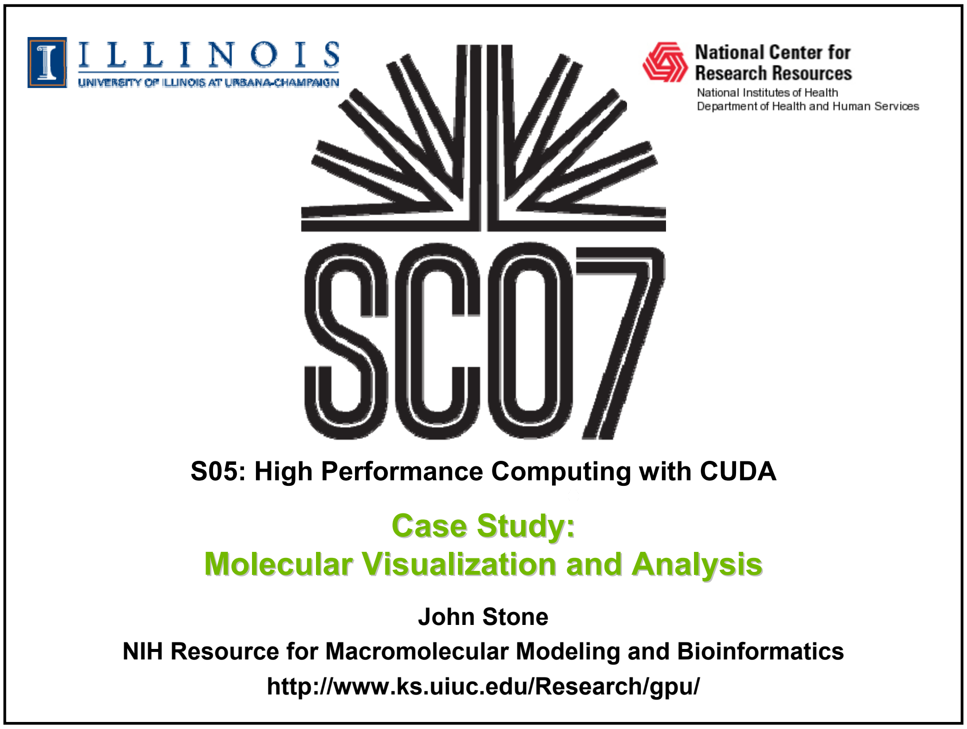


ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

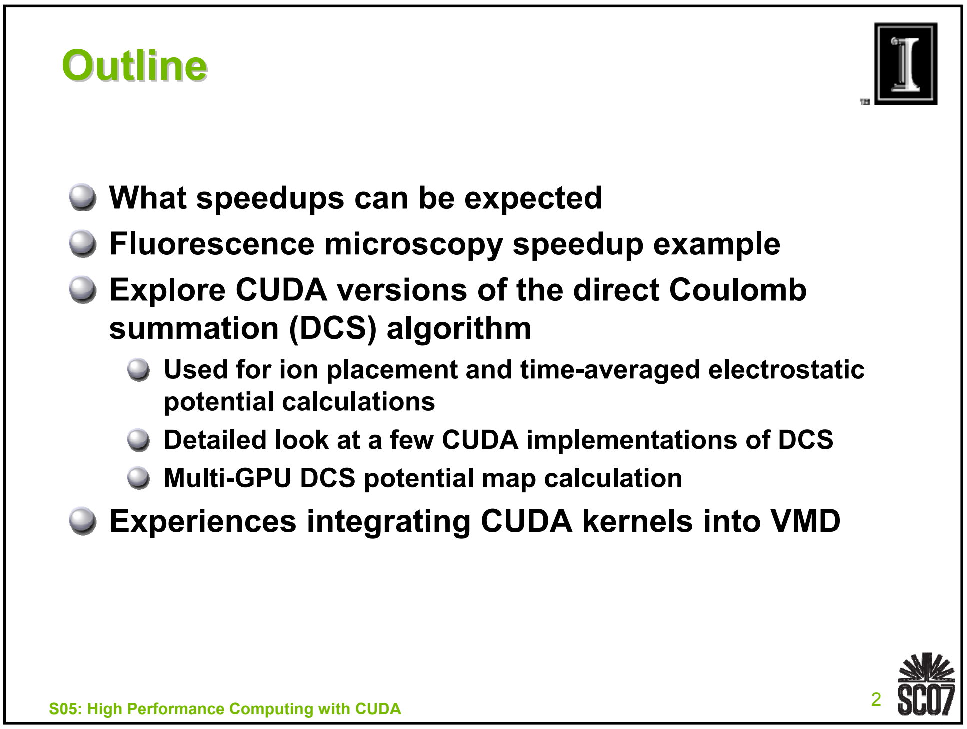
National Center for Research Resources
National Institutes of Health
Department of Health and Human Services

SC07

S05: High Performance Computing with CUDA

Case Study: Molecular Visualization and Analysis

John Stone
NIH Resource for Macromolecular Modeling and Bioinformatics
http://www.ks.uiuc.edu/Research/gpu/



Outline

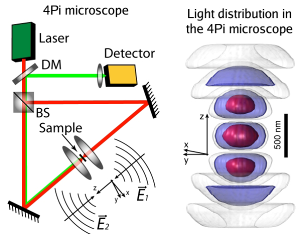
- What speedups can be expected
- Fluorescence microscopy speedup example
- Explore CUDA versions of the direct Coulomb summation (DCS) algorithm
  - Used for ion placement and time-averaged electrostatic potential calculations
  - Detailed look at a few CUDA implementations of DCS
  - Multi-GPU DCS potential map calculation
- Experiences integrating CUDA kernels into VMD

What Speedups Can GPUs Achieve?

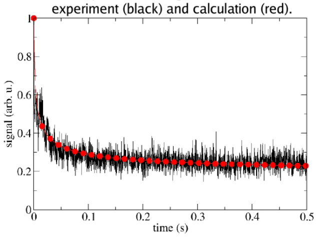
- Speedups of 8x to 30x are quite common
- Best speedups (100x!) are attained on codes that are skewed towards floating point arithmetic, esp. CPU-unfriendly operations that prevent effective use of SSE, vectorization, etc
- Amdahl’s Law can prevent legacy codes from achieving peak speedups with only shallow GPU acceleration efforts

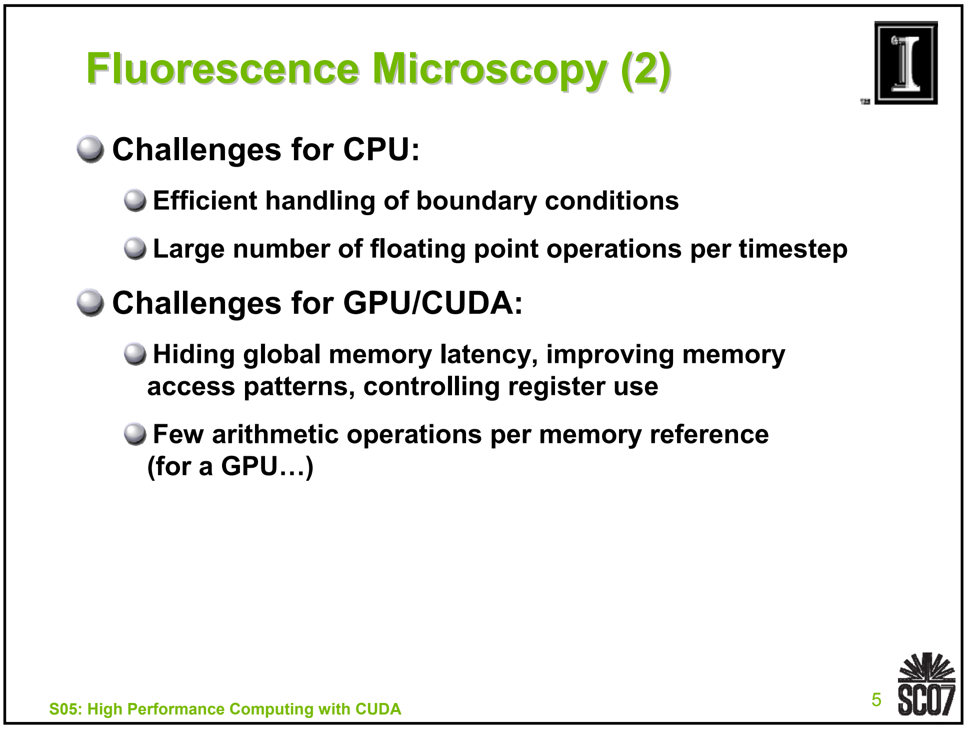
Fluorescence Microscopy

- 2-D reaction-diffusion simulation used to predict results of fluorescence microphotolysis experiments
- Simulate 1-10 second microscopy experiments, 0.1ms integration timesteps
- Goal: <= 1 min per simulation on commodity PC hardware
- Project home page:
  http://www.ks.uiuc.edu/Research/microscope/



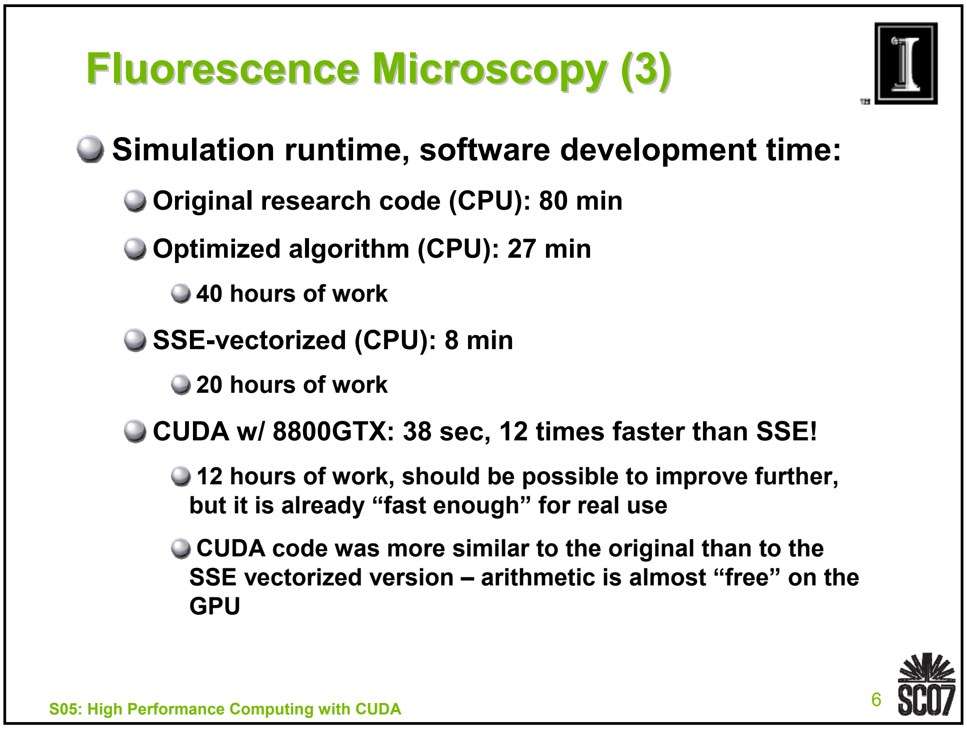
Fluorescence from the sample changes due to photobleaching and diffusion: experiment (black) and calculation (red).





