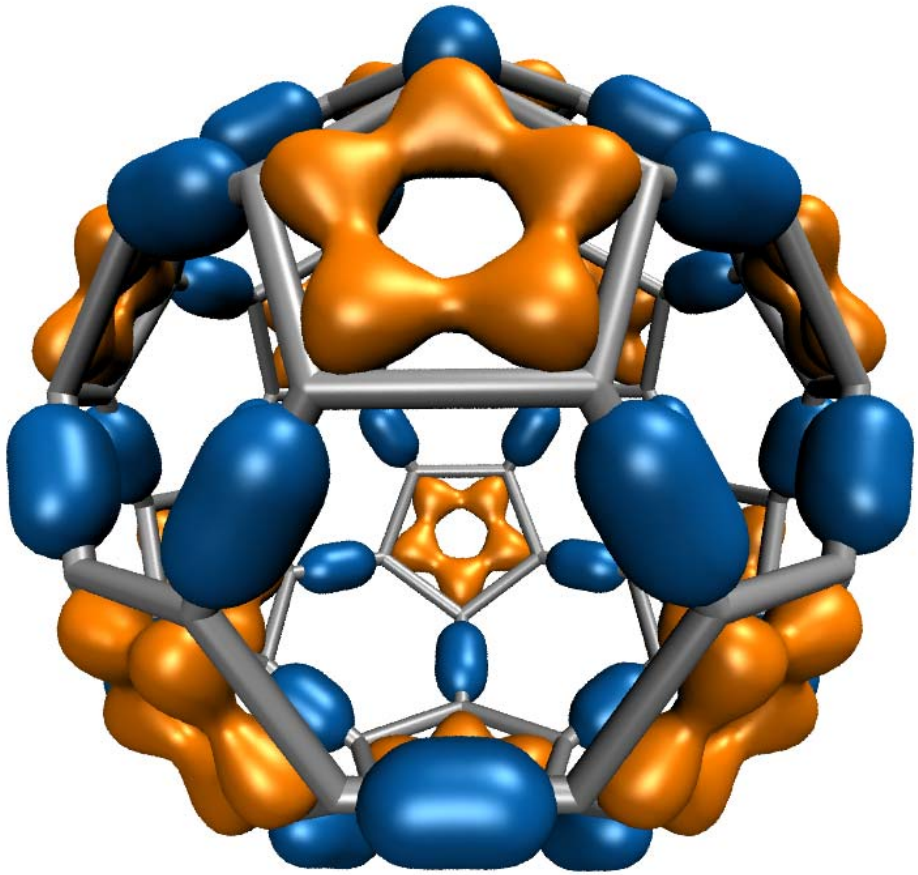
- Calculation of high resolution MO grids can require tens to hundreds of seconds in existing tools
- Existing tools cache MO grids as much as possible to avoid recomputation:
  - Doesn't eliminate the wait for initial calculation, hampers interactivity
