High Performance Molecular Visualization and Analysis with GPU Computing

John Stone

Theoretical and Computational Biophysics Group

Beckman Institute for Advanced Science and Technology

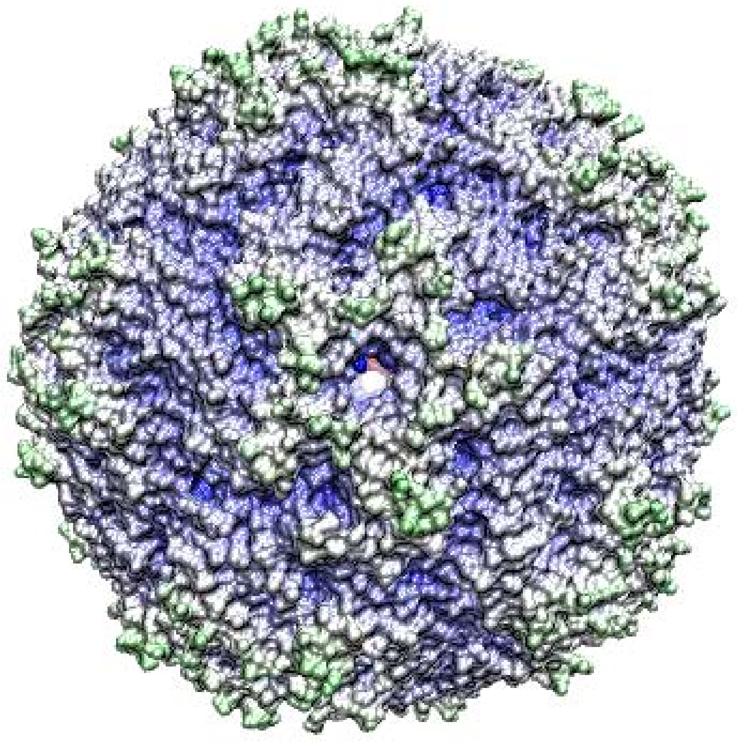
University of Illinois at Urbana-Champaign

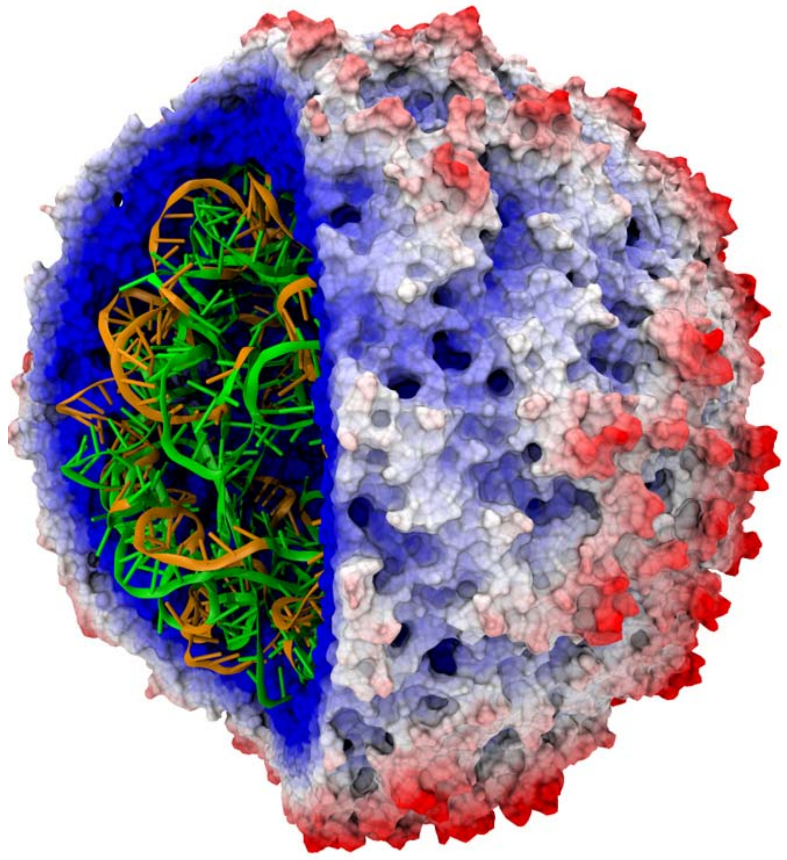
http://www.ks.uiuc.edu/Research/gpu/

BI Imaging and Visualization Forum, October 20, 2009

VMD – “Visual Molecular Dynamics”

- High performance molecular visualization and analysis
- User extensible with scripting and plugins
- http://www.ks.uiuc.edu/Research/vmd/





National Center for
Research Resources

VMD Handles Diverse Data

Whole cell as particle system

Atomic, CG, Particle, QM

Coordinates, Trajectories, Energies, Forces, Secondary Structure, Wavefunctions, …

Efficiency, Performance, Capacity

Load MD trajectories @ ~1GB/sec

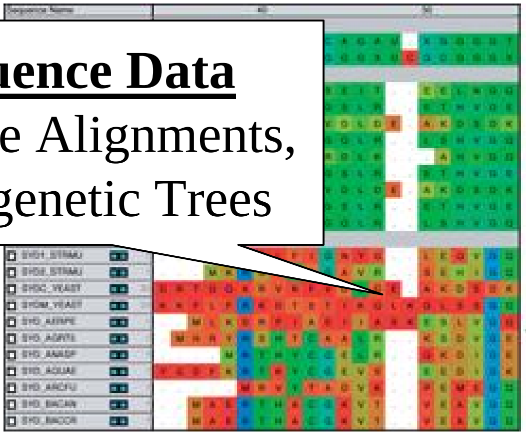
Improved disk storage efficiency: ~5GB for 100M atom model

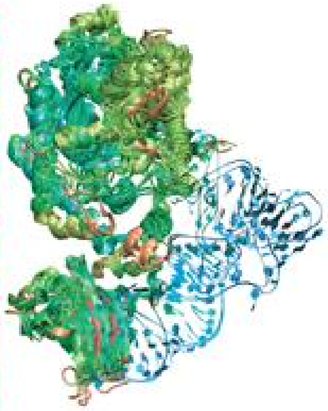
Model size limited only by RAM

Graphics, Geometry

Annotations

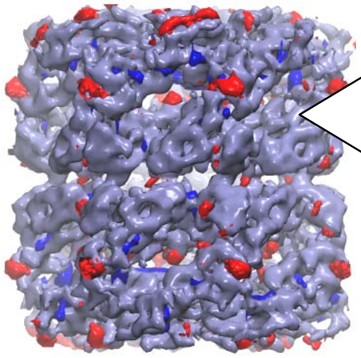
VMD





Sequence Data

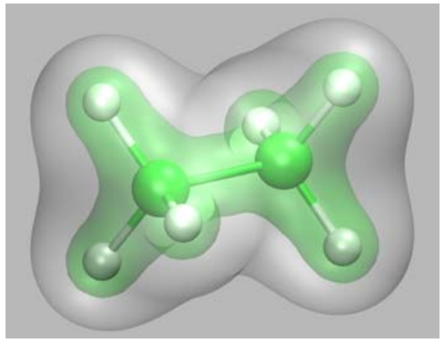
Multiple Alignments, Phylogenetic Trees



GroEL

Volumetric Data

Cryo-EM density maps, Electron orbitals, Electrostatic potential, MRI scans, …

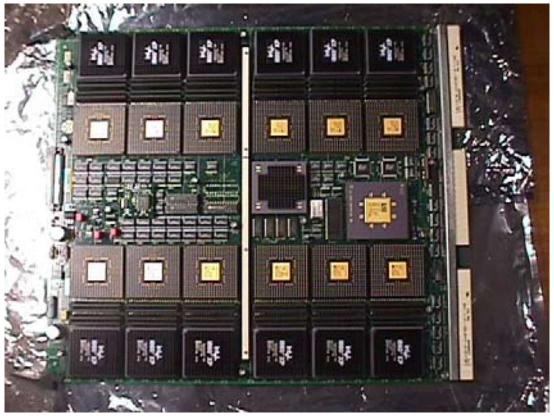


Ethane

Programmable Graphics Hardware Evolution

- 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**

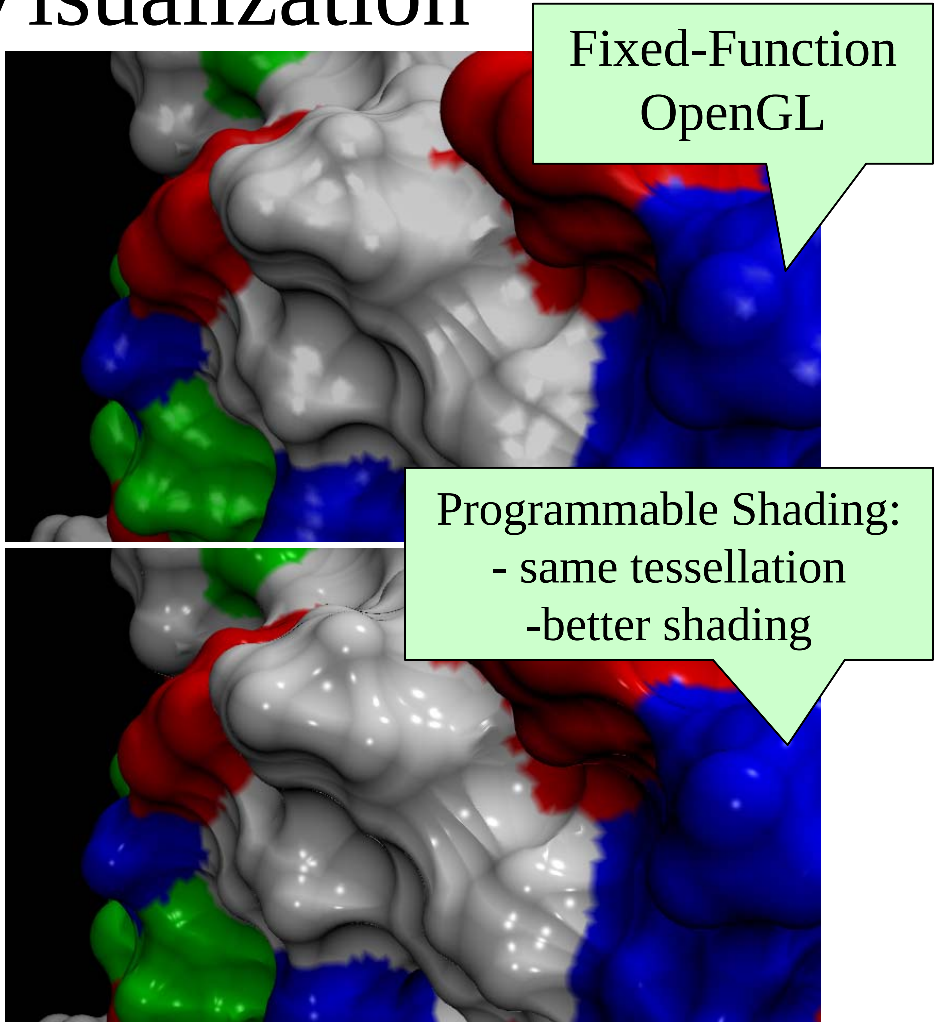
UNC PixelFlow Rack



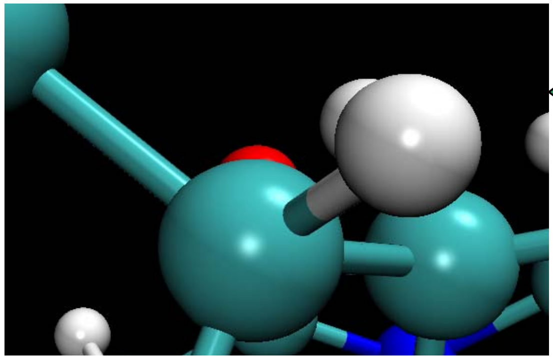
SGI Reality Engine i860 Vertex Processors