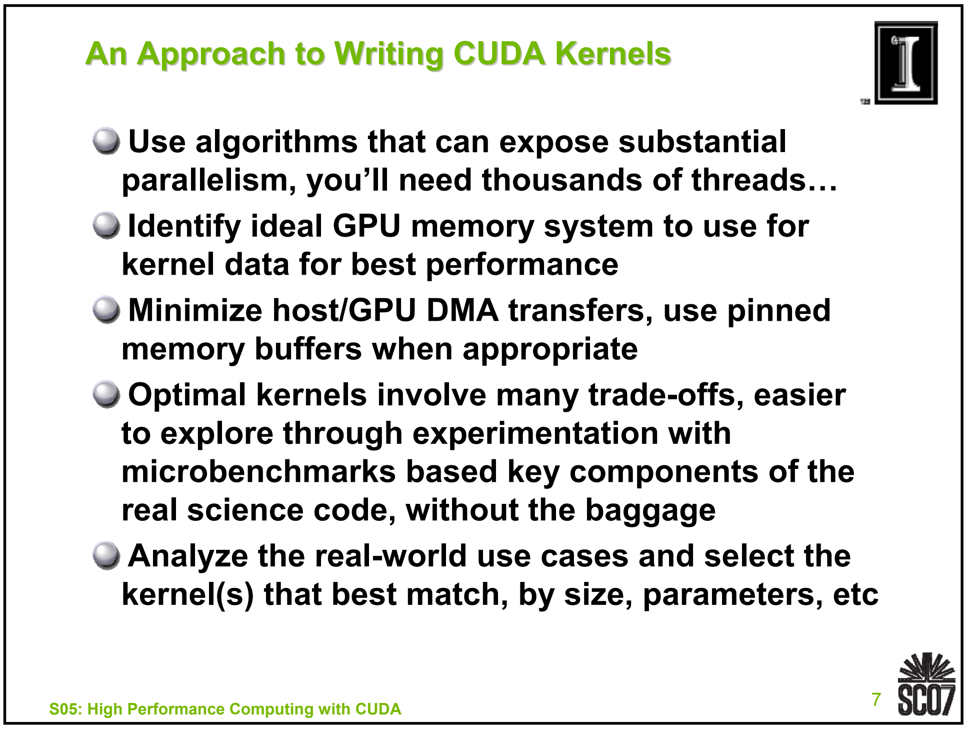
Fluorescence Microscopy (2)

- Challenges for CPU:
  - Efficient handling of boundary conditions
  - Large number of floating point operations per timestep
- Challenges for GPU/CUDA:
  - Hiding global memory latency, improving memory access patterns, controlling register use
  - Few arithmetic operations per memory reference (for a GPU…)



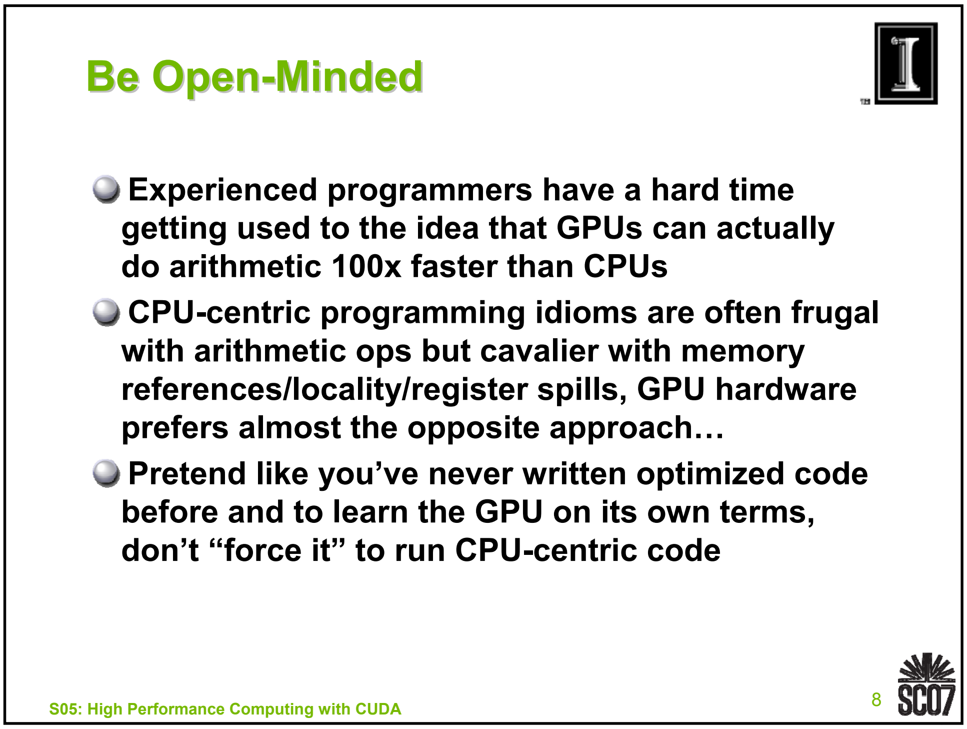
Fluorescence Microscopy (3)

- Simulation runtime, software development time:
  - Original research code (CPU): 80 min
  - Optimized algorithm (CPU): 27 min
    - 40 hours of work
  - SSE-vectorized (CPU): 8 min
    - 20 hours of work
  - CUDA w/ 8800GTX: 38 sec, 12 times faster than SSE!
    - 12 hours of work, should be possible to improve further, but it is already “fast enough” for real use
    - CUDA code was more similar to the original than to the SSE vectorized version – arithmetic is almost “free” on the GPU



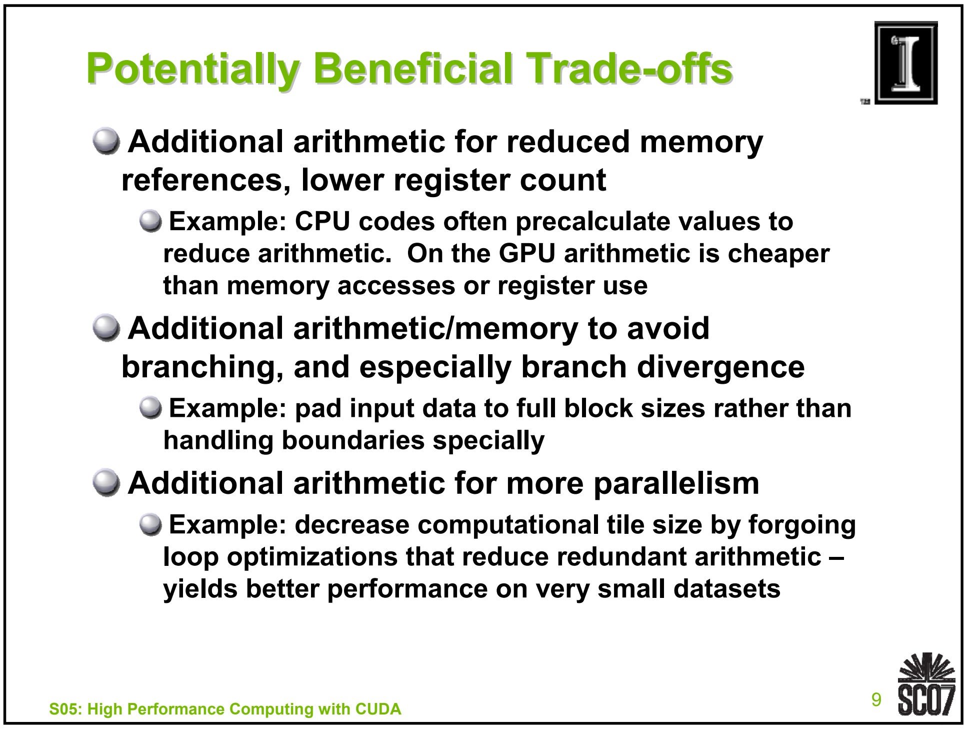
An Approach to Writing CUDA Kernels

- Use algorithms that can expose substantial parallelism, you'll need thousands of threads…
- Identify ideal GPU memory system to use for kernel data for best performance
- Minimize host/GPU DMA transfers, use pinned memory buffers when appropriate
- Optimal kernels involve many trade-offs, easier to explore through experimentation with microbenchmarks based key components of the real science code, without the baggage
- Analyze the real-world use cases and select the kernel(s) that best match, by size, parameters, etc



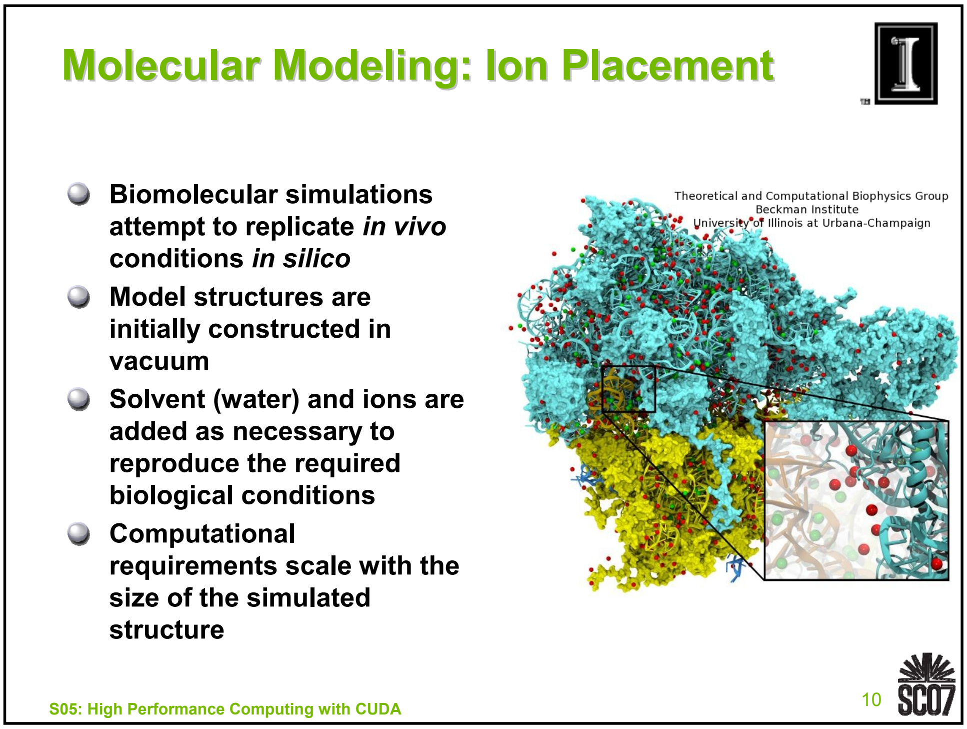
Be Open-Minded

- Experienced programmers have a hard time getting used to the idea that GPUs can actually do arithmetic 100x faster than CPUs
- CPU-centric programming idioms are often frugal with arithmetic ops but cavalier with memory references/locality/register spills, GPU hardware prefers almost the opposite approach…
- Pretend like you've never written optimized code before and to learn the GPU on its own terms, don't "force it" to run CPU-centric code



Potentially Beneficial Trade-offs

- Additional arithmetic for reduced memory references, lower register count
  - Example: CPU codes often precalculate values to reduce arithmetic. On the GPU arithmetic is cheaper than memory accesses or register use
- Additional arithmetic/memory to avoid branching, and especially branch divergence
  - Example: pad input data to full block sizes rather than handling boundaries specially
- Additional arithmetic for more parallelism
  - Example: decrease computational tile size by forgoing loop optimizations that reduce redundant arithmetic – yields better performance on very small datasets



Molecular Modeling: Ion Placement

- Biomolecular simulations attempt to replicate *in vivo* conditions *in silico*
- Model structures are initially constructed in vacuum
- Solvent (water) and ions are added as necessary to reproduce the required biological conditions
- Computational requirements scale with the size of the simulated structure

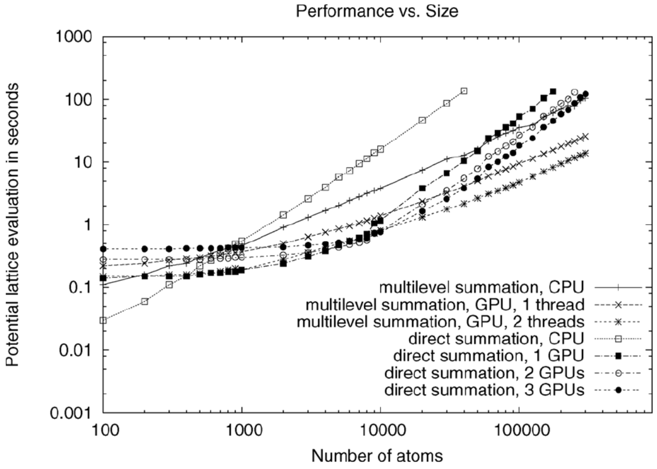
Theoretical and Computational Biophysics Group
Beckman Institute
University of Illinois at Urbana-Champaign