  - Cached grids consume 100x-1000x more memory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



C_{60}

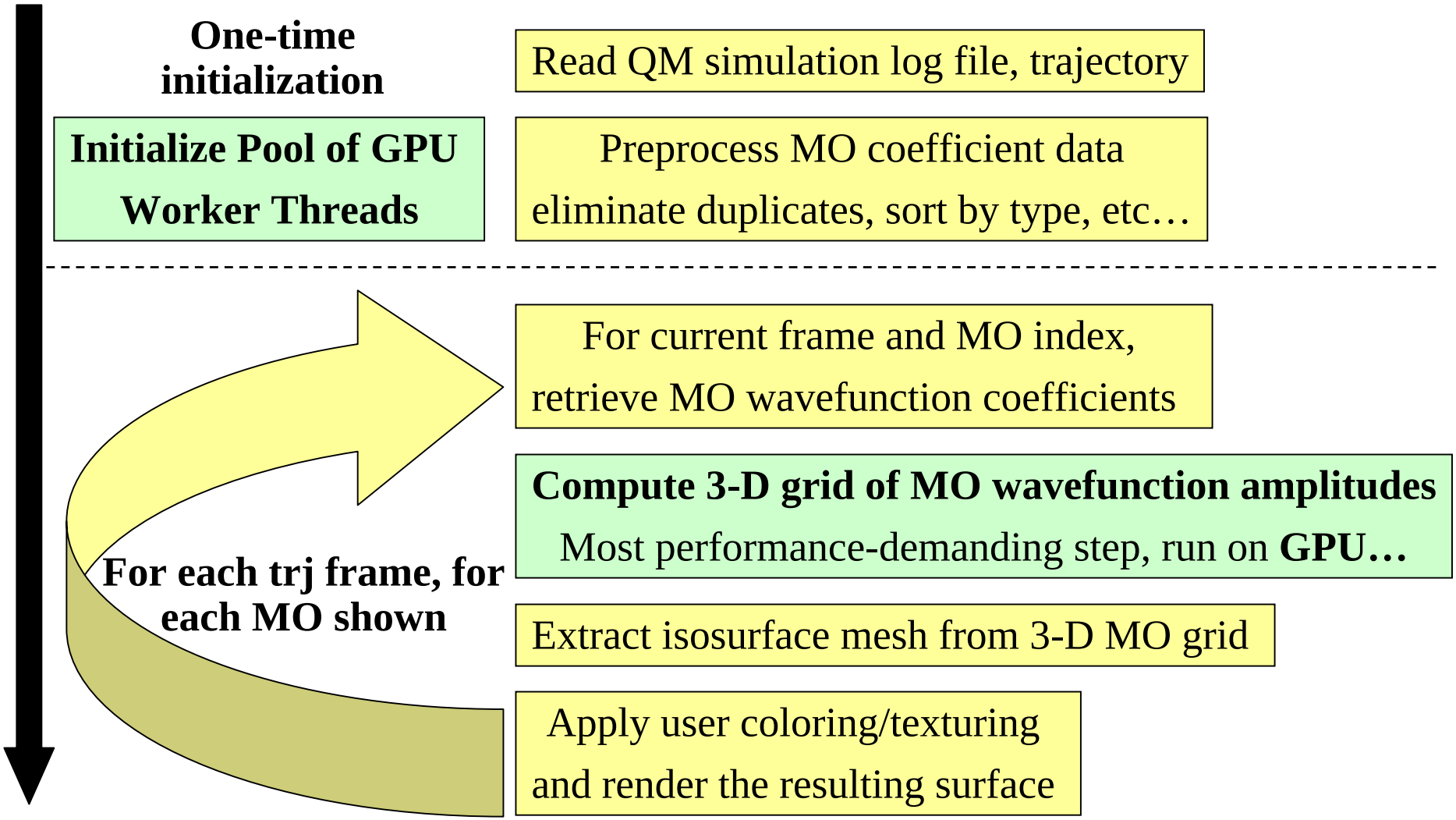
Animating Molecular Orbitals

- 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!



