Using GPUs to compute the multilevel summation of electrostatic forces

David J. Hardy

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/

Multiscale Molecular Modelling,
Edinburgh, June 30 - July 3, 2010

Outline

- Multilevel summation method (MSM)
- GPU architecture and kernel design considerations
- GPU kernel design alternatives for short-range non-bonded interactions
- GPU kernel for MSM long-range part
- Initial performance results for speeding up MD

Multilevel Summation Method

Skeel, *et al.*, *J. Comp. Chem.* **23**:673-684, 2002.

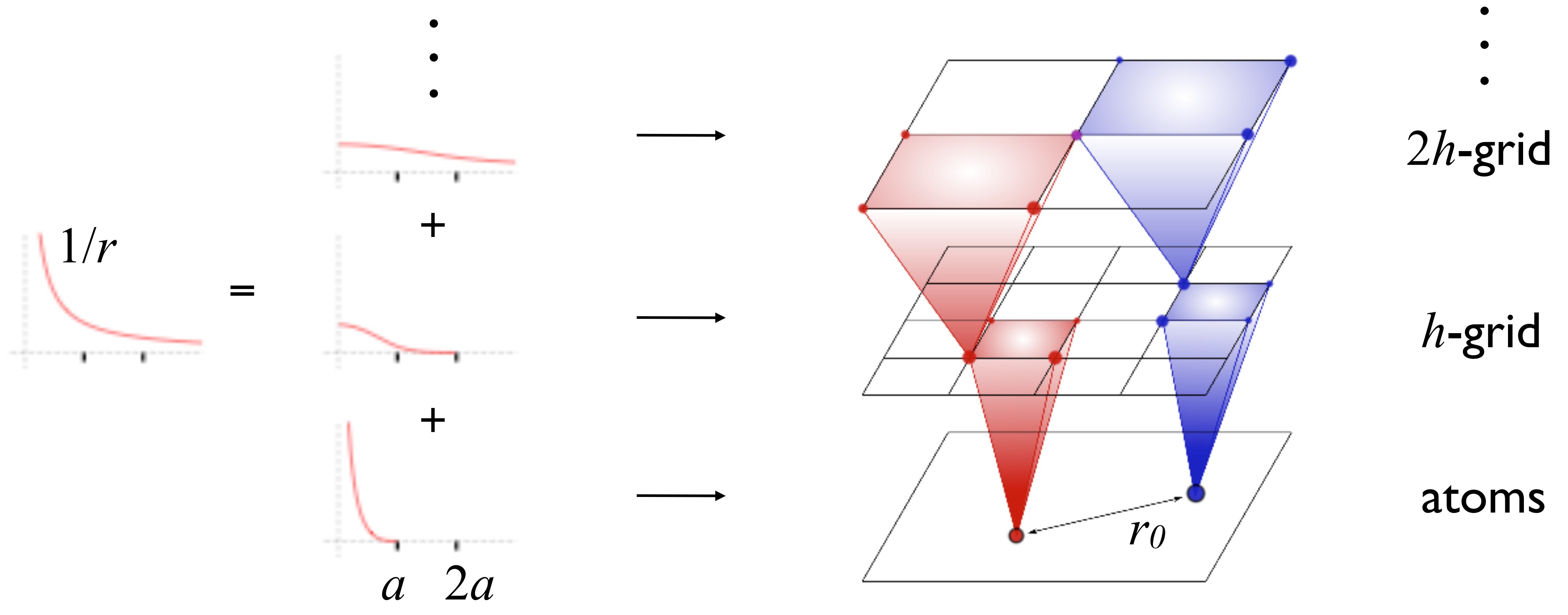
- Fast algorithm for N-body electrostatics
- Calculates sum of smoothed pairwise potentials interpolated from a hierarchal nesting of grids
- Advantages over PME (particle-mesh Ewald) and/or FMM (fast multipole method):
  - Algorithm has linear time complexity
  - Allows non-periodic or periodic boundaries
  - Produces continuous forces for dynamics (advantage over FMM)
  - Avoids 3D FFTs for better parallel scaling (advantage over PME)
  - Permits polynomial splittings (no *erfc()* evaluation, as used by PME)
  - Spatial separation allows use of multiple time steps
  - Can be extended to other types of pairwise interactions