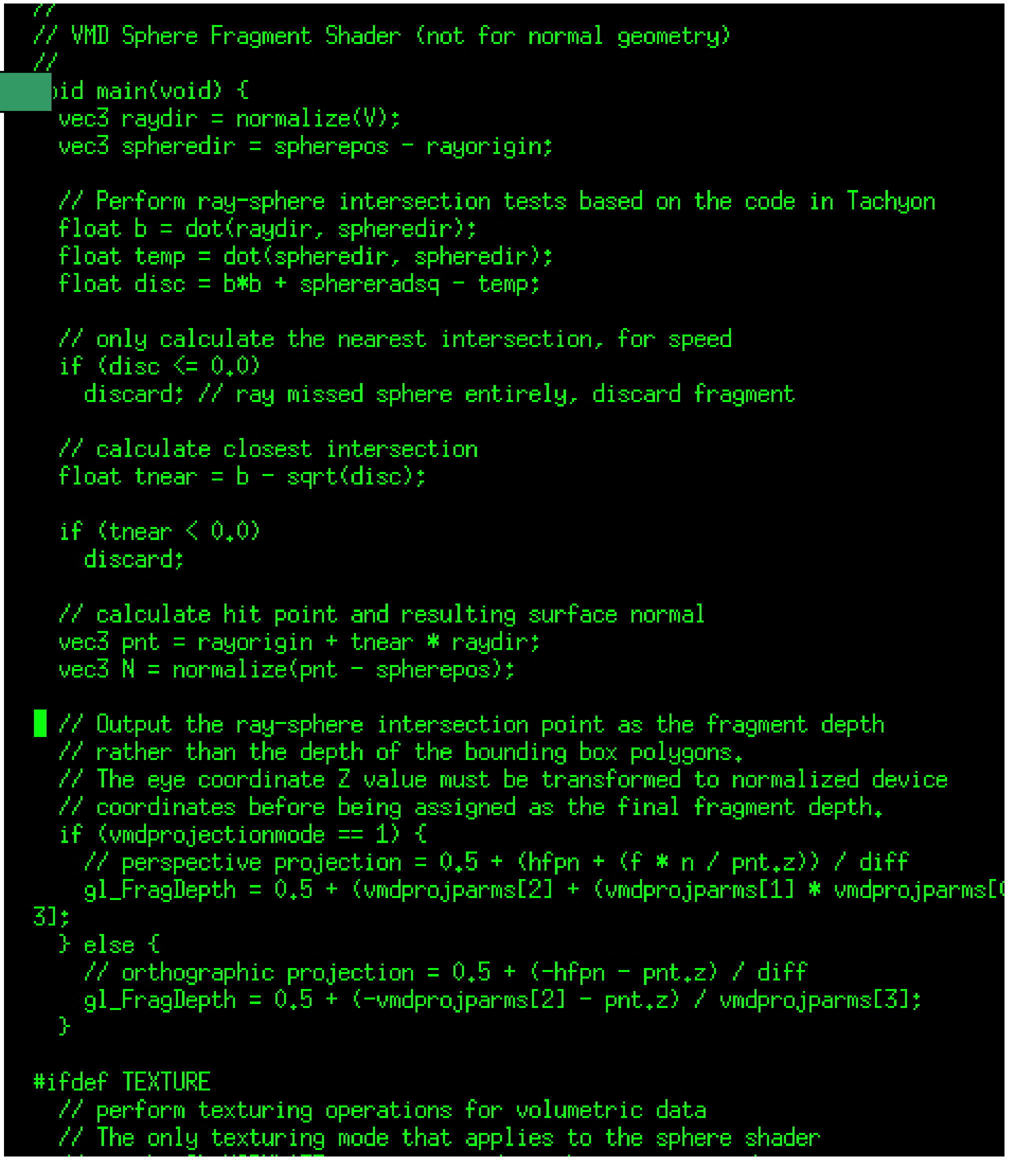
Benefits of Programmable Shading for Molecular Visualization

- Potential for superior image quality with better shading algorithms
- Direct rendering of curved surfaces
- Render density map data, solvent surfaces
- Offload work from host CPU to GPU



VMD Ray Traced Sphere Shader





```
//
// VMD Sphere Fragment Shader (not for normal geometry)
//
id 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
```

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

“GPGPU” and GPU Computing

- Although graphics-specific, programmable shading languages were (ab)used by early researchers to experiment with using GPUs for general purpose parallel computing, known as “GPGPU”
- Compute-specific GPU languages such as CUDA and OpenCL have eliminated the need for graphics expertise in order to use GPUs for general purpose computation!

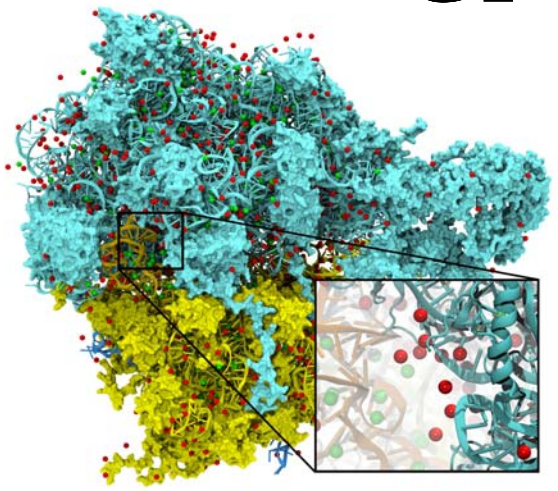
GPU Computing

- Current GPUs provide over >1 TFLOPS of arithmetic capability!
- Massively parallel hardware, hundreds of processing units, throughput oriented architecture
- Commodity devices, omnipresent in modern computers (over a **million GPUs sold per week**)
- 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

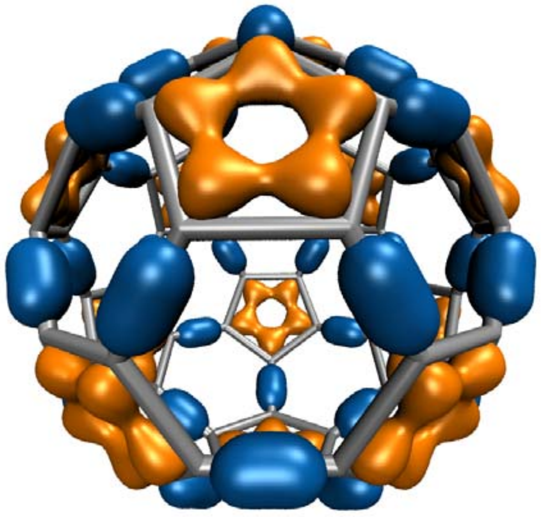
What Speedups Can GPUs Achieve?

- 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
- GPU acceleration provides an opportunity to make **slow, or batch** calculations capable of being run **interactively, on-demand…**

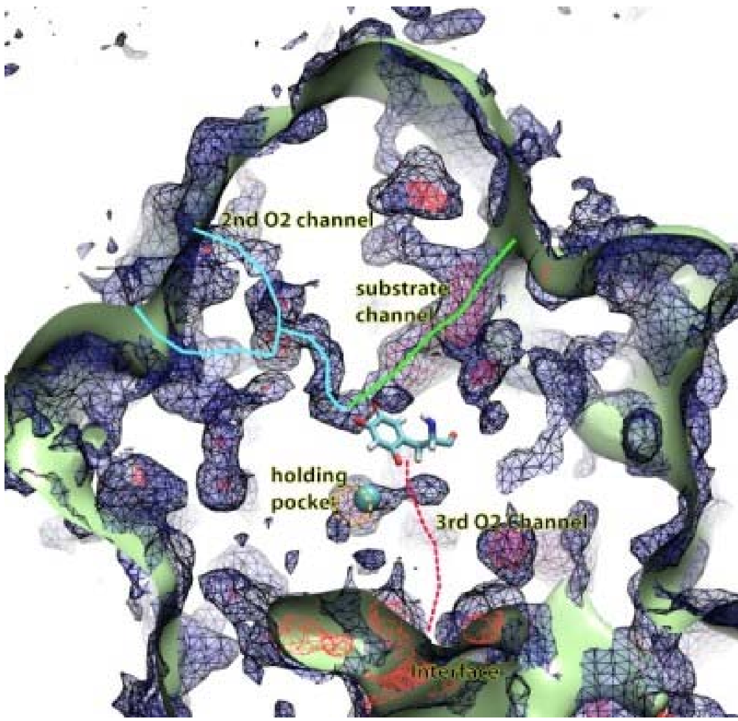
GPU Computing in VMD



Electrostatic field calculation, ion placement: factor of 20x to 44x faster

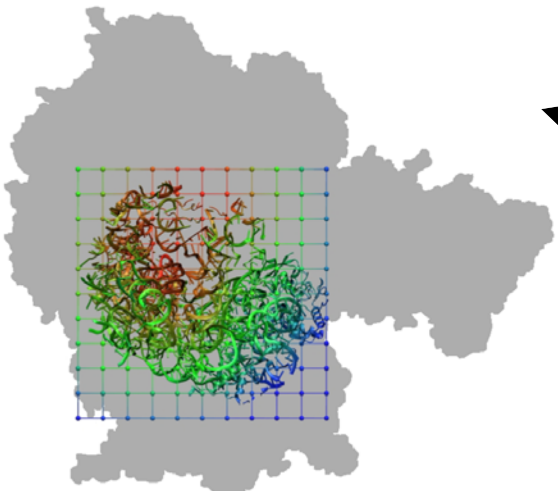


Molecular orbital calculation and display: factor of 120x faster



Imaging of gas migration pathways in proteins with implicit ligand sampling:

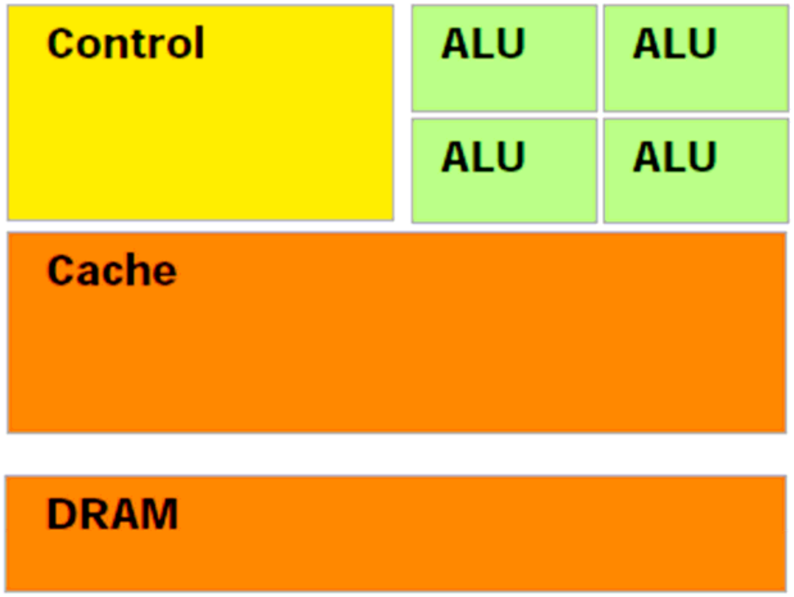
factor of 20x to 30x faster

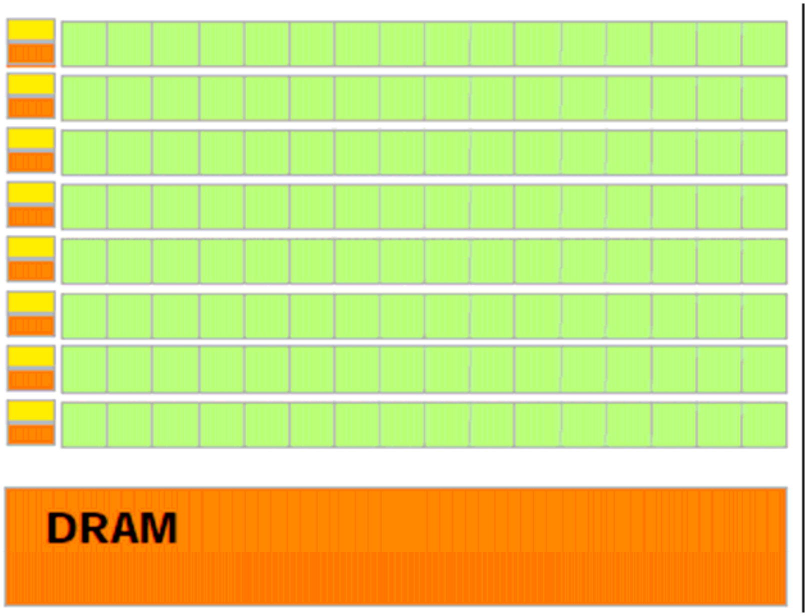


Comparison of CPU and GPU Hardware Architecture

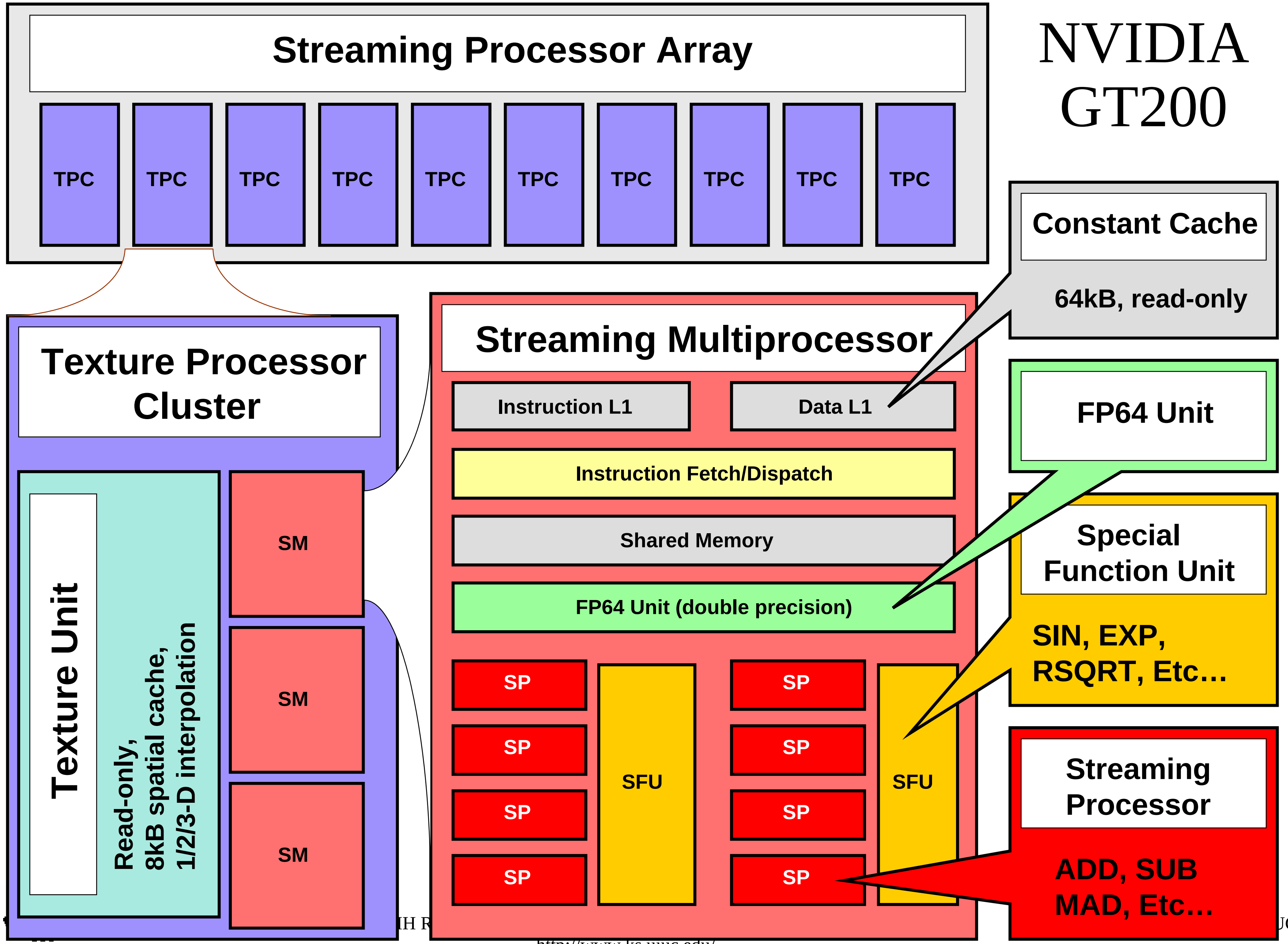
CPU: Cache heavy, focused on individual thread performance

GPU: ALU heavy, massively parallel, throughput oriented

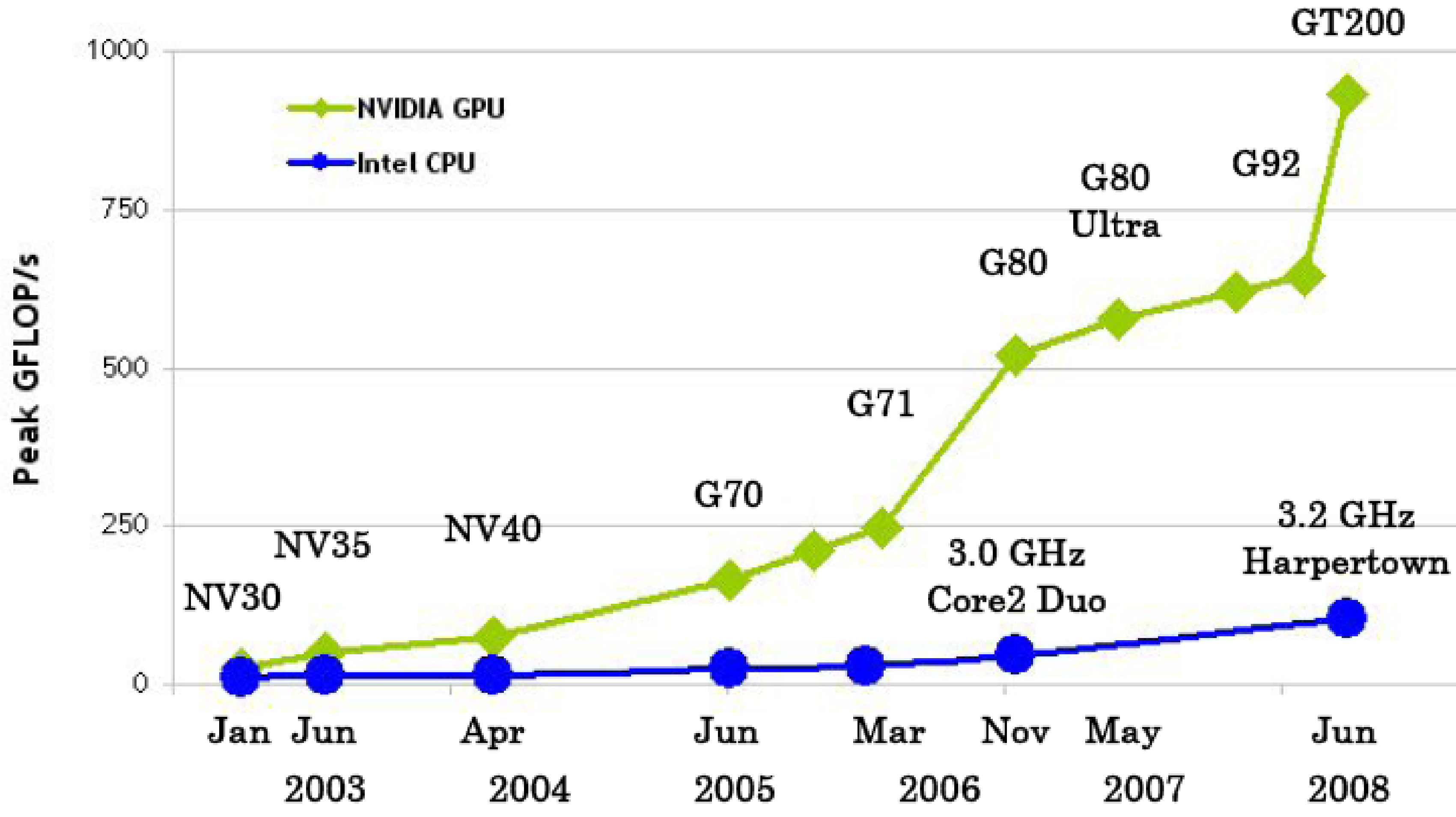




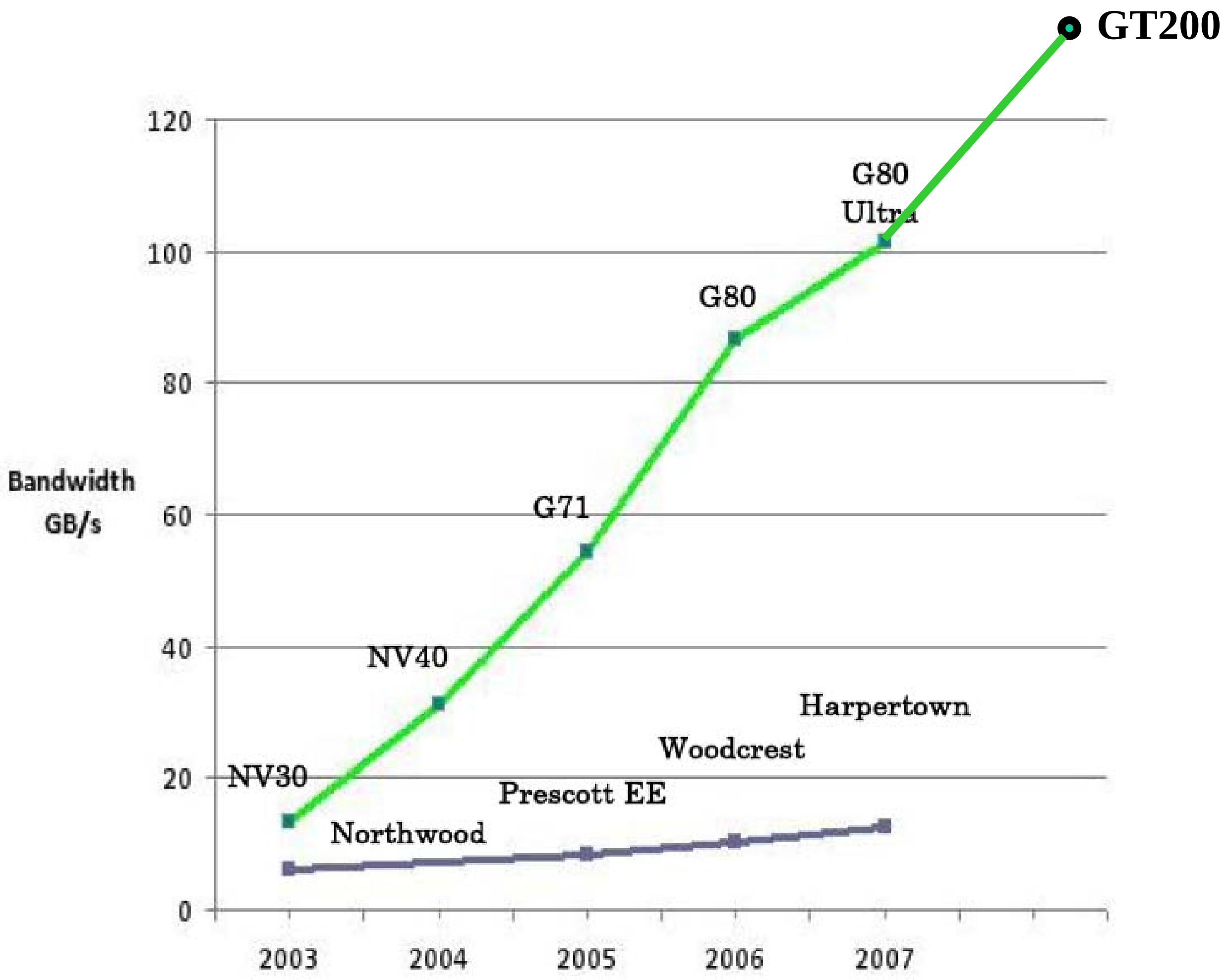
NVIDIA GT200



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)