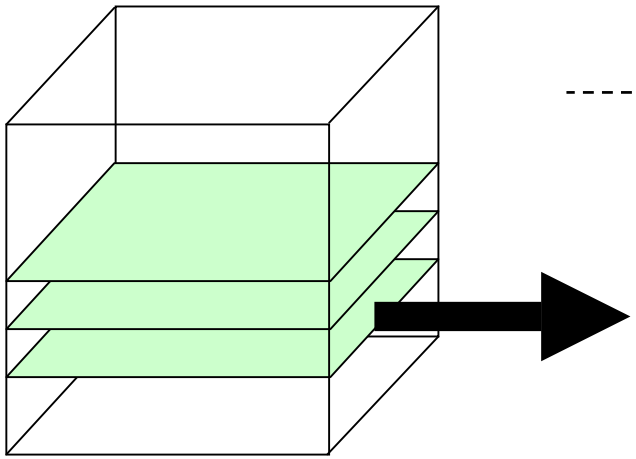
C_{60}

Molecular Orbital Computation and Display Process

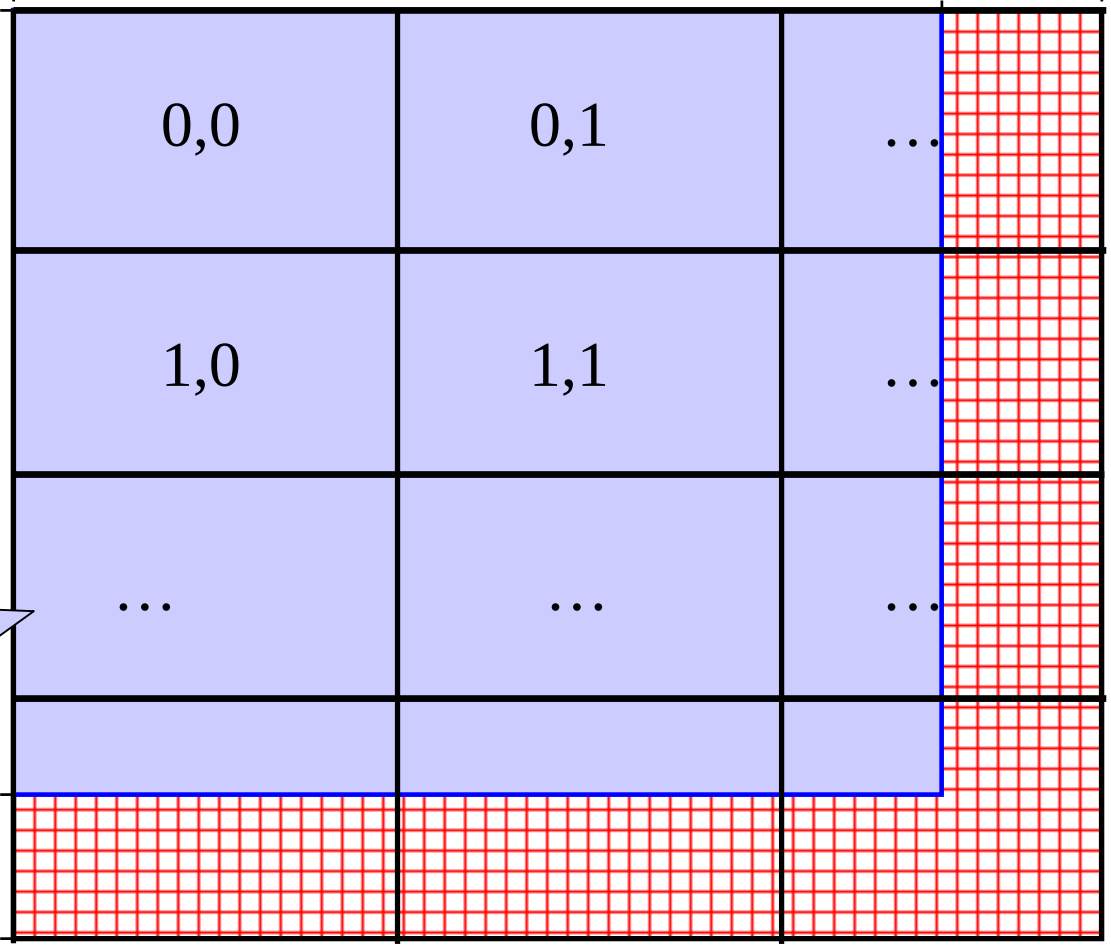


CUDA Block/Grid Decomposition

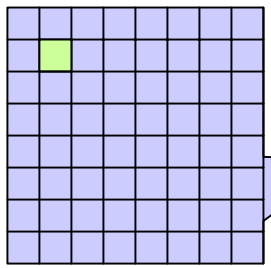
MO 3-D lattice decomposes into 2-D slices (CUDA grids)



Grid of thread blocks:



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



Padding optimizes glob. mem perf, guaranteeing coalescing

MO Kernel for One Grid Point (Naive C)

```
…
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()

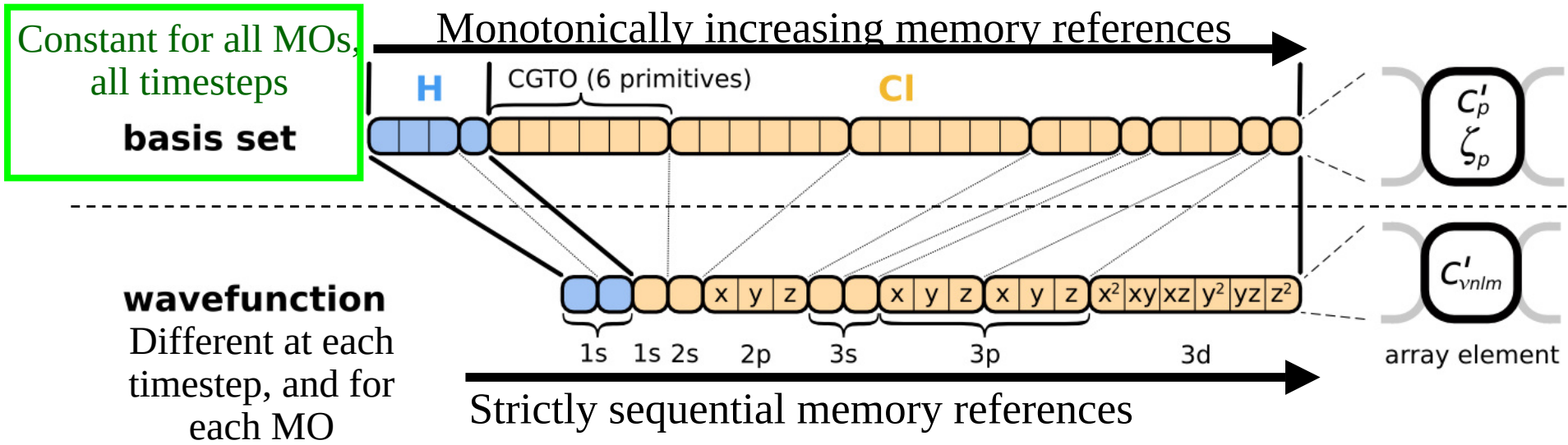
Loop over angular momenta

(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

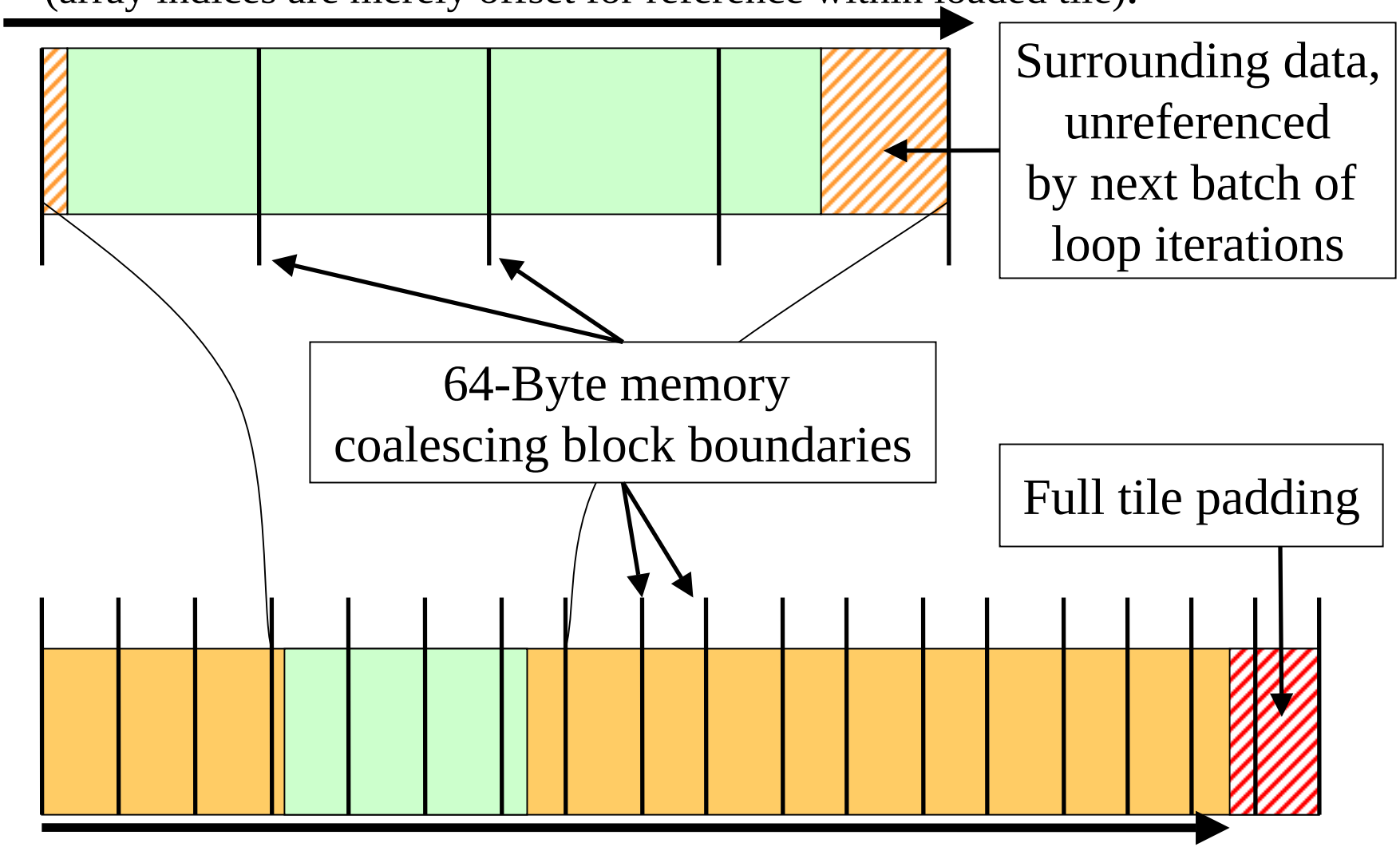