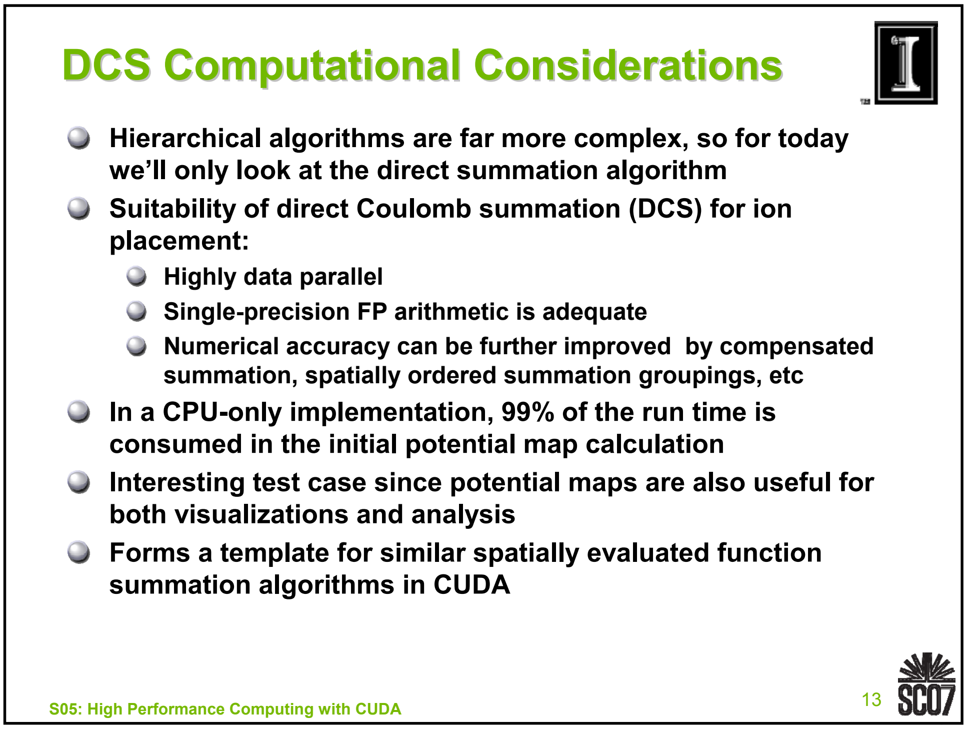
Overview of Ion Placement Process

- Calculate initial electrostatic potential map around the simulated structure via direct Coulomb summation algorithm (DCS):
  - For each lattice point, sum potential contributions for all atoms in the simulated structure:

    potential += charge[i] / (distance to atom[i])
- Ions are then placed one at a time:
  - Find the voxel containing the minimum potential value
  - Add a new ion atom at location of minimum potential
  - Add the potential contribution of the newly placed ion to the entire map
  - Repeat until the required number of ions have been added

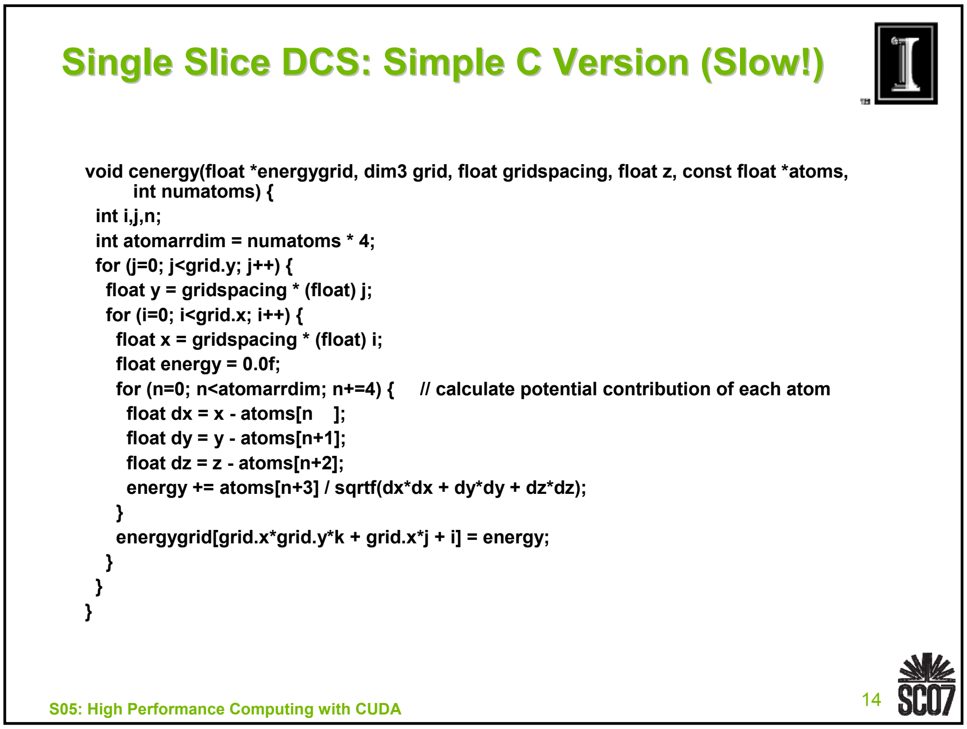
Runtime of Coulomb Summation Algorithms on CPU and GPU (parameterize kernels for data size)





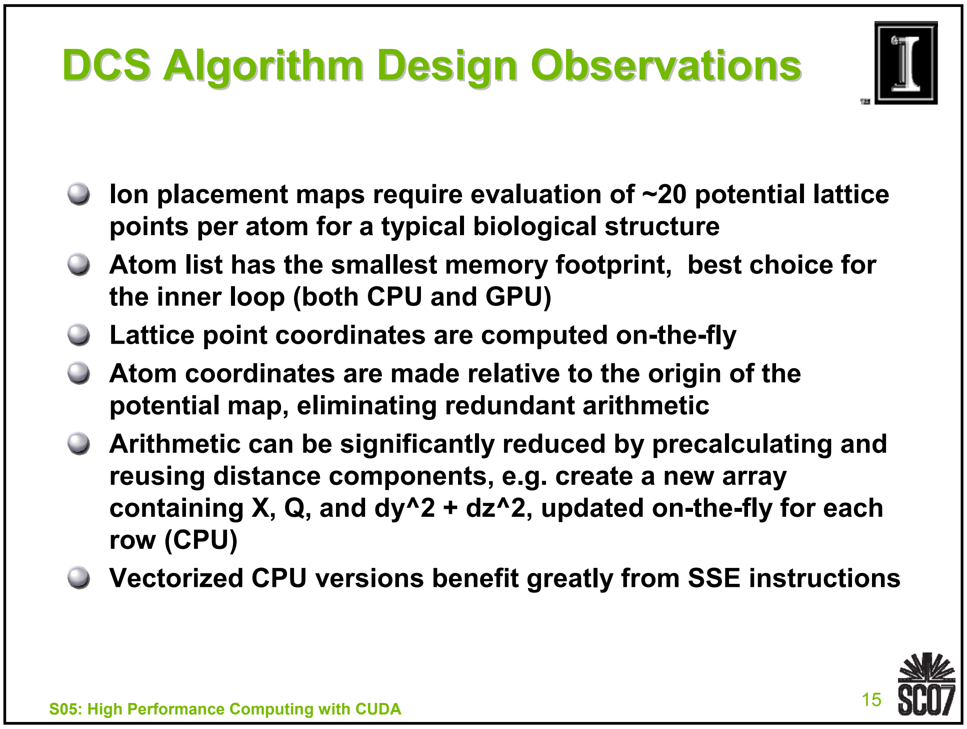
DCS Computational Considerations

- Hierarchical algorithms are far more complex, so for today we'll only look at the direct summation algorithm
- Suitability of direct Coulomb summation (DCS) for ion placement:
  - Highly data parallel
  - Single-precision FP arithmetic is adequate
  - Numerical accuracy can be further improved by compensated summation, spatially ordered summation groupings, etc
- In a CPU-only implementation, 99% of the run time is consumed in the initial potential map calculation
- Interesting test case since potential maps are also useful for both visualizations and analysis
- Forms a template for similar spatially evaluated function summation algorithms in CUDA



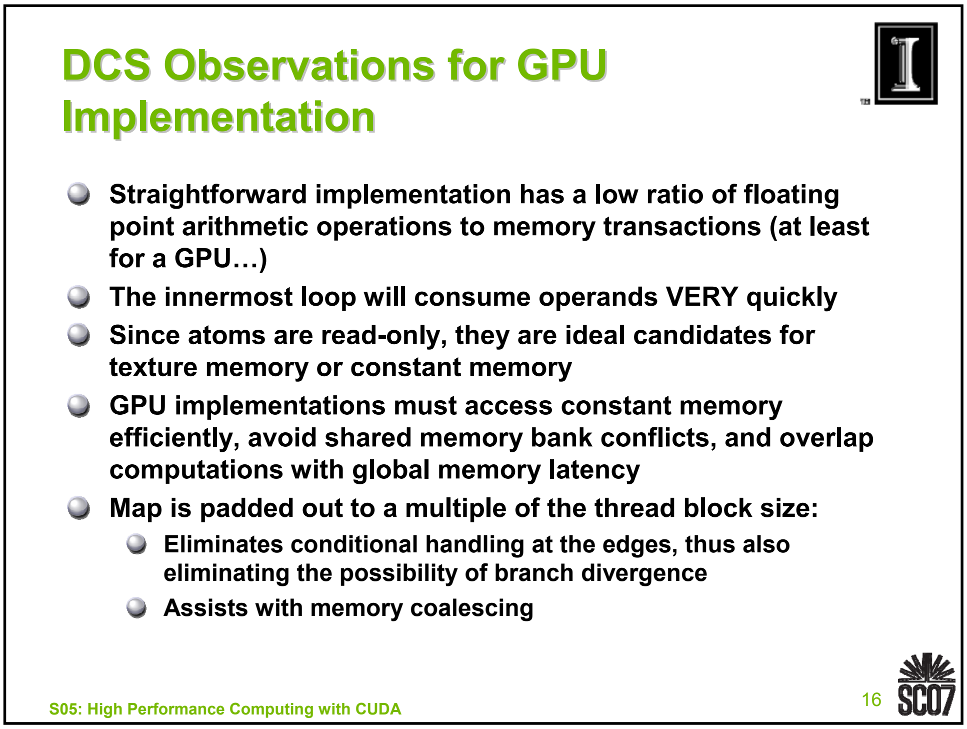
Single Slice DCS: Simple C Version (Slow!)

```
void cenergy(float *energygrid, dim3 grid, float gridspacing, float z, const float *atoms,
        int numatoms) {
  int i,j,n;
  int atomarrdim = numatoms * 4;
  for (j=0; j<grid.y; j++) {
    float y = gridspacing * (float) j;
    for (i=0; i<grid.x; i++) {
      float x = gridspacing * (float) i;
      float energy = 0.0f;
      for (n=0; n<atomarrdim; n+=4) {     // calculate potential contribution of each atom
        float dx = x - atoms[n    ];
        float dy = y - atoms[n+1];
        float dz = z - atoms[n+2];
        energy += atoms[n+3] / sqrtf(dx*dx + dy*dy + dz*dz);
      }
      energygrid[grid.x*grid.y*k + grid.x*j + i] = energy;
    }
  }
}
```



DCS Algorithm Design Observations

- Ion placement maps require evaluation of ~20 potential lattice points per atom for a typical biological structure
- Atom list has the smallest memory footprint, best choice for the inner loop (both CPU and GPU)
- Lattice point coordinates are computed on-the-fly
- Atom coordinates are made relative to the origin of the potential map, eliminating redundant arithmetic
- Arithmetic can be significantly reduced by precalculating and reusing distance components, e.g. create a new array containing X, Q, and dy^2 + dz^2, updated on-the-fly for each row (CPU)
- Vectorized CPU versions benefit greatly from SSE instructions