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

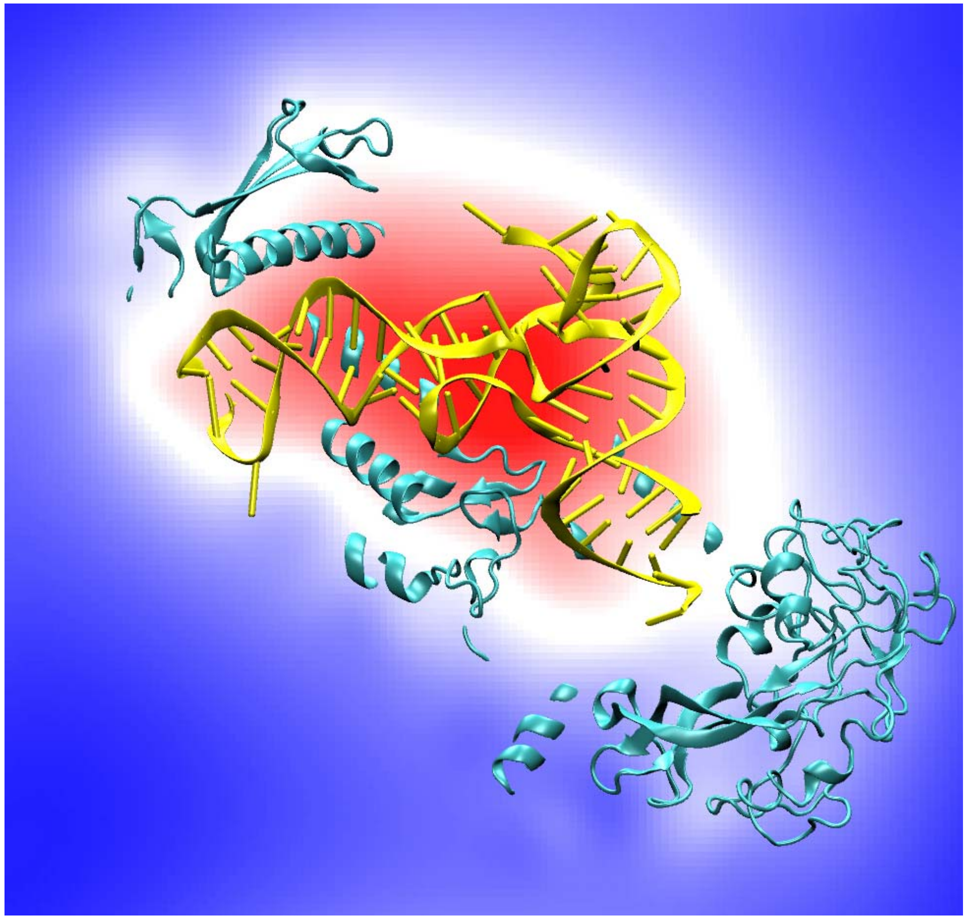
- Find an algorithm that exposes substantial parallelism, thousands of independent threads…
- Loops in a sequential code become a multitude of simultaneously executing threads organized into blocks of cooperating threads, and a grid of independent blocks…
- Identify appropriate GPU memory subsystems for storage of data used by kernel, design data structures accordingly
- Are there trade-offs that can be made to exchange computation for more parallelism?
  - "Brute force" methods that expose significant parallelism do surprisingly well on current GPUs

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

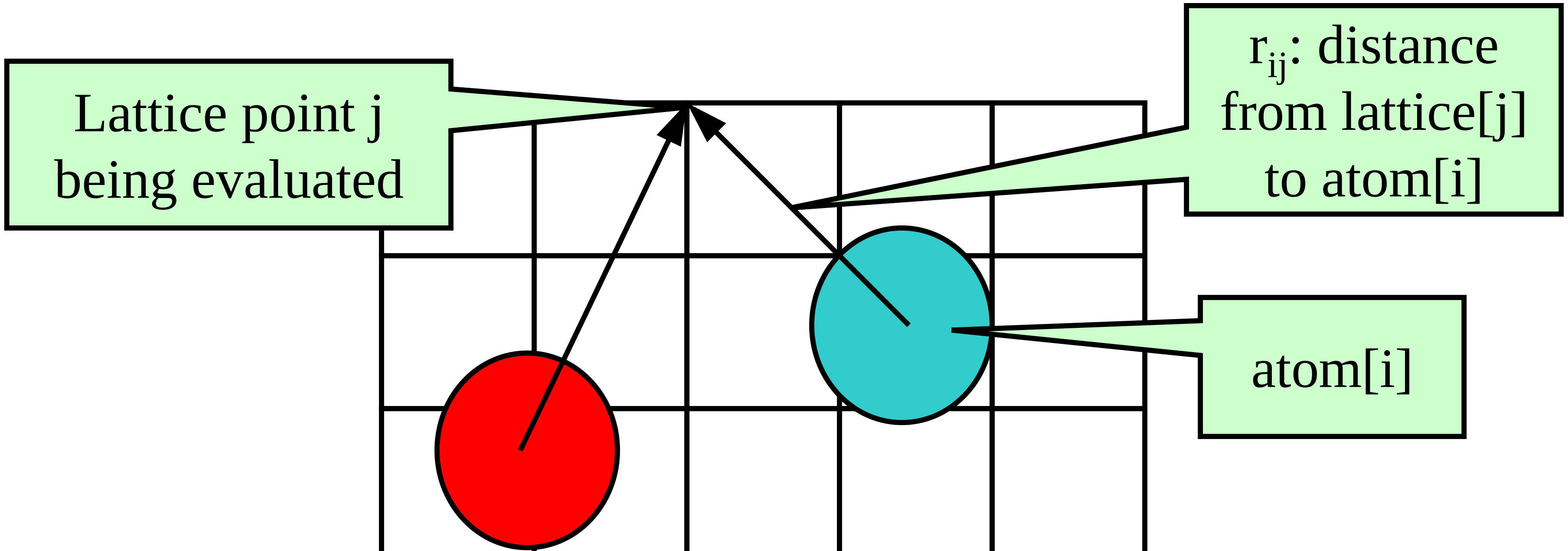


Isoleucine tRNA synthetase

Direct Coulomb Summation

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

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

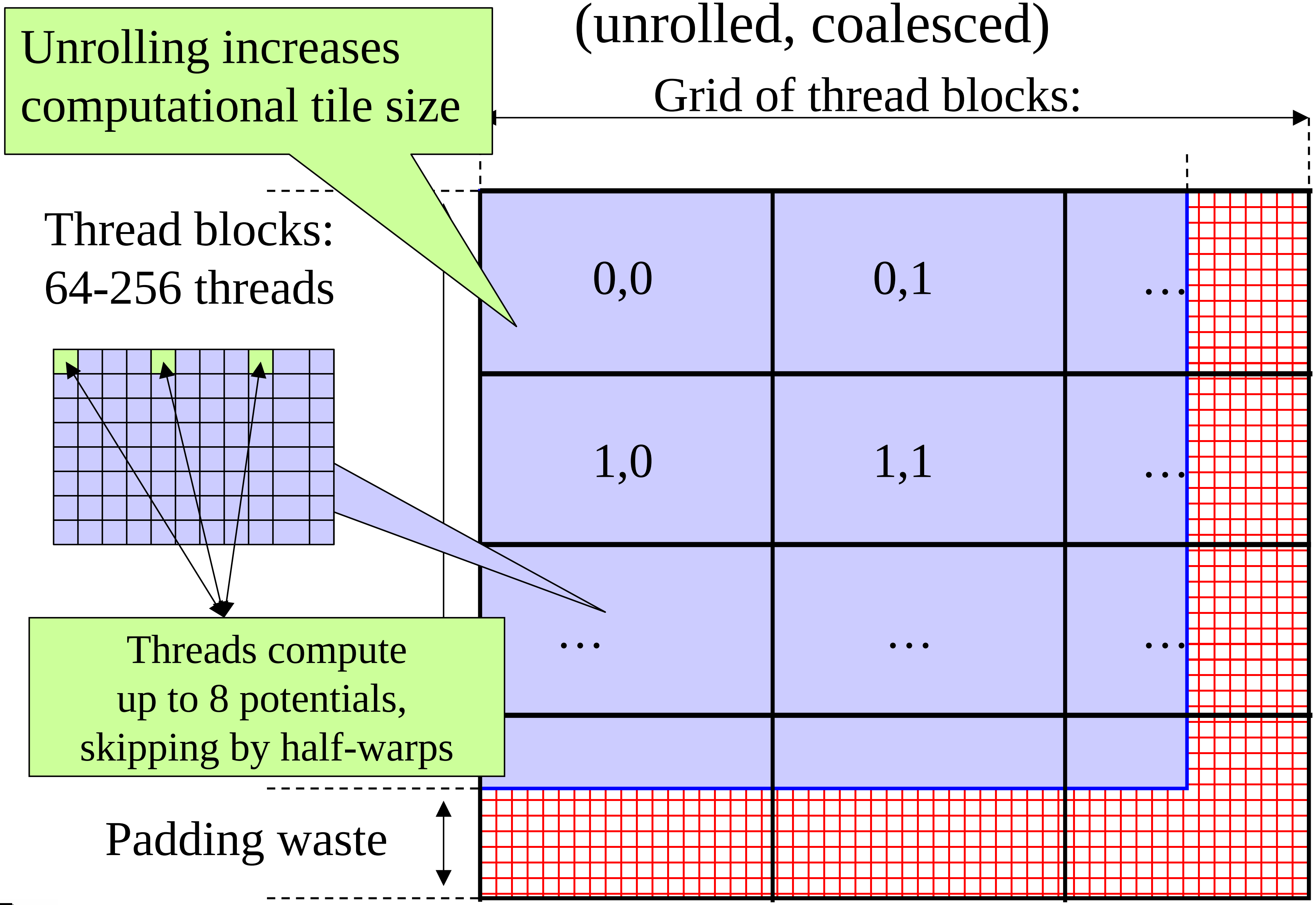

Lattice point j being evaluated
r_{ij}: distance from lattice[j] to atom[i]
atom[i]