- Loop iterations always access same or consecutive array elements for all threads in a thread block:
  - Yields good constant memory cache performance
  - Increases shared memory tile reuse

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).



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 $1/8^{th}$ 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

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)**

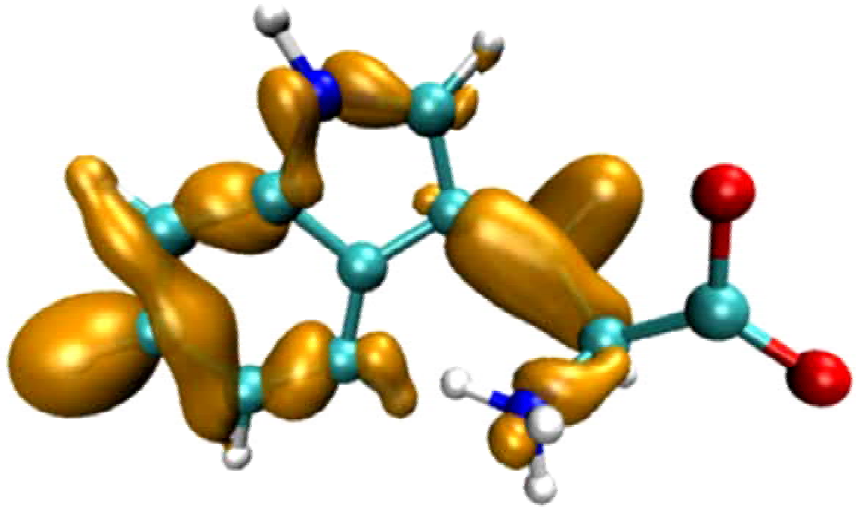
Kernel	**Cores GPUs**	Speedup vs. Molekel on 1 CPU core					
Molekel	1*	1.0	1.0	1.0	1.0	1.0	1.0
MacMolPlt	4	2.4	2.6	2.1	2.4	4.3	4.5
VMD GCC-cephes	4	3.2	4.0	3.0	3.5	4.3	6.5
VMD ICC-SSE-cephes	4	16.8	17.2	13.9	12.6	17.3	21.5
VMD ICC-SSE-approx**	4	59.3	53.4	50.4	49.2	54.8	69.8