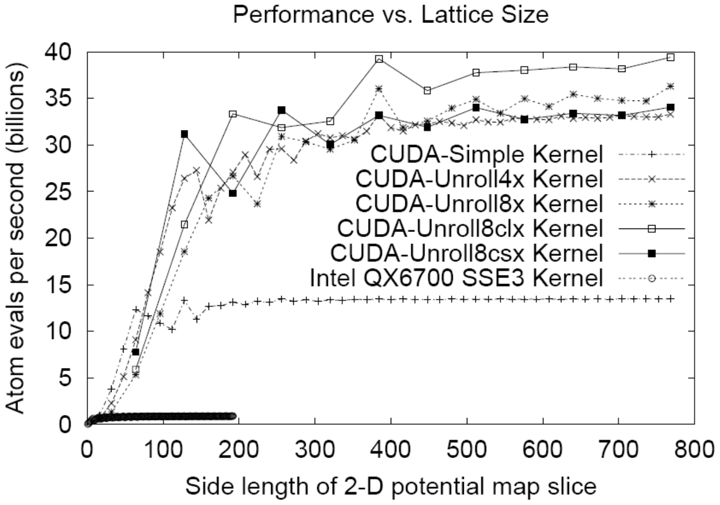
DCS Observations for GPU Implementation

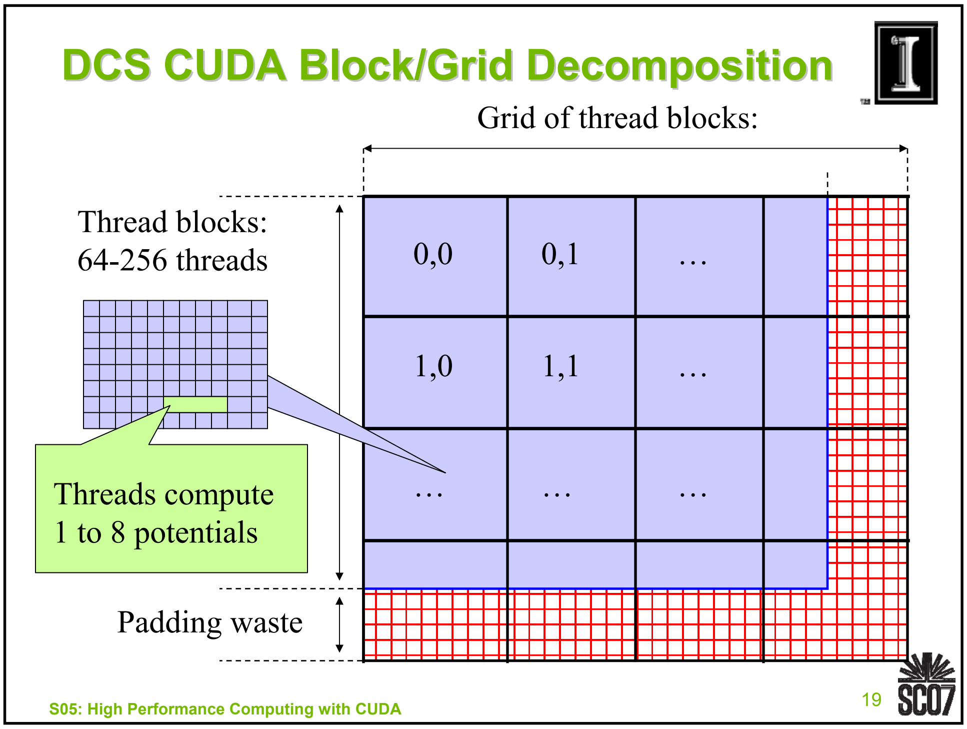
- Straightforward implementation has a low ratio of floating point arithmetic operations to memory transactions (at least for a GPU…)
- The innermost loop will consume operands VERY quickly
- Since atoms are read-only, they are ideal candidates for texture memory or constant memory
- GPU implementations must access constant memory efficiently, avoid shared memory bank conflicts, and overlap computations with global memory latency
- Map is padded out to a multiple of the thread block size:
  - Eliminates conditional handling at the edges, thus also eliminating the possibility of branch divergence
  - Assists with memory coalescing

CUDA DCS Implementation Overview

- Allocate and initialize potential map memory on host CPU
- Allocate potential map slice buffer on GPU
- Preprocess atom coordinates and charges
- Loop over slices:
  - Copy slice from host to GPU
  - Loop over groups of atoms: (if necessary)
    - Copy atom data to GPU
    - Run CUDA Kernel on atoms and slice resident on GPU
  - Copy slice from GPU to host
- Free resources

Comparison of Direct Coulomb Summation Kernels on CPU and GPU





DCS CUDA Block/Grid Decomposition

Grid of thread blocks:

Thread blocks:
64-256 threads

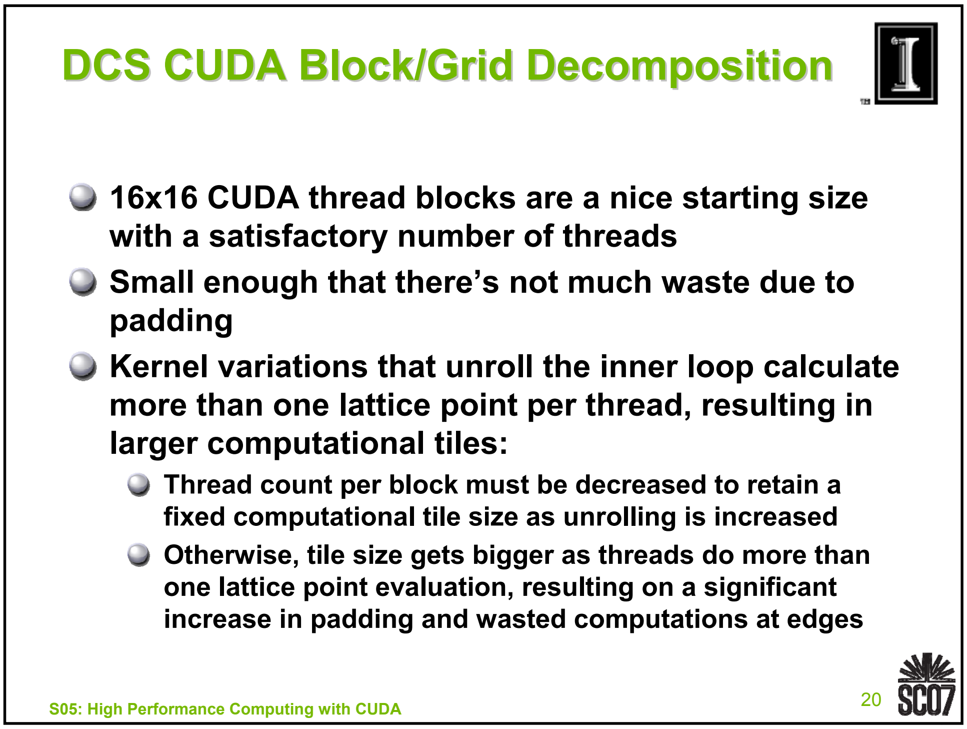
0,0	0,1	…	
1,0	1,1	…	
…	…	…	

Threads compute
1 to 8 potentials

Padding waste

S05: High Performance Computing with CUDA

19

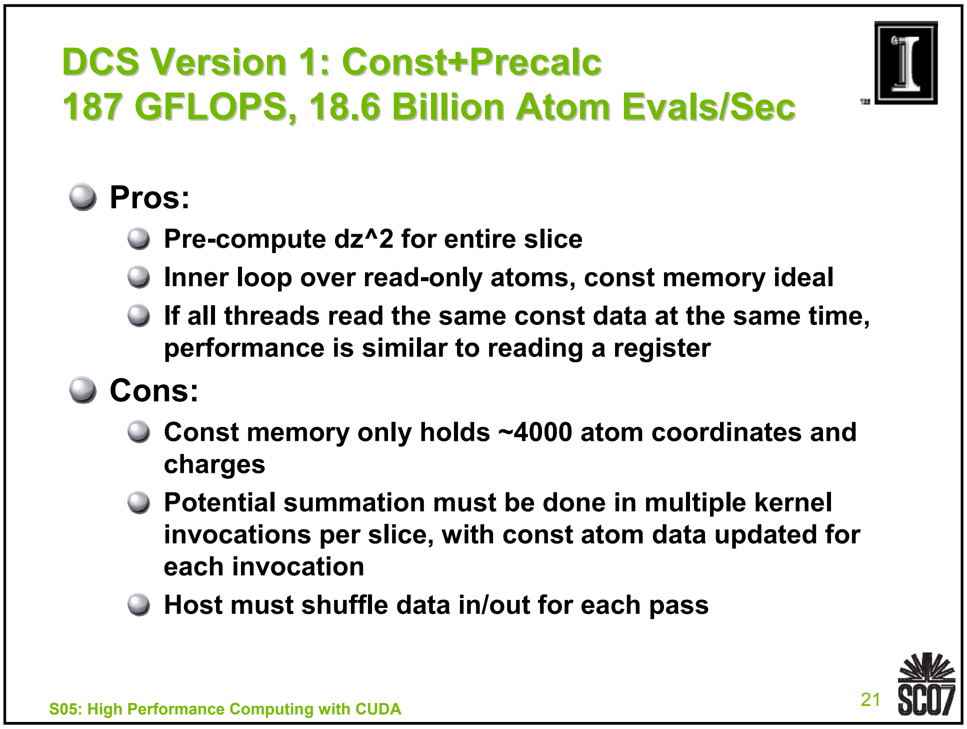


DCS CUDA Block/Grid Decomposition

- 16x16 CUDA thread blocks are a nice starting size with a satisfactory number of threads
- Small enough that there's not much waste due to padding
- Kernel variations that unroll the inner loop calculate more than one lattice point per thread, resulting in larger computational tiles:
  - Thread count per block must be decreased to retain a fixed computational tile size as unrolling is increased
  - Otherwise, tile size gets bigger as threads do more than one lattice point evaluation, resulting on a significant increase in padding and wasted computations at edges

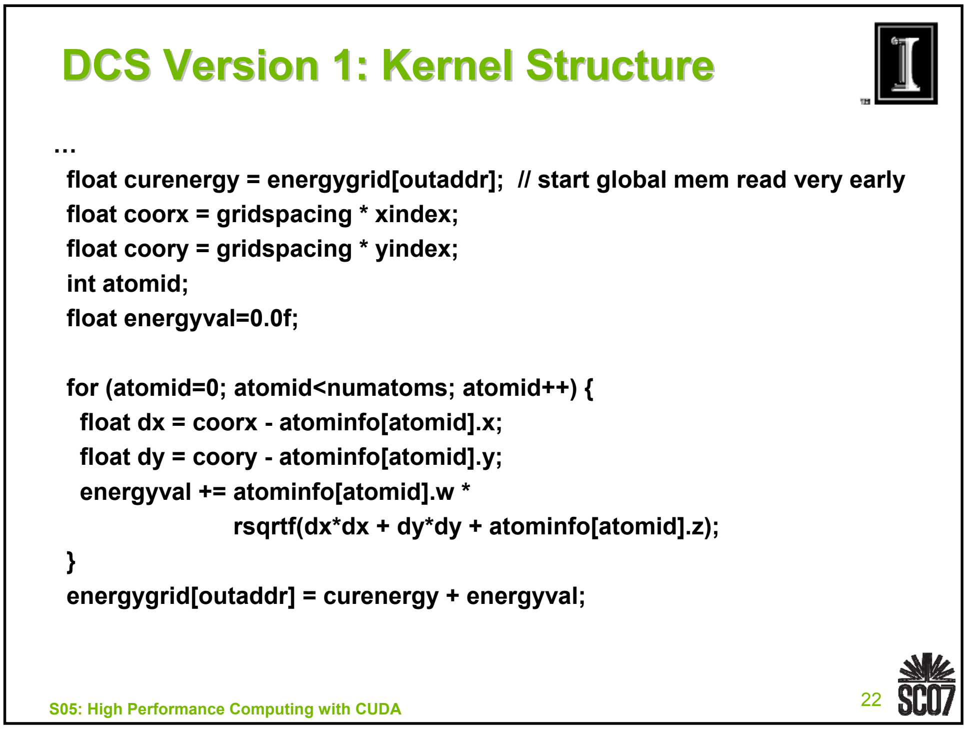
S05: High Performance Computing with CUDA