Direct Coulomb Summation on the GPU

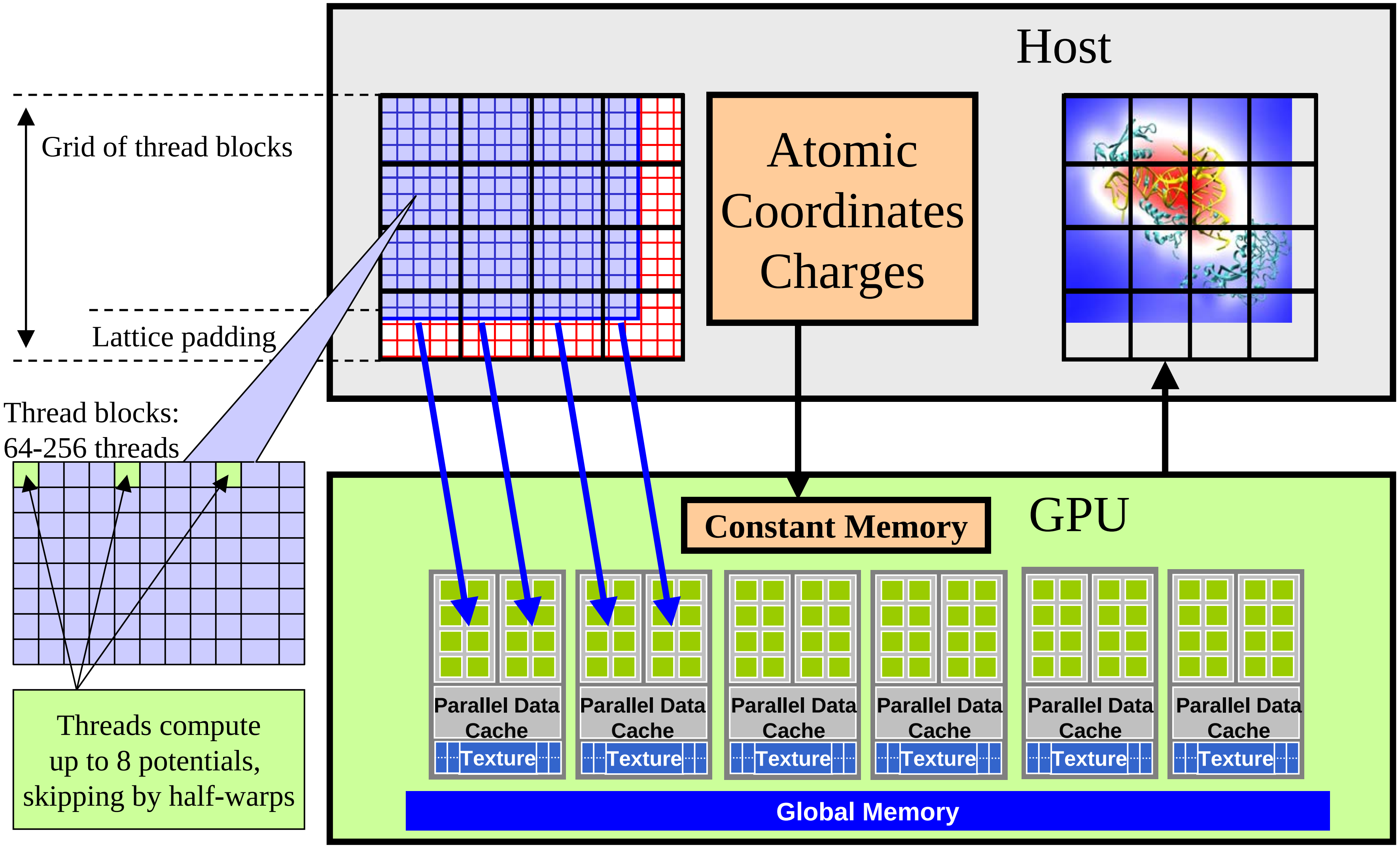
- GPU outruns a CPU core by 44x
- Work is decomposed into tens of thousands of independent threads, multiplexed onto hundreds of GPU processing units
- Single-precision FP arithmetic is adequate for intended application
- Numerical accuracy can be improved by compensated summation, spatially ordered 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

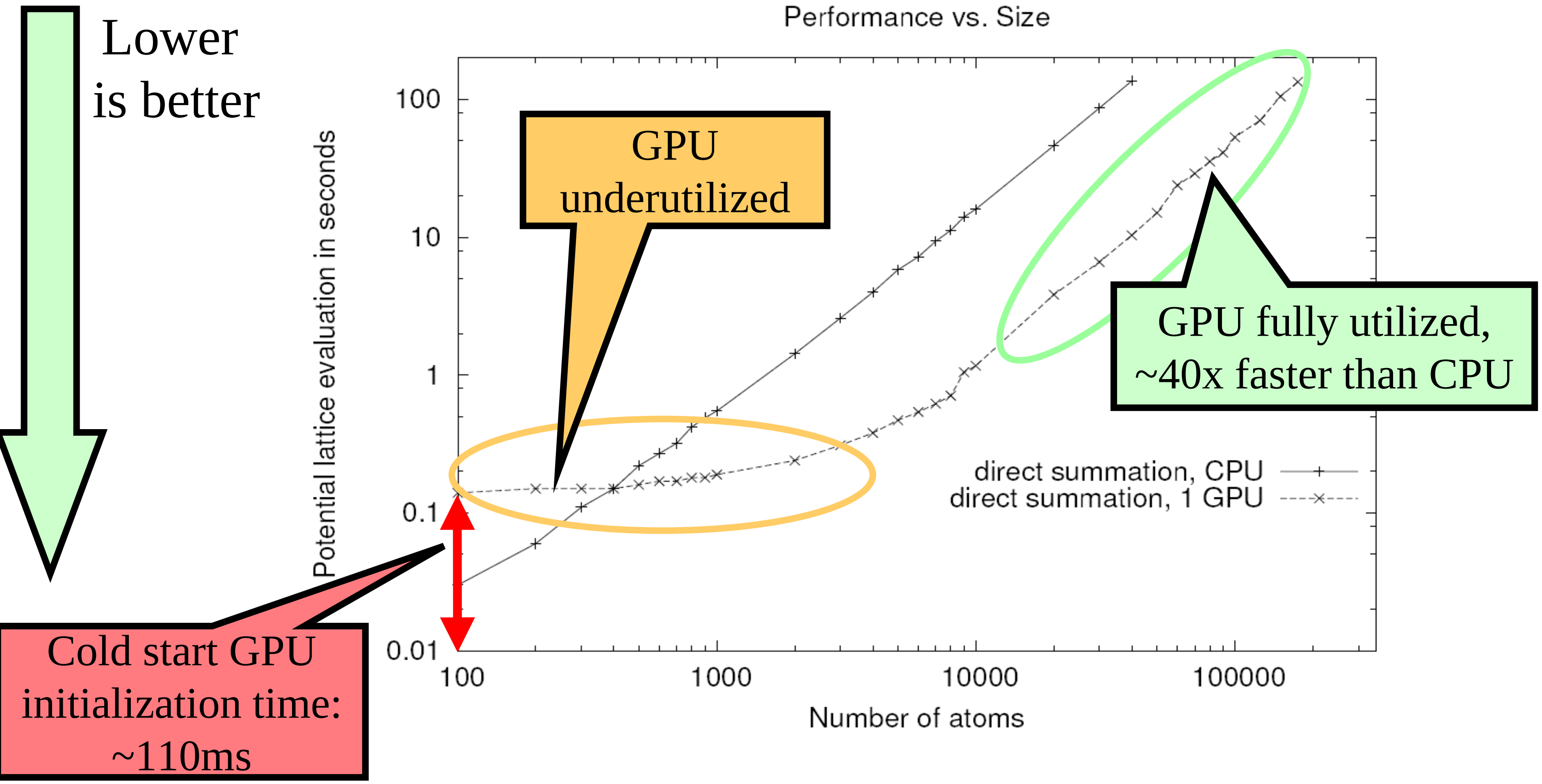
(unrolled, coalesced)


Unrolling increases computational tile size
Grid of thread blocks:
Thread blocks: 64-256 threads
0,0
0,1
…
1,0
1,1
…
…
…
…
Threads compute up to 8 potentials, skipping by half-warps
Padding waste

Direct Coulomb Summation on the GPU


Host
Grid of thread blocks
Lattice padding
Atomic Coordinates Charges
Thread blocks: 64-256 threads
Threads compute up to 8 potentials, skipping by half-warps
Constant Memory
GPU
Parallel Data Cache
Texture
Global Memory

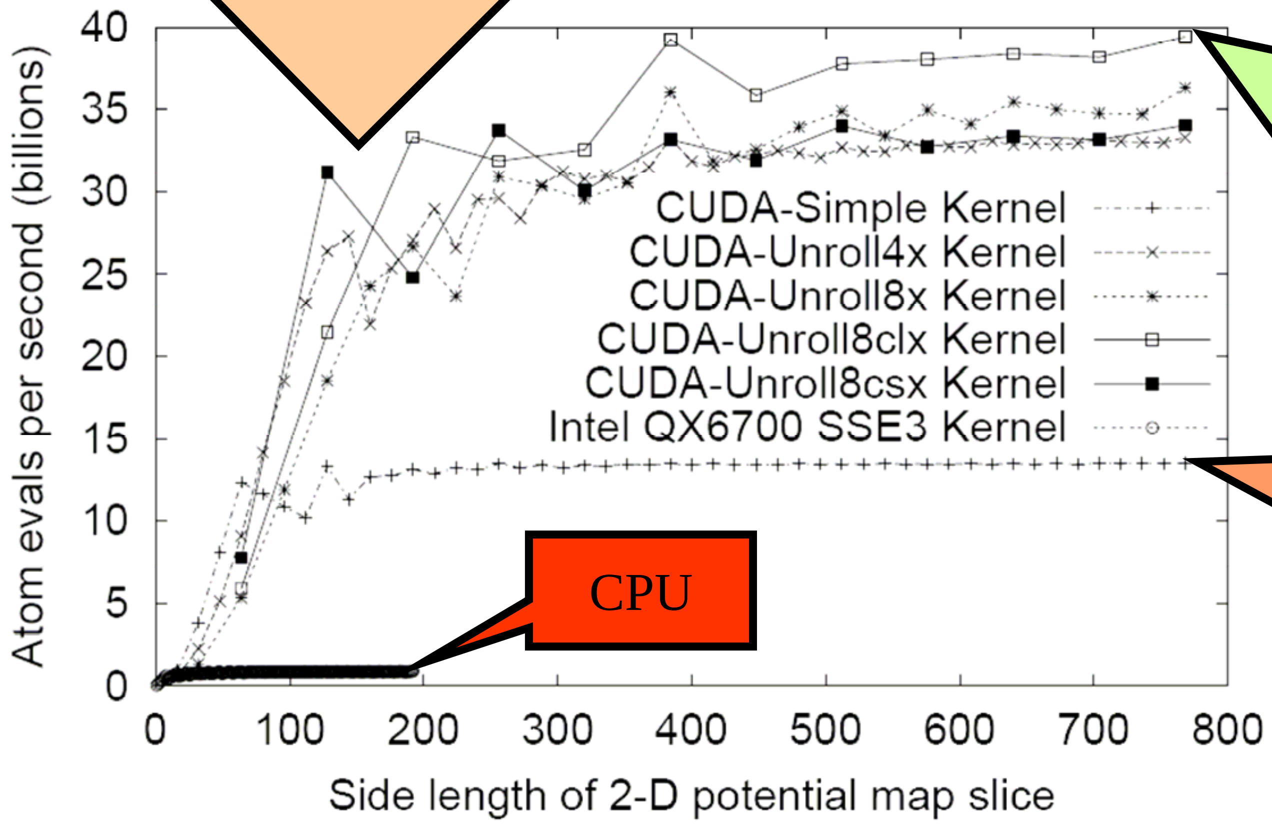
Direct Coulomb Summation Runtime



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

Direct Coulomb Summation Performance

Number of thread blocks modulo number of SMs results in significant performance variation for small workloads