MSM Main Ideas

- Split the $1/r$ potential into a short-range cutoff part plus smoothed parts that are successively more slowly varying. All but the top level potential are cut off.
- Smoothed potentials are interpolated from successively coarser grids.
- Finest grid spacing h and smallest cutoff distance a are doubled at each successive level.

Split the $1/r$ potential

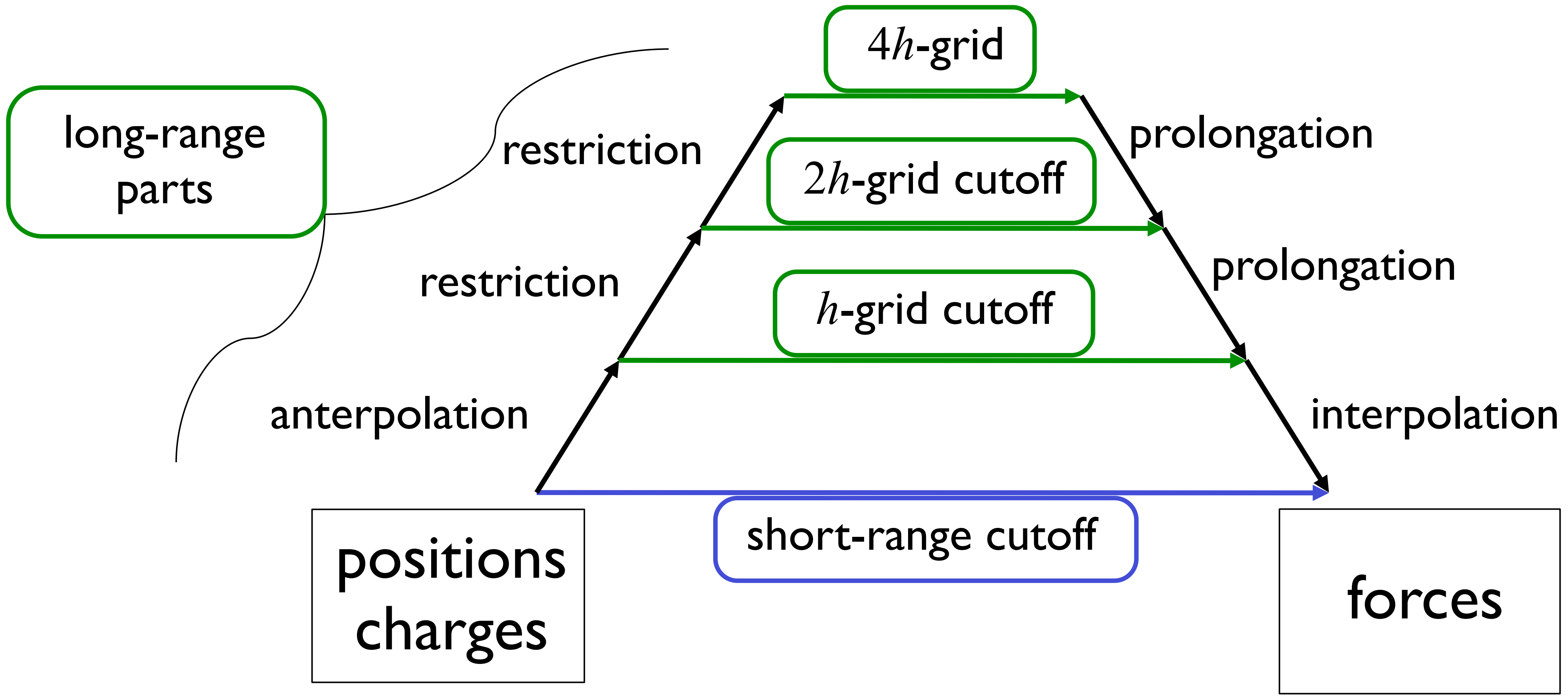
Interpolate the smoothed potentials



MSM Calculation

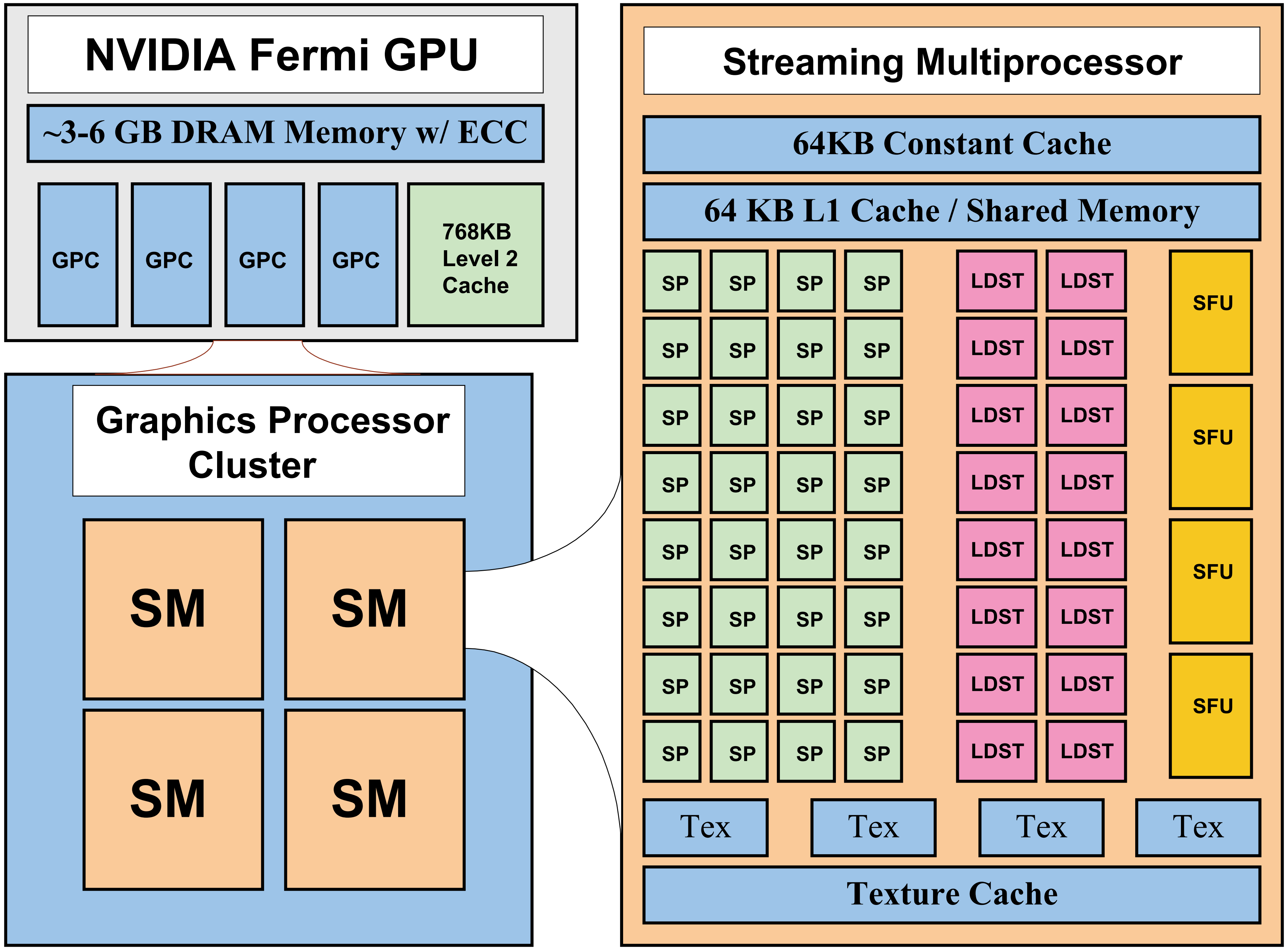
$$\boxed{\textbf{force} = \begin{array}{c}\textbf{exact}\\ \textbf{short-range}\\ \textbf{part}\end{array} + \begin{array}{c}\textbf{interpolated}\\ \textbf{long-range}\\ \textbf{part}\end{array}}$$