20



DCS Version 1: Const+Precalc 187 GFLOPS, 18.6 Billion Atom Evals/Sec

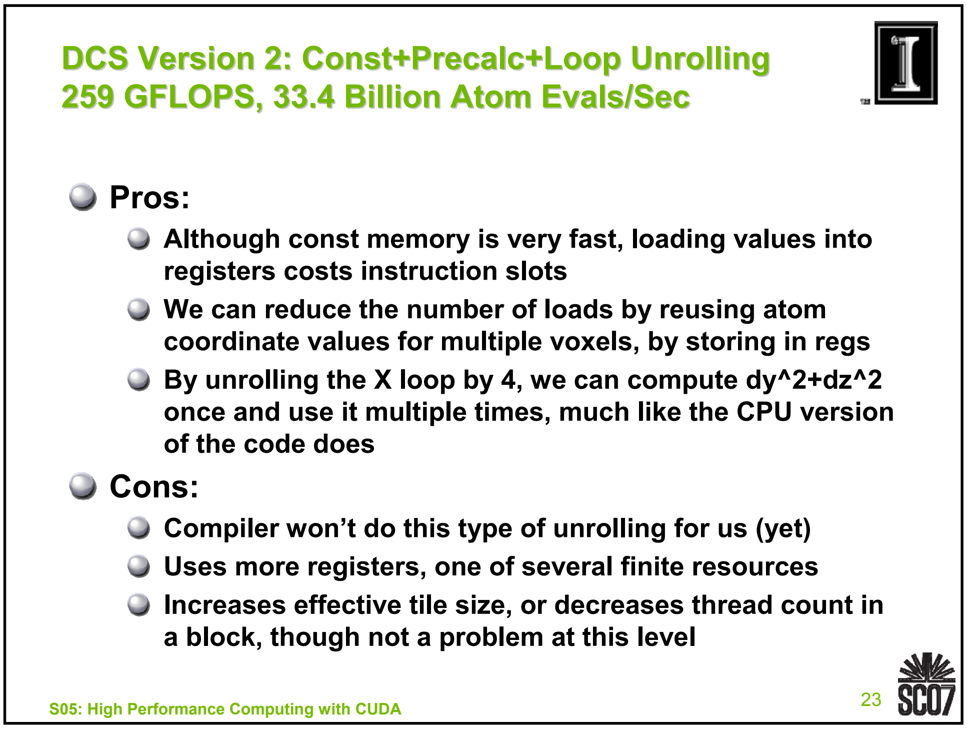
- Pros:
  - Pre-compute dz^2 for entire slice
  - Inner loop over read-only atoms, const memory ideal
  - If all threads read the same const data at the same time, performance is similar to reading a register
- Cons:
  - Const memory only holds ~4000 atom coordinates and charges
  - Potential summation must be done in multiple kernel invocations per slice, with const atom data updated for each invocation
  - Host must shuffle data in/out for each pass



DCS Version 1: Kernel Structure

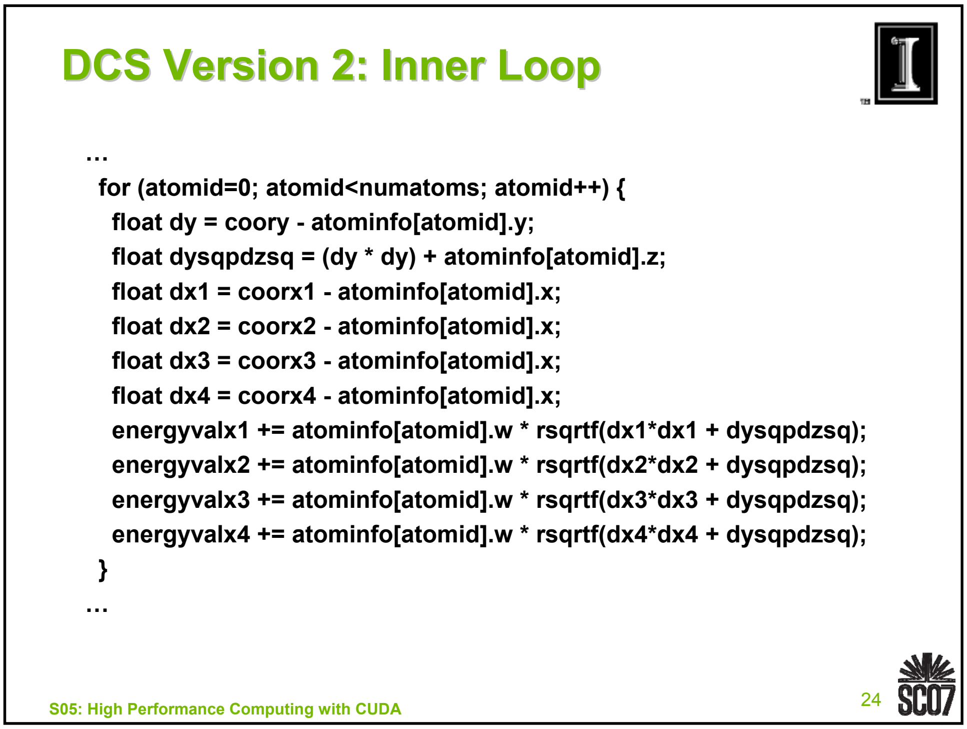
```
…
 float curenergy = energygrid[outaddr];  // start global mem read very early
 float coorx = gridspacing * xindex;
 float coory = gridspacing * yindex;
 int atomid;
 float energyval=0.0f;

 for (atomid=0; atomid<numatoms; atomid++) {
   float dx = coorx - atominfo[atomid].x;
   float dy = coory - atominfo[atomid].y;
   energyval += atominfo[atomid].w *
                  rsqrtf(dx*dx + dy*dy + atominfo[atomid].z);
 }
 energygrid[outaddr] = curenergy + energyval;
```



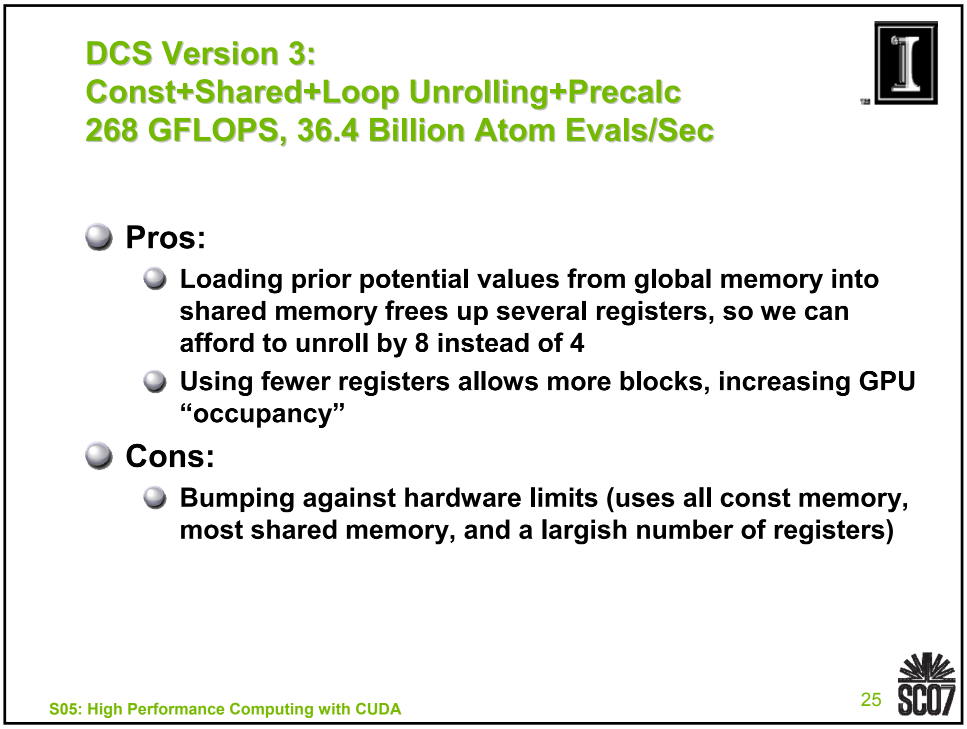
DCS Version 2: Const+Precalc+Loop Unrolling 259 GFLOPS, 33.4 Billion Atom Evals/Sec

- Pros:
  - Although const memory is very fast, loading values into registers costs instruction slots
  - We can reduce the number of loads by reusing atom coordinate values for multiple voxels, by storing in regs
  - By unrolling the X loop by 4, we can compute dy^2+dz^2 once and use it multiple times, much like the CPU version of the code does
- Cons:
  - Compiler won’t do this type of unrolling for us (yet)
  - Uses more registers, one of several finite resources
  - Increases effective tile size, or decreases thread count in a block, though not a problem at this level



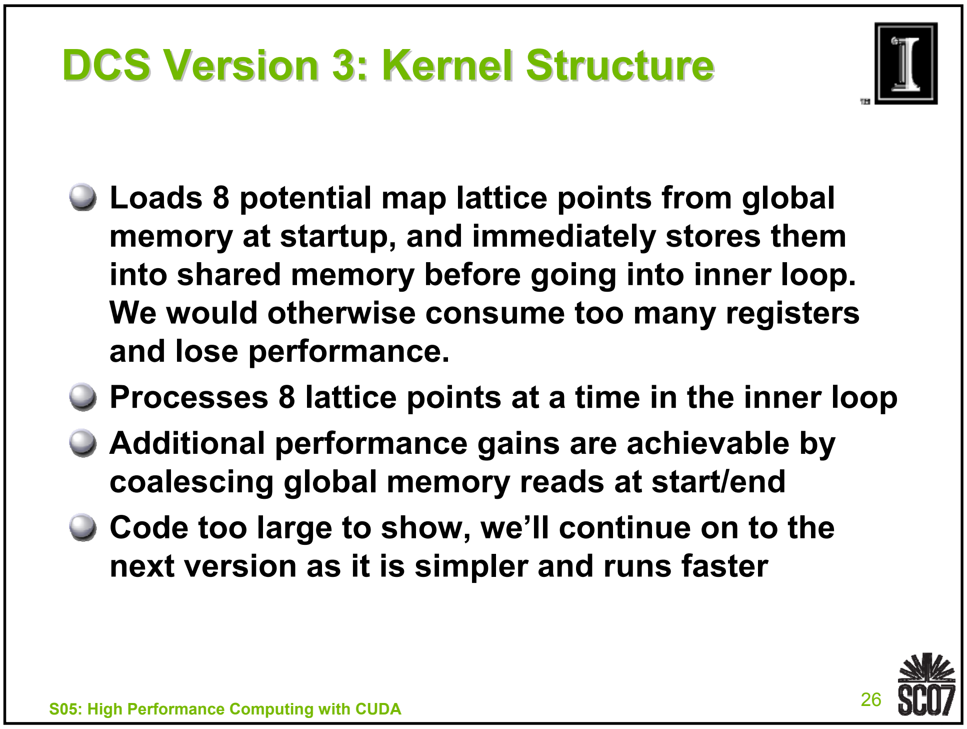
DCS Version 2: Inner Loop

```
…
 for (atomid=0; atomid<numatoms; atomid++) {
  float dy = coory - atominfo[atomid].y;
  float dysqpdzsq = (dy * dy) + atominfo[atomid].z;
  float dx1 = coorx1 - atominfo[atomid].x;
  float dx2 = coorx2 - atominfo[atomid].x;
  float dx3 = coorx3 - atominfo[atomid].x;
  float dx4 = coorx4 - atominfo[atomid].x;
  energyvalx1 += atominfo[atomid].w * rsqrtf(dx1*dx1 + dysqpdzsq);
  energyvalx2 += atominfo[atomid].w * rsqrtf(dx2*dx2 + dysqpdzsq);
  energyvalx3 += atominfo[atomid].w * rsqrtf(dx3*dx3 + dysqpdzsq);
  energyvalx4 += atominfo[atomid].w * rsqrtf(dx4*dx4 + dysqpdzsq);
 }
…
```



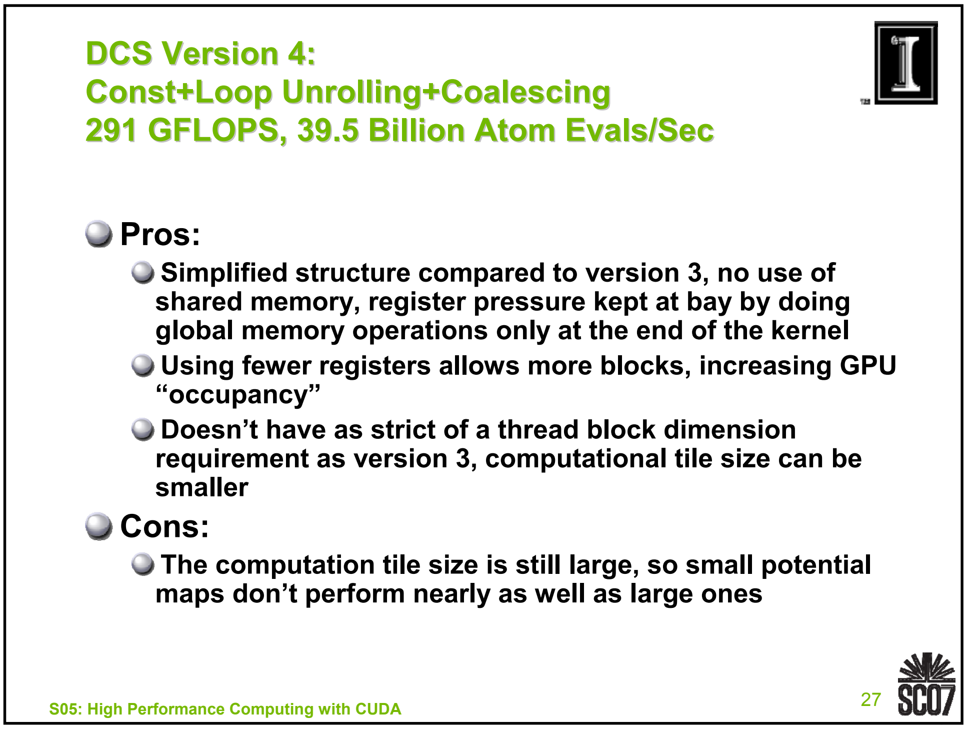
DCS Version 3:
Const+Shared+Loop Unrolling+Precalc
268 GFLOPS, 36.4 Billion Atom Evals/Sec