CUDA-Unroll8clx: fastest GPU kernel, 44x faster than CPU, 291 GFLOPS on GeForce 8800GTX

CUDA-Simple: 14.8x faster, 33% of fastest GPU kernel

CPU

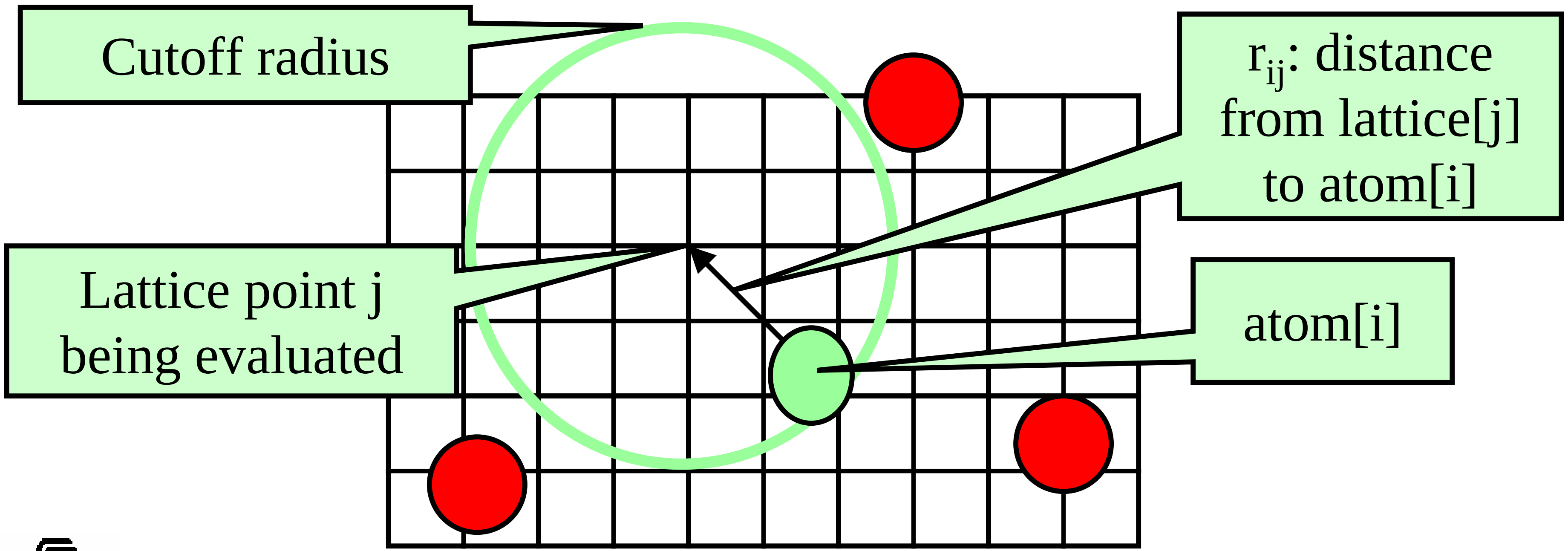
GPU computing. J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips. *Proceedings of the IEEE*, 96:879-899, 2008.

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 radius
r_{ij}: distance from lattice[j] to atom[i]
Lattice point j being evaluated
atom[i]

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,

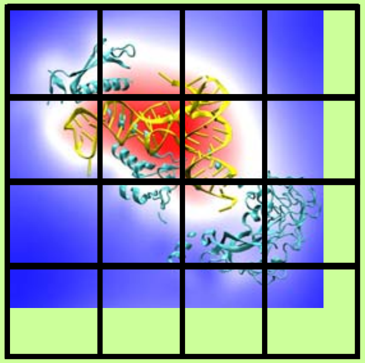
Bin/region neighbor checks costly; solved with universal table look-up

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

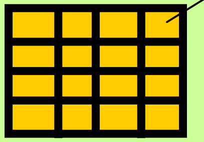
Shared memory

atom bin

Global memory



Potential map regions

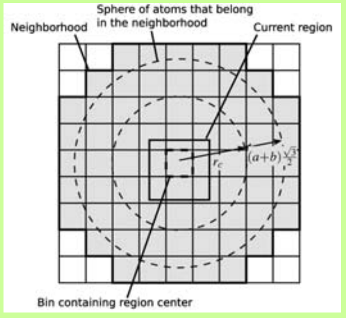


Bins of atoms

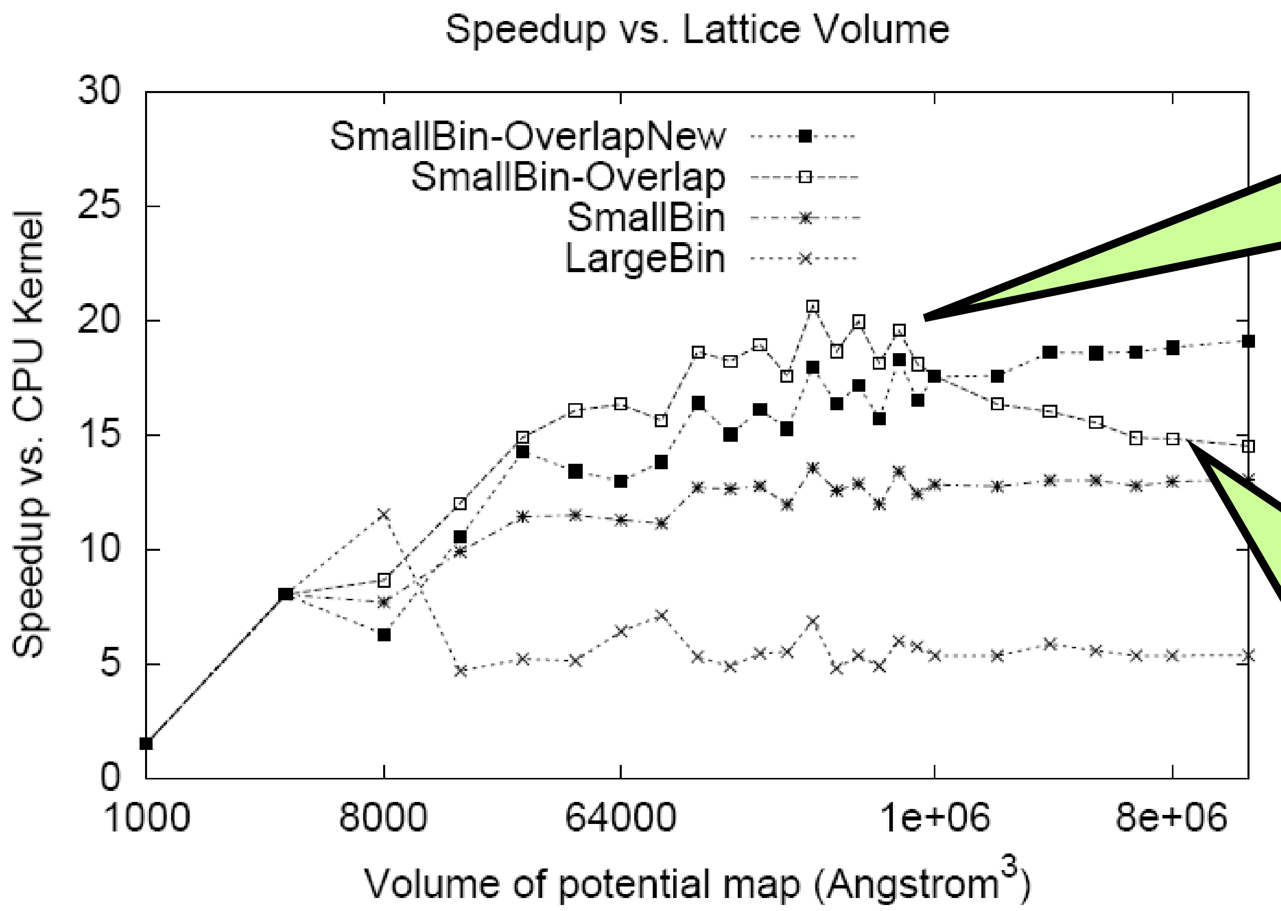
Constant memory

Offsets for bin neighborhood

Look-up table encodes "logic" of spatial geometry


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

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.

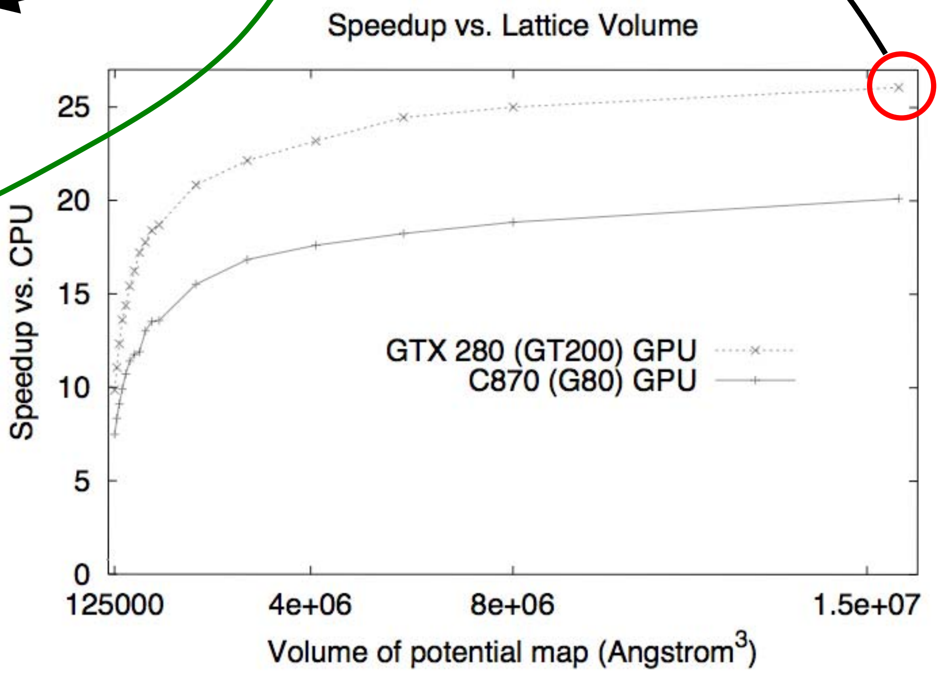
Multilevel Summation on the GPU

GPU computes **short-range cutoff** and **lattice cutoff** parts:

Factor of 26x faster

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

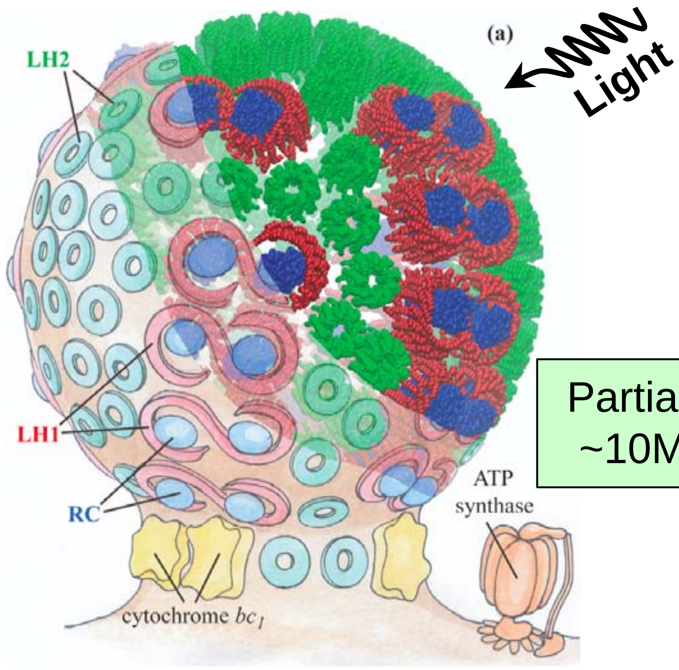


Multilevel summation of electrostatic potentials using graphics processing units.