VMD CUDA-const-cache	1	552.3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

VMD 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**

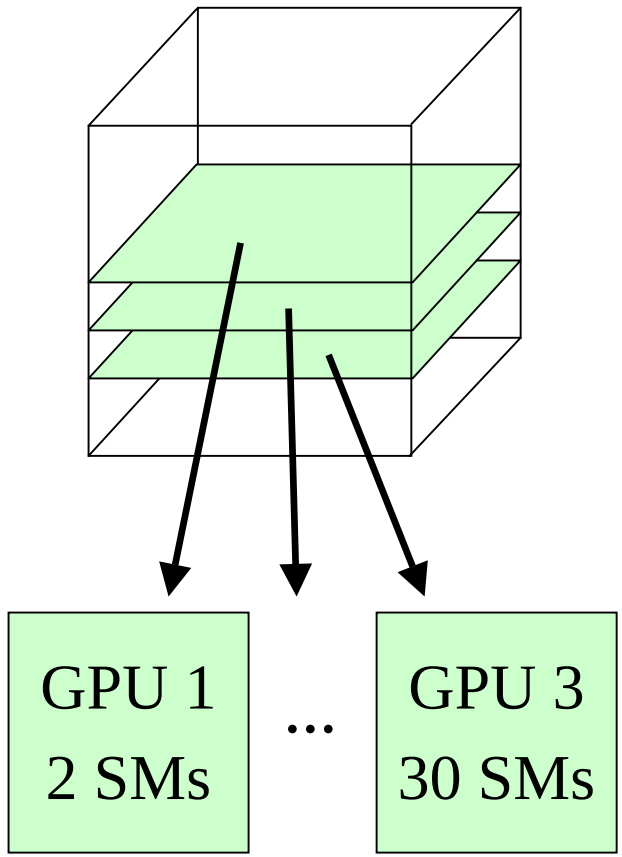


threonine

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



VMD Multi-GPU Molecular Orbital Performance Results for C_{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

MO Kernel Structure, Opportunity for JIT…

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

Loop over atoms (1 to ~200) {

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

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