- Pros:
  - Loading prior potential values from global memory into shared memory frees up several registers, so we can afford to unroll by 8 instead of 4
  - Using fewer registers allows more blocks, increasing GPU "occupancy"
- Cons:
  - Bumping against hardware limits (uses all const memory, most shared memory, and a largish number of registers)



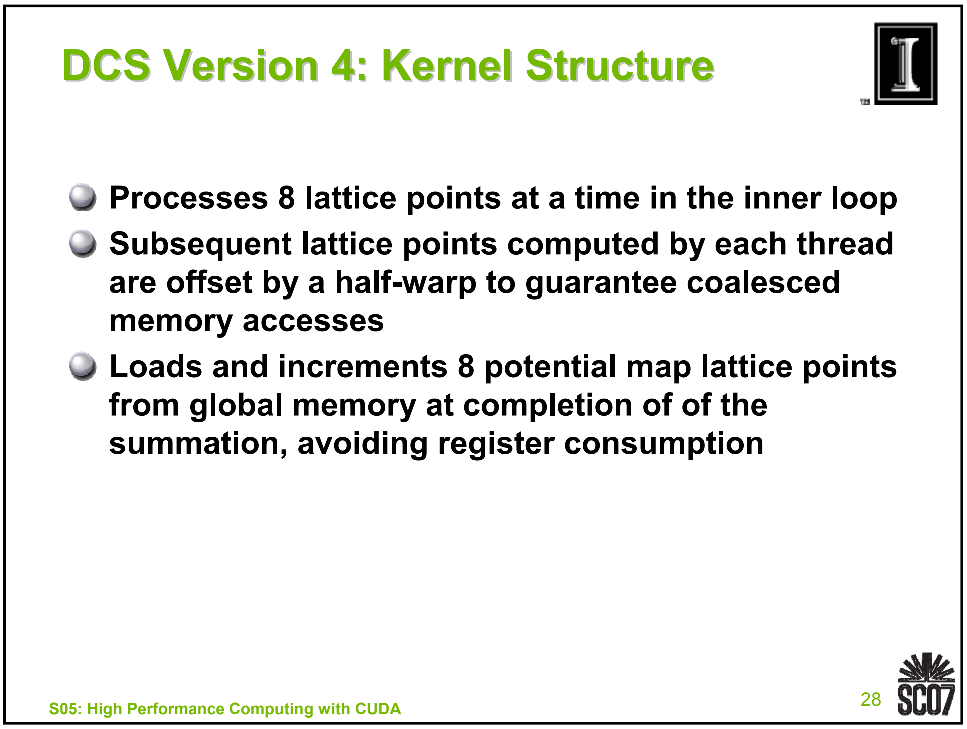
DCS Version 3: Kernel Structure

- Loads 8 potential map lattice points from global memory at startup, and immediately stores them into shared memory before going into inner loop. We would otherwise consume too many registers and lose performance.
- Processes 8 lattice points at a time in the inner loop
- Additional performance gains are achievable by coalescing global memory reads at start/end
- Code too large to show, we'll continue on to the next version as it is simpler and runs faster



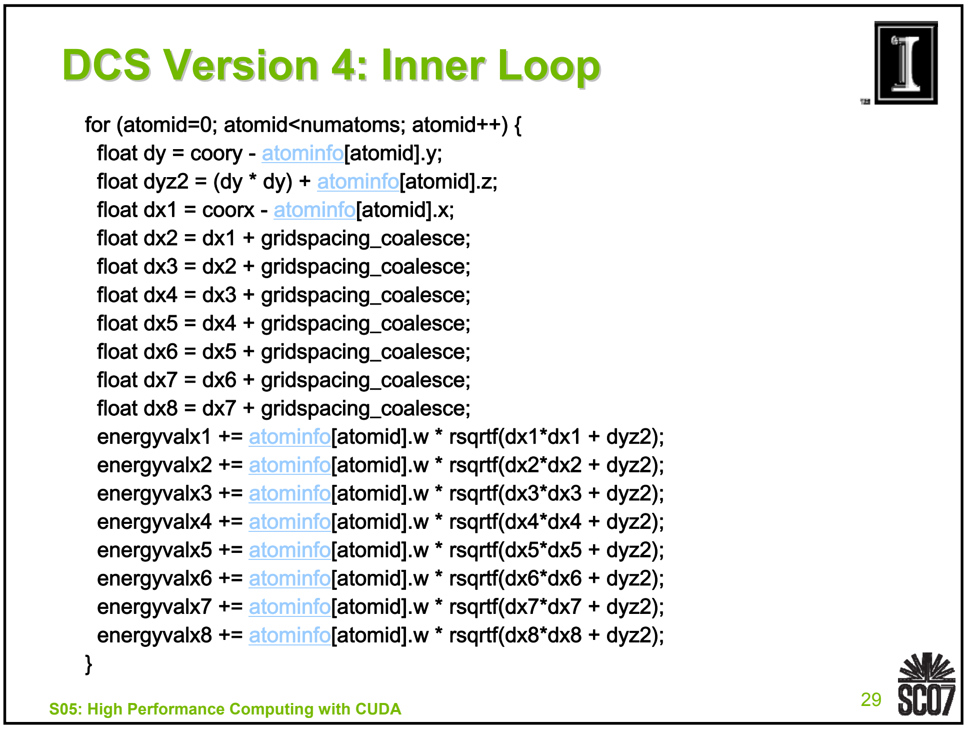
DCS Version 4:
Const+Loop Unrolling+Coalescing
291 GFLOPS, 39.5 Billion Atom Evals/Sec

- Pros:
  - Simplified structure compared to version 3, no use of shared memory, register pressure kept at bay by doing global memory operations only at the end of the kernel
  - Using fewer registers allows more blocks, increasing GPU “occupancy”
  - Doesn’t have as strict of a thread block dimension requirement as version 3, computational tile size can be smaller
- Cons:
  - The computation tile size is still large, so small potential maps don’t perform nearly as well as large ones



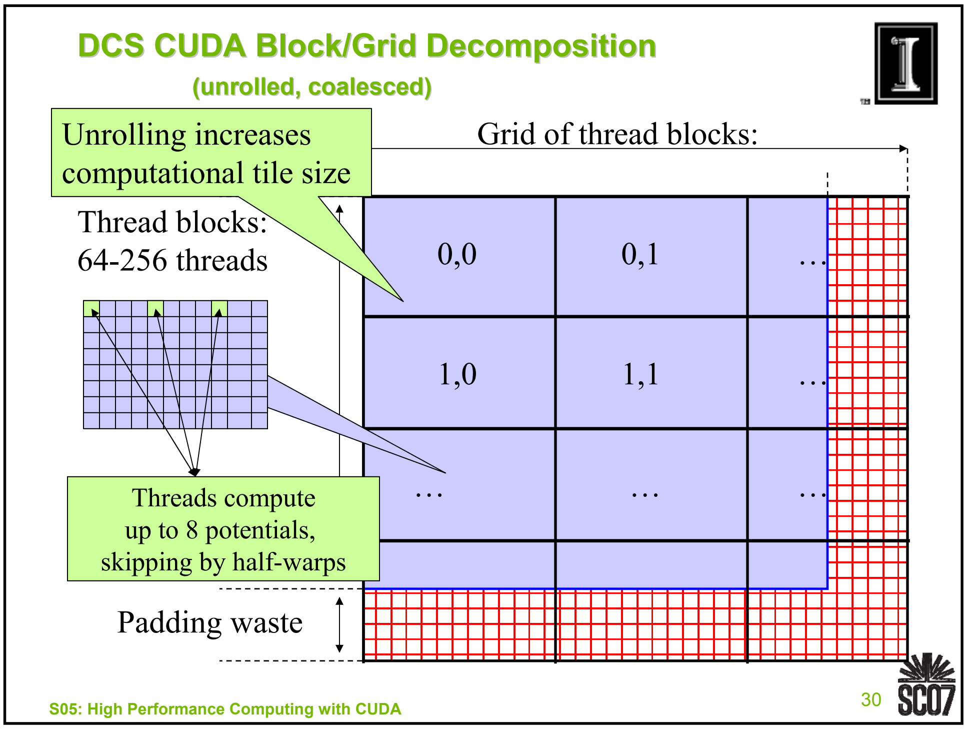
DCS Version 4: Kernel Structure

- Processes 8 lattice points at a time in the inner loop
- Subsequent lattice points computed by each thread are offset by a half-warp to guarantee coalesced memory accesses
- Loads and increments 8 potential map lattice points from global memory at completion of of the summation, avoiding register consumption



DCS Version 4: Inner Loop

```
for (atomid=0; atomid<numatoms; atomid++) {
  float dy = coory - atominfo[atomid].y;
  float dyz2 = (dy * dy) + atominfo[atomid].z;
  float dx1 = coorx - atominfo[atomid].x;
  float dx2 = dx1 + gridspacing_coalesce;
  float dx3 = dx2 + gridspacing_coalesce;
  float dx4 = dx3 + gridspacing_coalesce;
  float dx5 = dx4 + gridspacing_coalesce;
  float dx6 = dx5 + gridspacing_coalesce;
  float dx7 = dx6 + gridspacing_coalesce;
  float dx8 = dx7 + gridspacing_coalesce;
  energyvalx1 += atominfo[atomid].w * rsqrtf(dx1*dx1 + dyz2);
  energyvalx2 += atominfo[atomid].w * rsqrtf(dx2*dx2 + dyz2);
  energyvalx3 += atominfo[atomid].w * rsqrtf(dx3*dx3 + dyz2);
  energyvalx4 += atominfo[atomid].w * rsqrtf(dx4*dx4 + dyz2);
  energyvalx5 += atominfo[atomid].w * rsqrtf(dx5*dx5 + dyz2);
  energyvalx6 += atominfo[atomid].w * rsqrtf(dx6*dx6 + dyz2);
  energyvalx7 += atominfo[atomid].w * rsqrtf(dx7*dx7 + dyz2);
  energyvalx8 += atominfo[atomid].w * rsqrtf(dx8*dx8 + dyz2);
}
```



DCS CUDA Block/Grid Decomposition

(unrolled, coalesced)

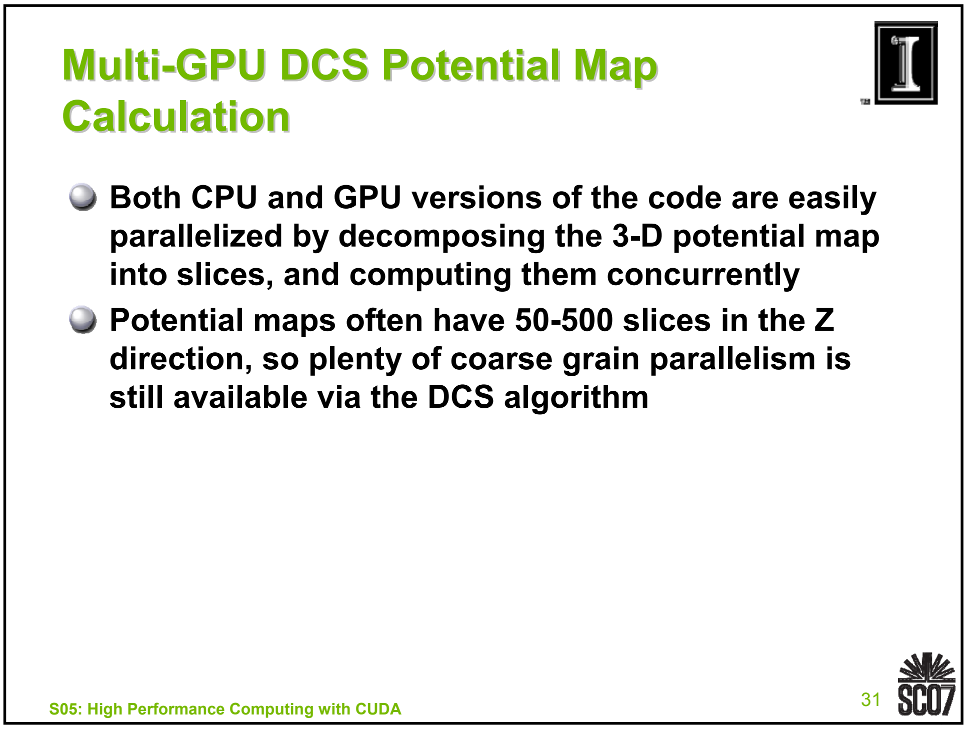
Unrolling increases computational tile size