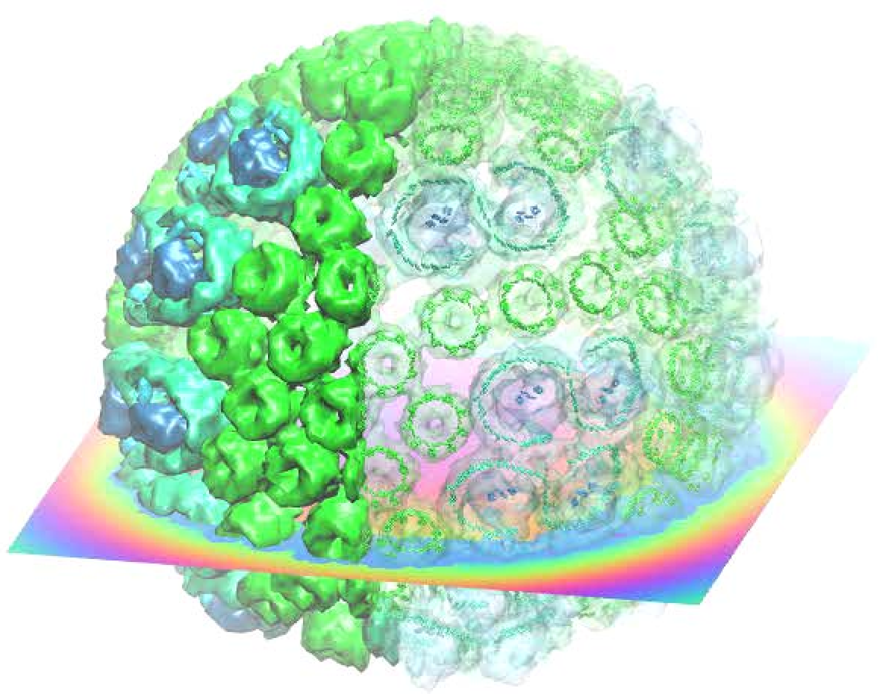
D. Hardy, J. Stone, K. Schulten. *J. Parallel Computing*, 35:164-177, 2009.

Photobiology of Vision and Photosynthesis

Investigations of the chromatophore, a photosynthetic organelle



Partial model: ~10M atoms



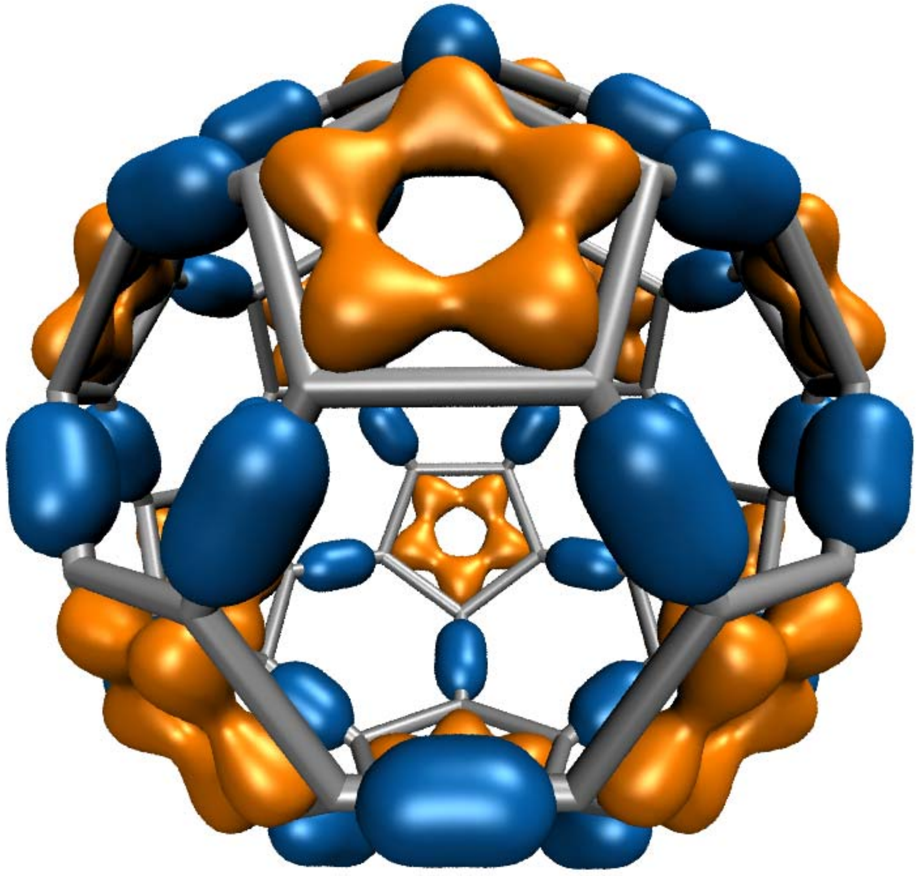
Electrostatics needed to build full structural model, place ions, study macroscopic properties

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

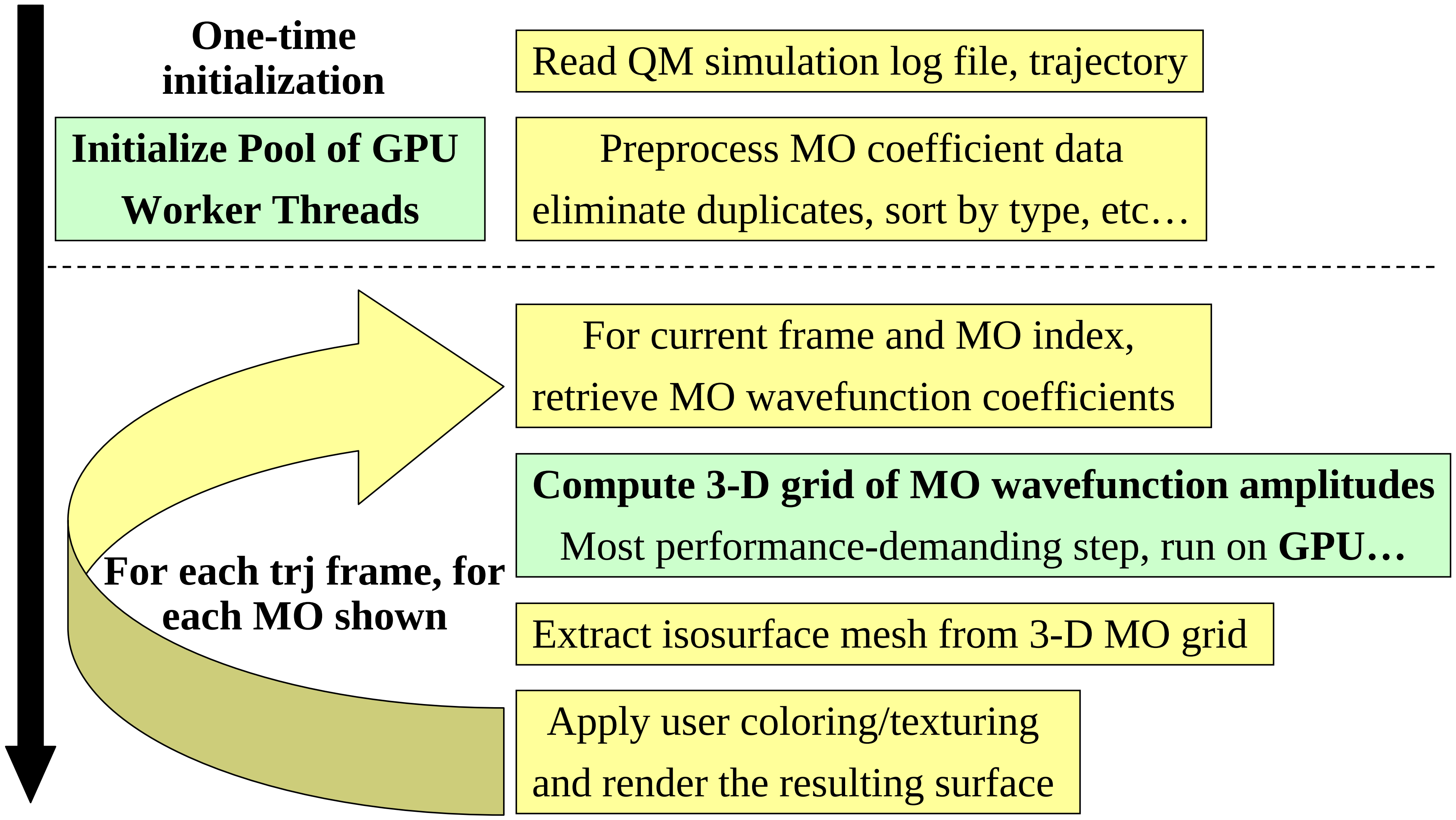
Computing Molecular Orbitals

- Visualization of MOs aids in understanding the chemistry of molecular system
- MO spatial distribution is correlated with electron probability density
- Calculation of high resolution MO grids can require tens to hundreds of seconds on CPUs
- >100x speedup allows interactive animation of MOs @ 10 FPS



C_{60}

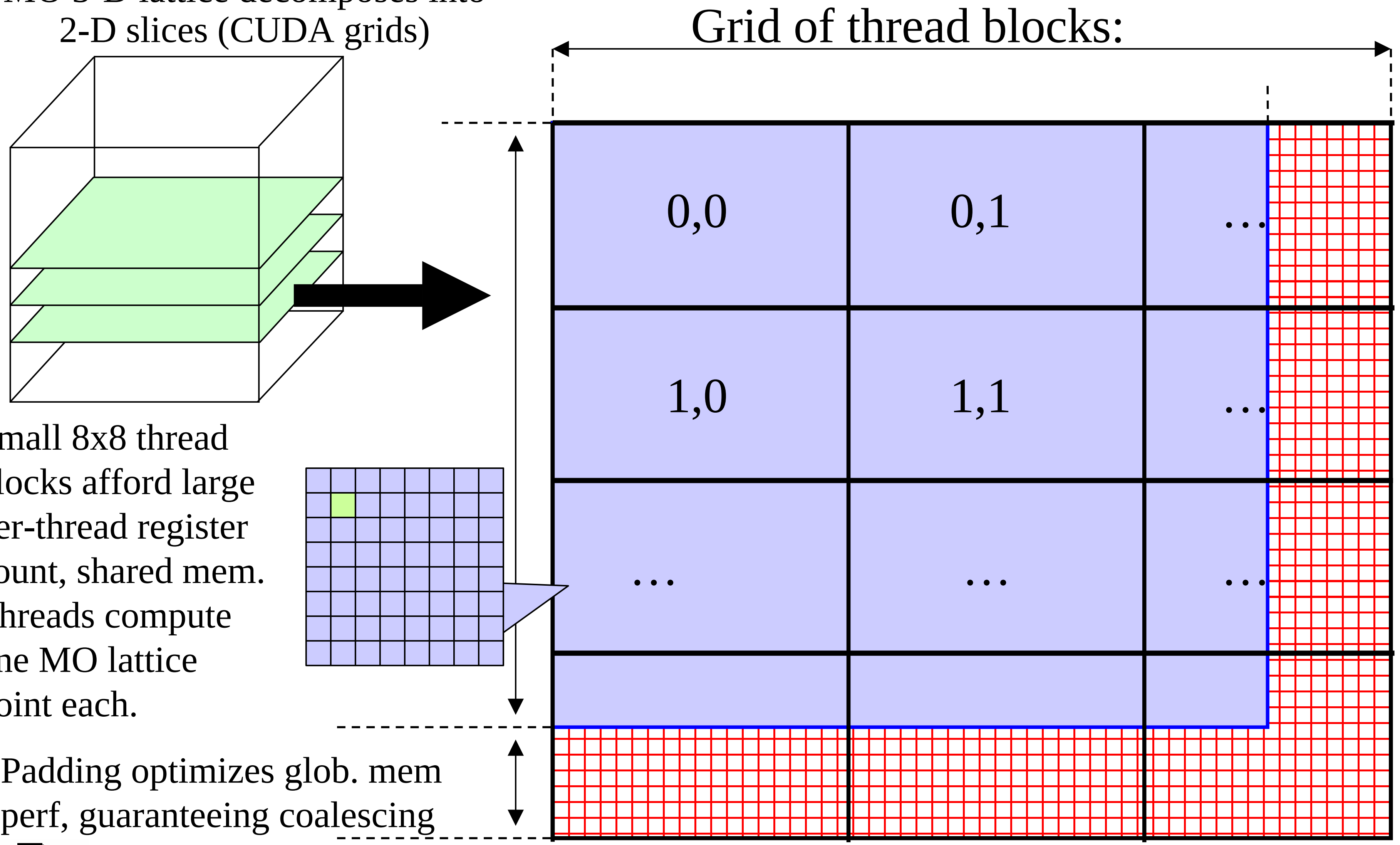
Molecular Orbital Computation and Display Process