Computational Steps



GPU Computing Concepts

- Heterogeneous computing model
  - CPU host manages GPU devices, memories, invokes “kernels”
- GPU hardware supports standard integer and floating point types and math operations, designed for fine-grained data parallelism
  - Hundreds of “threads” grouped together into a “block” performing SIMD execution
  - Need hundreds of blocks (~10,000 - 30,000 threads) to fully utilize hardware
- Great speedups attainable (with commodity hardware)
  - 10x - 20x over one CPU core are common
  - 100x or more possible in some cases
- Programming toolkits (CUDA and OpenCL) in C/C++ dialects, allowing GPUs to be integrated into legacy software



SP = Streaming Processor

SFU = Special Function Unit

Tex = Texture Unit

NVIDIA Fermi New Features

- ECC memory support
- L1 (64 KB) and L2 (768 KB) caches
- 512 cores, faster arithmetic ("madd") by 2x
  - HOWEVER: memory bandwidth increases only 20 - 30%
- Support for additional functions in SFU: erfc() for PME!
  - HOWEVER: no commensurate improvement in SFU performance
- Improved double precision performance and accuracy
- Improved integer performance
- Allow multiple kernel execution
  - Could prove extremely important for keeping GPU fully utilized

GPU Kernel Design (CUDA)

- Problem must offer substantial data parallelism
- Data access using gather memory patterns (reads) rather than scatter (writes)
- Increase arithmetic intensity through effective use of memory subsystems
  - registers: the fastest memory, but smallest in size
  - constant memory: near register-speed when all threads together read a single location (fast broadcast)
  - shared memory: near register-speed when all threads access without bank conflicts
  - texture cache: can improve performance for irregular memory access
  - global memory: large but slow, coalesced access for best performance
- May benefit from trading memory access for more arithmetic

Additional GPU Considerations

- Coalescing reads and writes from global memory
- Avoiding bank conflicts accessing shared memory
- Avoiding branch divergence
- Synchronizing threads within thread blocks
- Atomic memory operations can provide synchronization across thread blocks
- “Stream” management for invoking kernels and transferring memory asynchronously
- Page-locked host memory to speed up memory transfers between CPU and GPU