Grid of thread blocks:

Thread blocks: 64-256 threads

Threads compute up to 8 potentials, skipping by half-warps

Padding waste

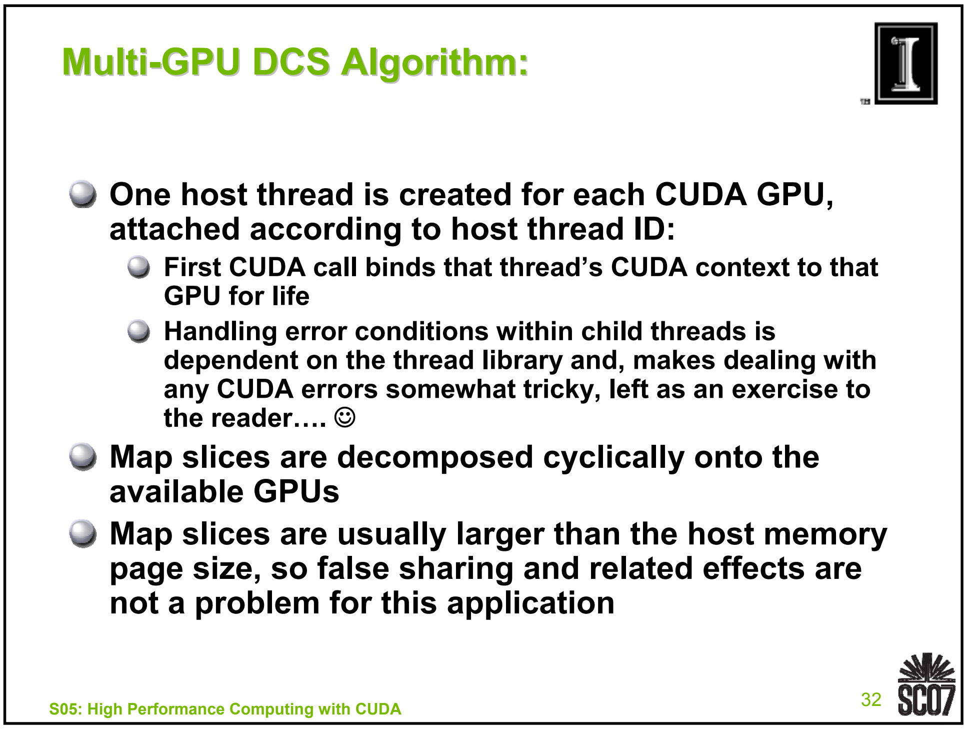


Multi-GPU DCS Potential Map Calculation

- Both CPU and GPU versions of the code are easily parallelized by decomposing the 3-D potential map into slices, and computing them concurrently
- Potential maps often have 50-500 slices in the Z direction, so plenty of coarse grain parallelism is still available via the DCS algorithm

S05: High Performance Computing with CUDA

31



Multi-GPU DCS Algorithm:

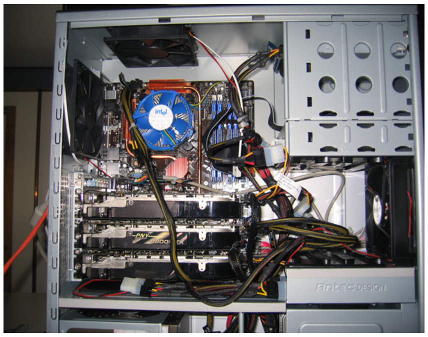
- One host thread is created for each CUDA GPU, attached according to host thread ID:
  - First CUDA call binds that thread's CUDA context to that GPU for life
  - Handling error conditions within child threads is dependent on the thread library and, makes dealing with any CUDA errors somewhat tricky, left as an exercise to the reader…. ☺
- Map slices are decomposed cyclically onto the available GPUs
- Map slices are usually larger than the host memory page size, so false sharing and related effects are not a problem for this application

S05: High Performance Computing with CUDA

32

Multi-GPU DCS Performance

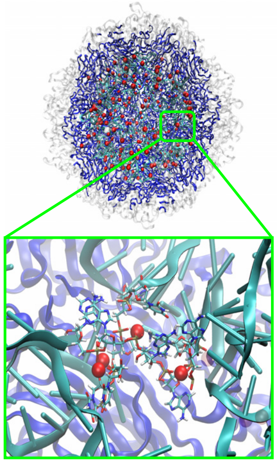
- Effective memory bandwidth scales with the number of GPUs utilized
- PCIe bus bandwidth not a bottleneck for this algorithm
- Multi-GPU DCS achieves over 117 GEvals/sec, 863 GFLOPS on this machine
- Power: three GPU system consumes 700 watts running flat out on all 4 cores, and all three GPUs
- A 4-GPU (dual Quadroplex) Opteron system achieves 157 GEvals/sec, 1.16 TFLOPS



Host system: Quad core Intel QX6700, three NVIDIA GeForce 8800GTX GPUs, RHEL4 Linux

Multi-GPU DCS Performance: Initial Ion Placement Lattice Calculation

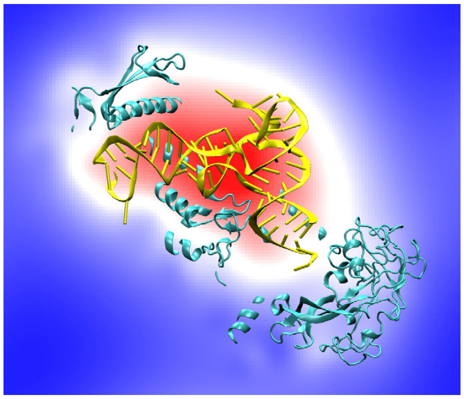
- Original virus DCS ion placement ran for 110 CPU-hours on SGI Altix Itanium2
- Same calculation now takes 1.35 GPU-hours
- 27 minutes (wall clock) if three GPUs are used concurrently
- CUDA Initial ion placement lattice calculation performance:
  - 82 times faster for virus (STMV) structure
  - 110 times faster for ribosome



Satellite Tobacco Mosaic Virus (STMV) Ion Placement

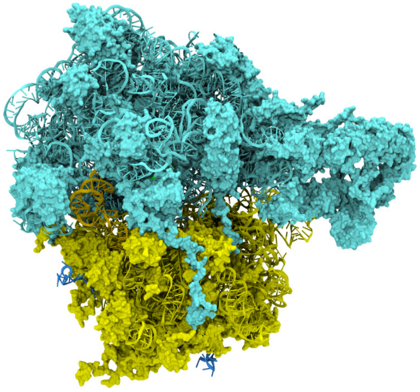
Multi-GPU DCS Performance: Time-averaged Electrostatics Calculation

- DCS algorithm has fewer limitations and restrictions than high performance approximation based methods, allowing it to be used in certain cases that are troublesome for PME and other methods
- Time-averaged DCS electrostatics would be too computationally costly for everyday use on CPUs
- Performance of Multi-GPU DCS algorithm makes it practical to perform mean field calculations over molecular dynamics trajectories

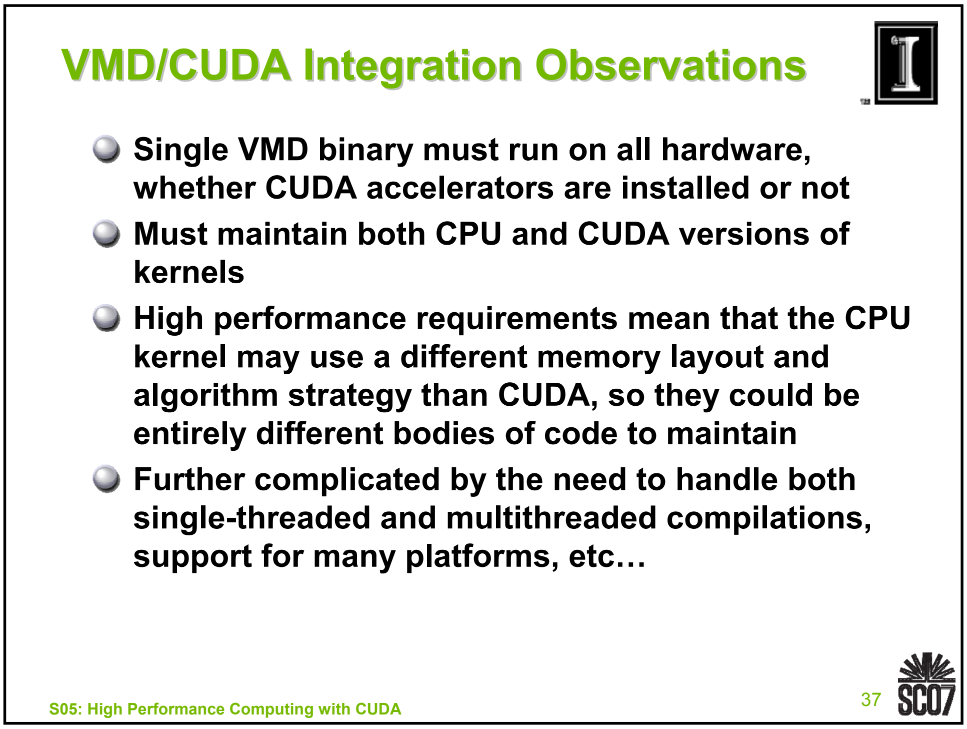


Experiences Integrating CUDA Kernels Into VMD

- VMD: molecular visualization and analysis
- State-of-the-art simulations require more viz/analysis power than ever before
- For some algorithms, CUDA can bring what was previously supercomputer class performance to an appropriately equipped desktop workstation