Unpredictable (at compile-time, since data-driven) but small loop trip counts result in significant loop overhead. **Dynamic kernel generation and JIT compilation can unroll entirely, resulting in 40% speed boost**

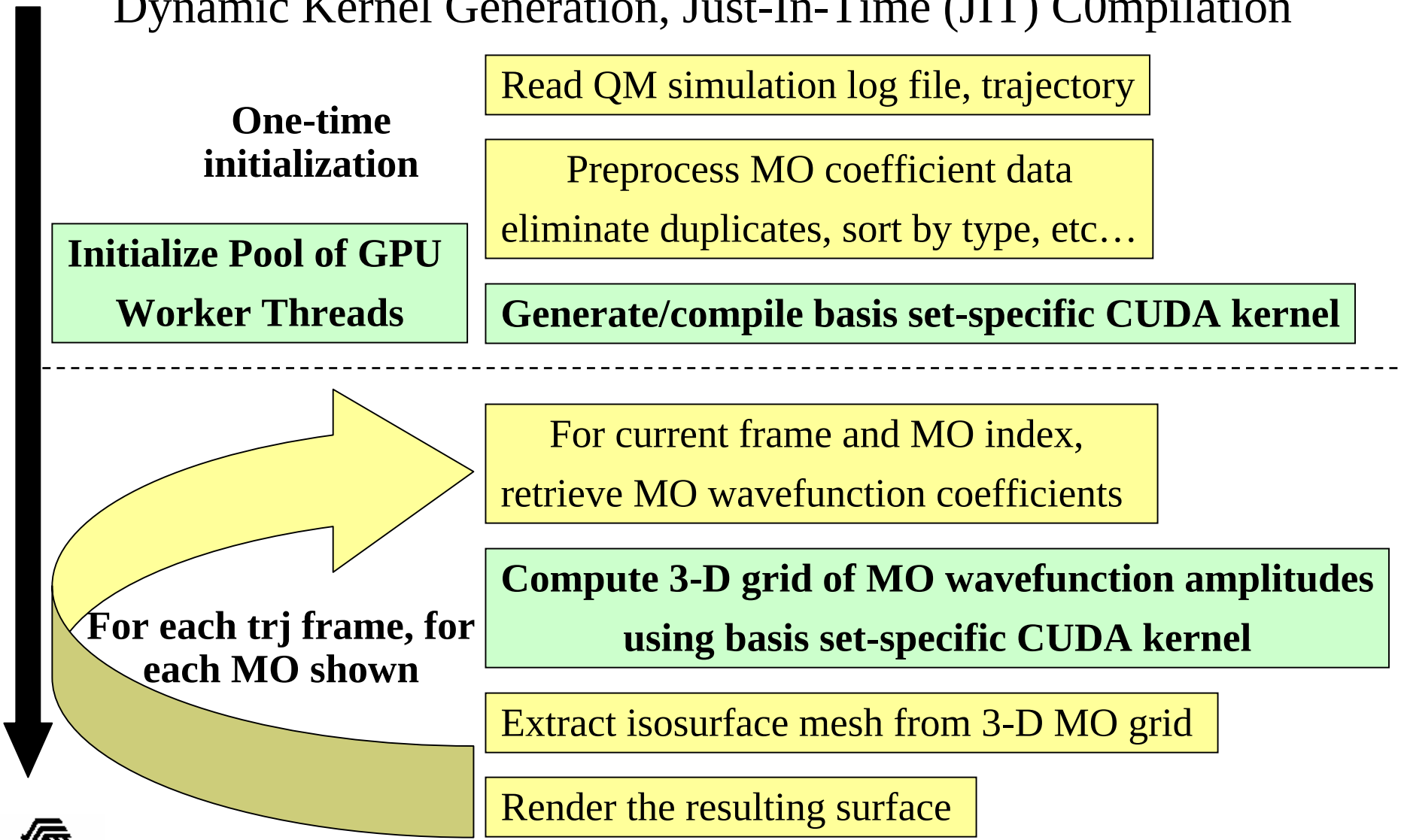
}

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

}

}

Molecular Orbital Computation and Display Process
Dynamic Kernel Generation, Just-In-Time (JIT) 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 kernel

Dynamically-generated CUDA kernel (JIT)

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

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

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

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

Lessons Learned

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

Lessons Learned (2)

- The host CPU can potentially be used to "regularize" the computation for the GPU, yielding better overall performance
- Overlapping CPU work with GPU can hide some communication and unaccelerated computation
- Targeted use of double-precision floating point arithmetic, or compensated summation can reduce the effects of floating point truncation at low cost to performance

Summary

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

Acknowledgements

- 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 Center of Excellence, University of Illinois at Urbana-Champaign
- NCSA Innovative Systems Lab
- David Kirk and the CUDA team at NVIDIA
- NIH support: P41-RR05969