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 current frame and MO index, retrieve MO wavefunction coefficients
For each trj frame, for each MO shown
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()

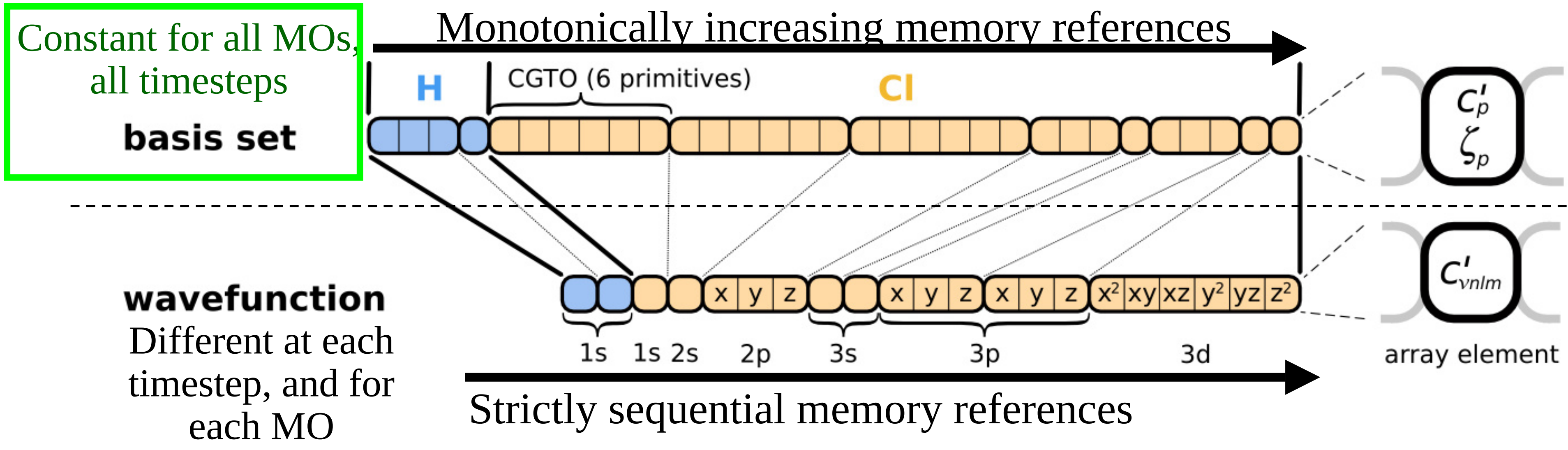
Loop over angular momenta

(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

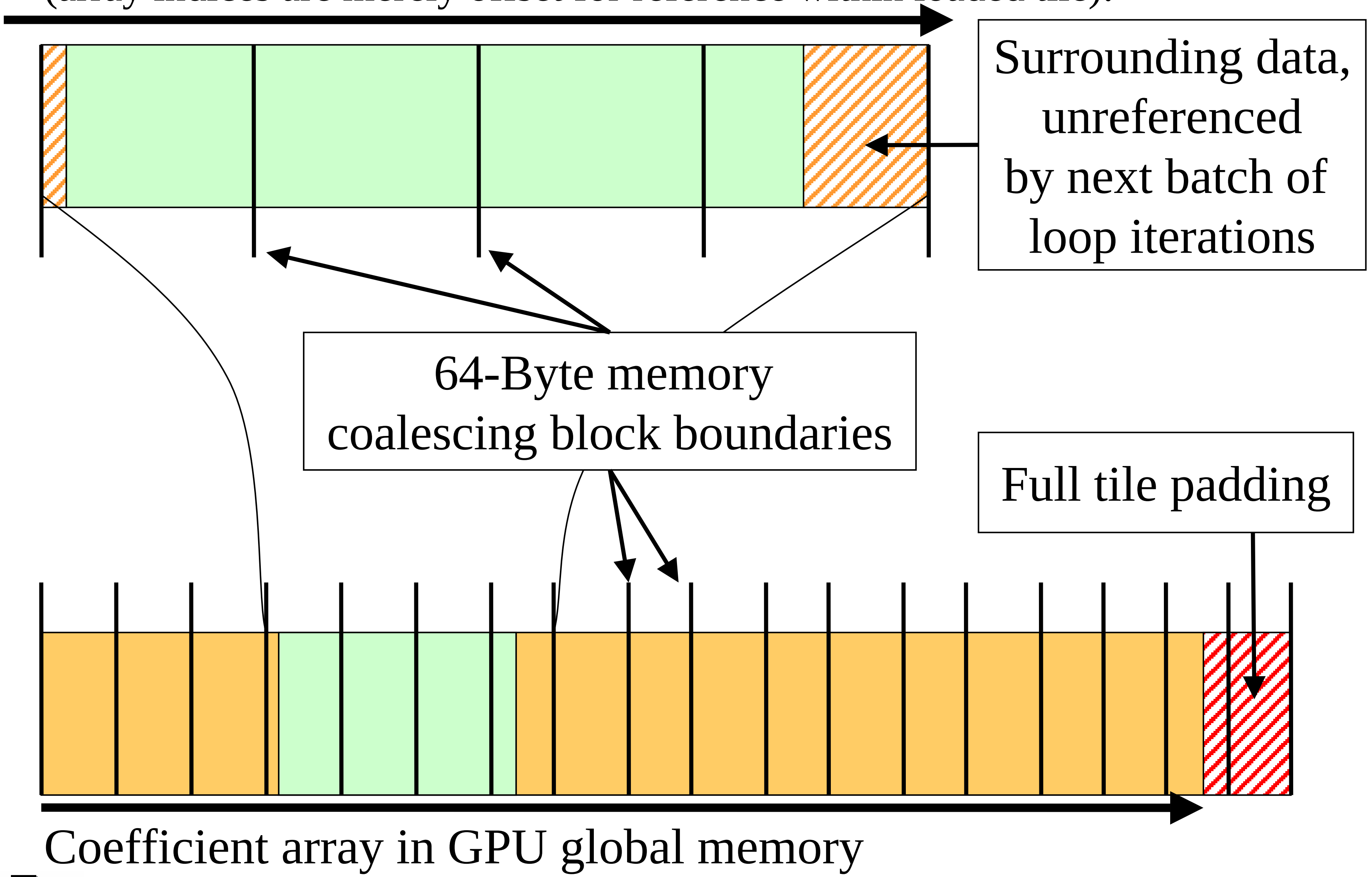

Constant for all MOs, all timesteps
basis set
Monotonically increasing memory references
H
CGTO (6 primitives)
Cl
C'_p
ζ_p
wavefunction
Different at each timestep, and for each MO
x y z x y z x y z
x^2 xy xz y^2 yz z^2
1s 1s 2s 2p 3s 3p 3d
$C'_{\nu nlm}$
array element
Strictly sequential memory references

- Loop iterations always access same or consecutive array elements for all threads in a thread block:
  - Yields good constant memory 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

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)**

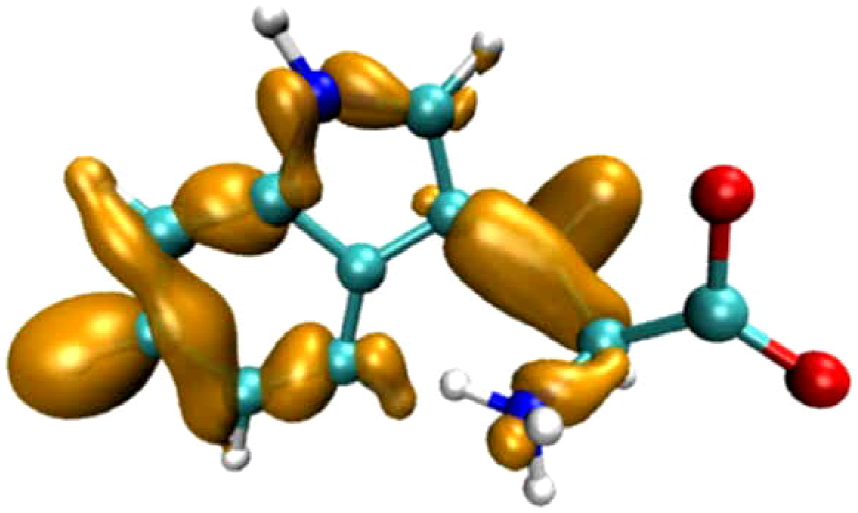
Kernel	**Cores GPUs**	Speedup vs. Molekel on 1 CPU core					
Molekel	1*	1.0	1.0	1.0	1.0	1.0	1.0
MacMolPlt	4	2.4	2.6	2.1	2.4	4.3	4.5
VMD GCC-cephes	4	3.2	4.0	3.0	3.5	4.3	6.5
VMD ICC-SSE-cephes	4	16.8	17.2	13.9	12.6	17.3	21.5
VMD ICC-SSE-approx**	4	59.3	53.4	50.4	49.2	54.8	69.8
VMD CUDA-const-cache	1	552.3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



tryptophane

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…

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

Future Work

- Near term work on GPU acceleration:
  - Radial distribution functions, histogramming
  - Secondary structure rendering
  - Isosurface extraction, volumetric data processing
  - Principle component analysis
- Replace CPU SSE code with OpenCL
- Port some of the existing CUDA GPU kernels to OpenCL where appropriate

Acknowledgements

- Additional Information and References:
  - http://www.ks.uiuc.edu/Research/gpu/
- Questions, source code requests:
  - John Stone: johns@ks.uiuc.edu
- Acknowledgements:

- J. Phillips, D. Hardy, J. Saam,
  UIUC Theoretical and Computational Biophysics Group,
  NIH Resource for Macromolecular Modeling and Bioinformatics
- Prof. Wen-mei Hwu, Christopher Rodrigues, UIUC IMPACT Group
- CUDA team at NVIDIA
- UIUC NVIDIA CUDA Center of Excellence
- NIH support: P41-RR05969

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

- Probing Biomolecular Machines with Graphics Processors. J. Phillips, J. Stone. *Communications of the ACM*, 52(10):34-41, 2009.
- 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)*, IEEE Cluster 2009. In press.
- 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.
- Multilevel summation of electrostatic potentials using graphics processing units. D. Hardy, J. Stone, K. Schulten. *J. Parallel Computing*, 35:164-177, 2009.

Publications (cont)

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.