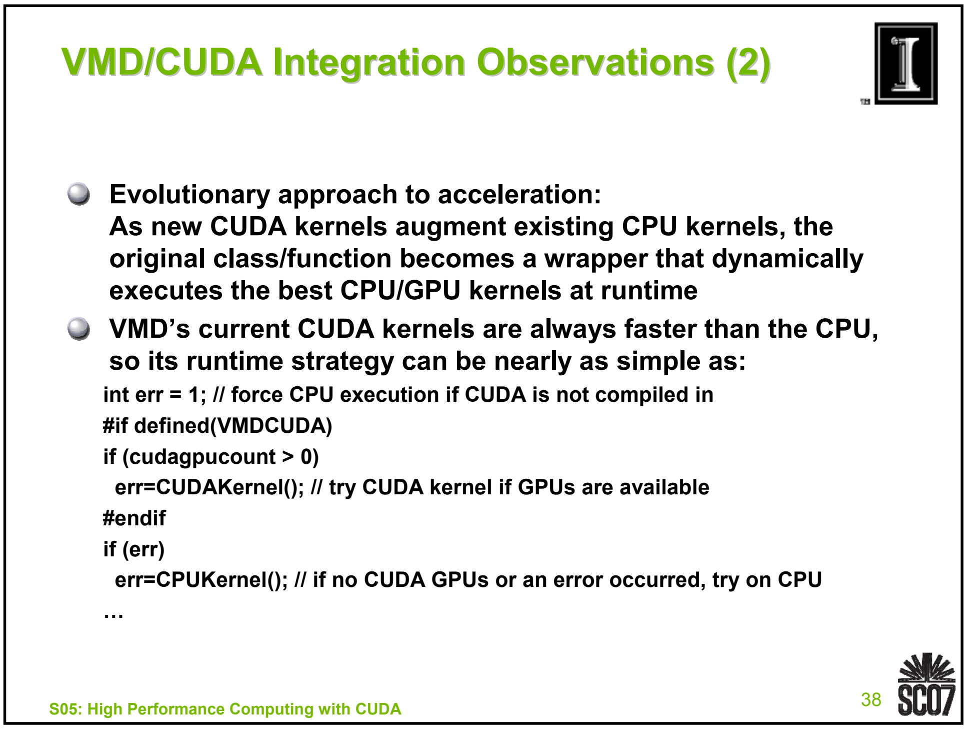


Ribosome: 260,790 atoms before adding solvent/ions



VMD/CUDA Integration Observations

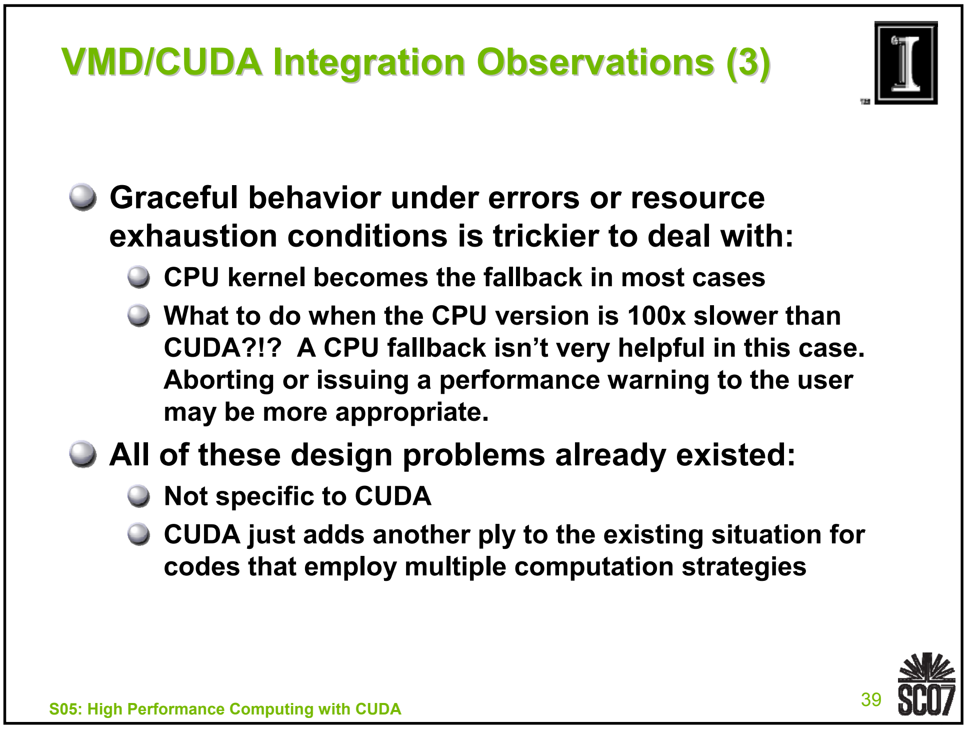
- Single VMD binary must run on all hardware, whether CUDA accelerators are installed or not
- Must maintain both CPU and CUDA versions of kernels
- High performance requirements mean that the CPU kernel may use a different memory layout and algorithm strategy than CUDA, so they could be entirely different bodies of code to maintain
- Further complicated by the need to handle both single-threaded and multithreaded compilations, support for many platforms, etc…



VMD/CUDA Integration Observations (2)

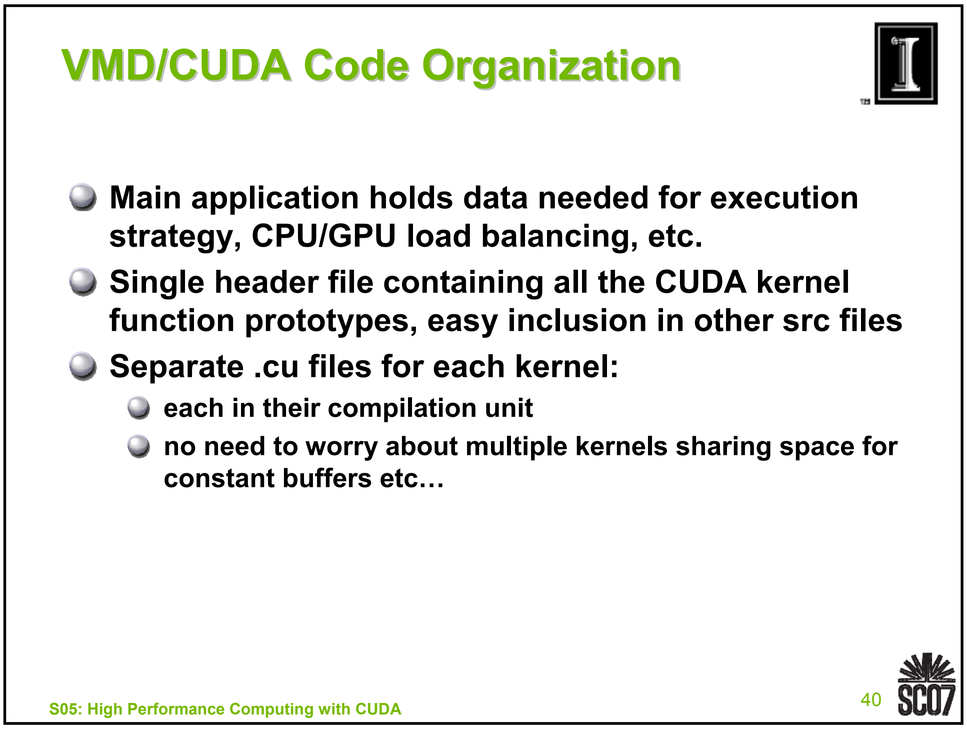
- Evolutionary approach to acceleration:
  As new CUDA kernels augment existing CPU kernels, the original class/function becomes a wrapper that dynamically executes the best CPU/GPU kernels at runtime
- VMD's current CUDA kernels are always faster than the CPU, so its runtime strategy can be nearly as simple as:

```
int err = 1; // force CPU execution if CUDA is not compiled in
#if defined(VMDCUDA)
if (cudagpucount > 0)
  err=CUDAKernel(); // try CUDA kernel if GPUs are available
#endif
if (err)
  err=CPUKernel(); // if no CUDA GPUs or an error occurred, try on CPU
…
```



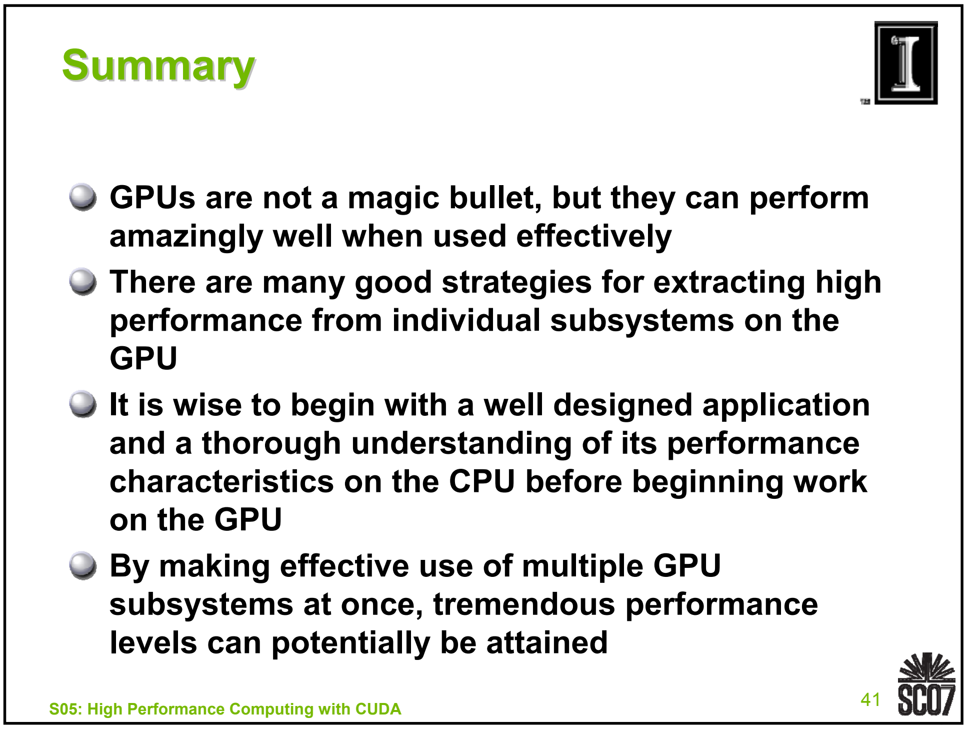
VMD/CUDA Integration Observations (3)

- Graceful behavior under errors or resource exhaustion conditions is trickier to deal with:
  - CPU kernel becomes the fallback in most cases
  - What to do when the CPU version is 100x slower than CUDA?!? A CPU fallback isn't very helpful in this case. Aborting or issuing a performance warning to the user may be more appropriate.
- All of these design problems already existed:
  - Not specific to CUDA
  - CUDA just adds another ply to the existing situation for codes that employ multiple computation strategies



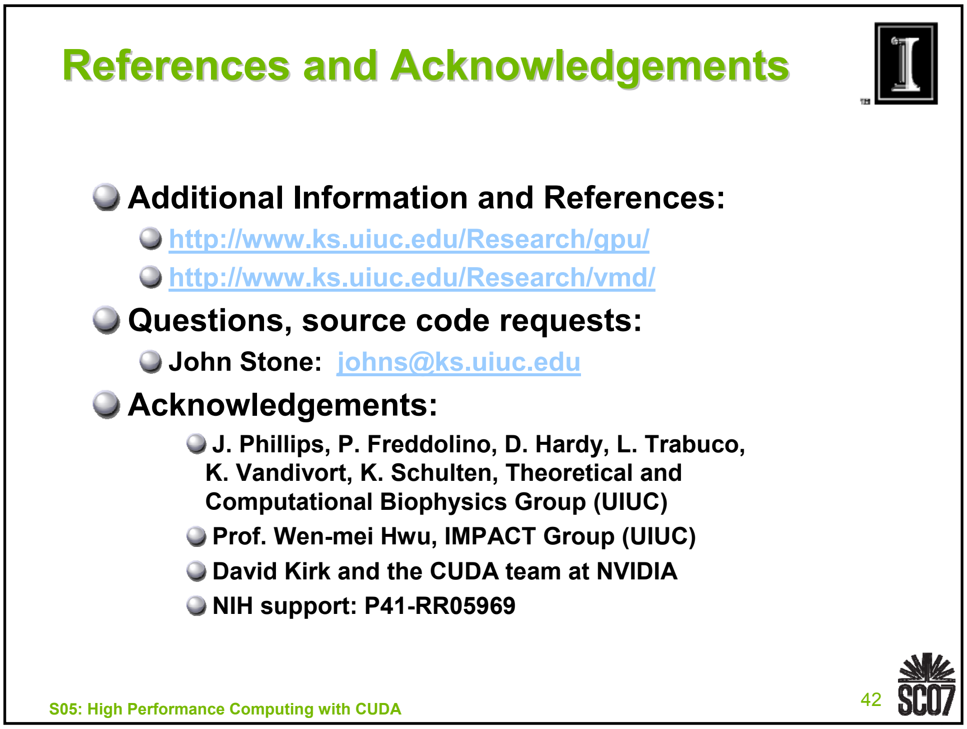
VMD/CUDA Code Organization

- Main application holds data needed for execution strategy, CPU/GPU load balancing, etc.
- Single header file containing all the CUDA kernel function prototypes, easy inclusion in other src files
- Separate .cu files for each kernel:
  - each in their compilation unit
  - no need to worry about multiple kernels sharing space for constant buffers etc…



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



References and Acknowledgements

- Additional Information and References:
  - http://www.ks.uiuc.edu/Research/gpu/
  - http://www.ks.uiuc.edu/Research/vmd/
- Questions, source code requests:
  - John Stone: johns@ks.uiuc.edu
- Acknowledgements:
  - J. Phillips, P. Freddolino, D. Hardy, L. Trabuco, K. Vandivort, K. Schulten, Theoretical and Computational Biophysics Group (UIUC)
  - Prof. Wen-mei Hwu, IMPACT Group (UIUC)
  - David Kirk and the CUDA team at NVIDIA
  - NIH support: P41-RR05969