Designing GPU Kernels for Short-range Non-bonded Forces

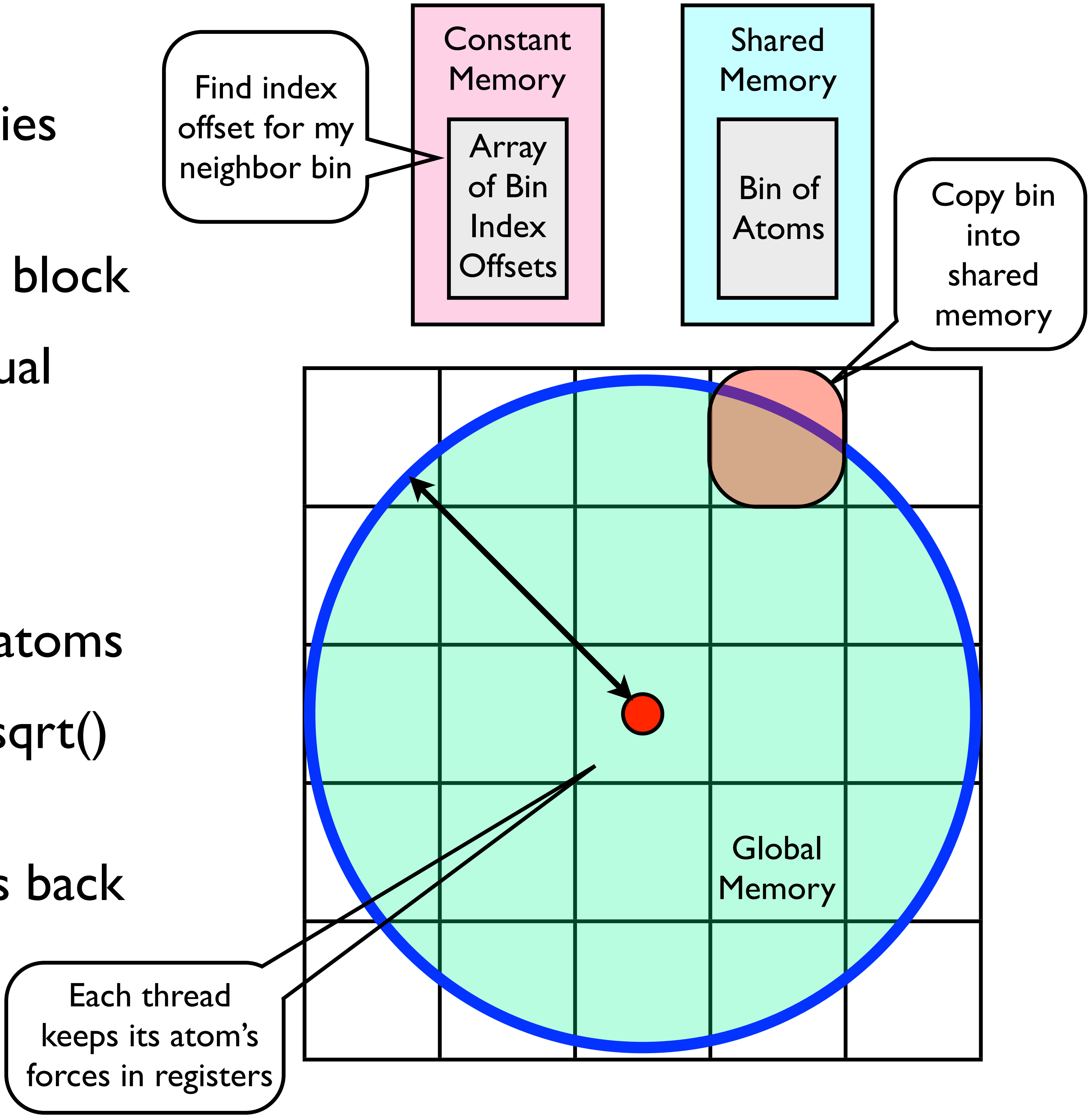
- Calculate both electrostatics and van der Waals interactions (need atom coordinates and parameters)
- Spatial hashing of atoms into bins (best done on CPU)
- Should we use pairlists?
  - Reduces computation, increases and delocalizes memory access
- Should we make use of Newton's 3rd Law to reduce work?
- Is single precision enough? Do we need double precision?
- How might we handle non-bonded exclusions?
  - Detect and omit excluded pairs (use bit masks)
  - Ignore, fix with CPU (use force clamping)

Designing GPU Kernels for Short-range Non-bonded Forces

- How do we map work to the GPU threads?
  - Fine-grained: assign threads to sum forces on atoms
  - Extremely fine-grained: assign threads to pairwise interactions
- How do we decompose work into thread blocks?
  - Non-uniform: assign thread blocks to bins
  - Uniform: assign thread blocks to entries of the force matrix
- How do we compute potential energies or the virial?
- How do we calculate expensive functional forms?
  - PME requires erfc(): is it faster to use an interpolation table?
- Other issues: supporting NBFix parameters

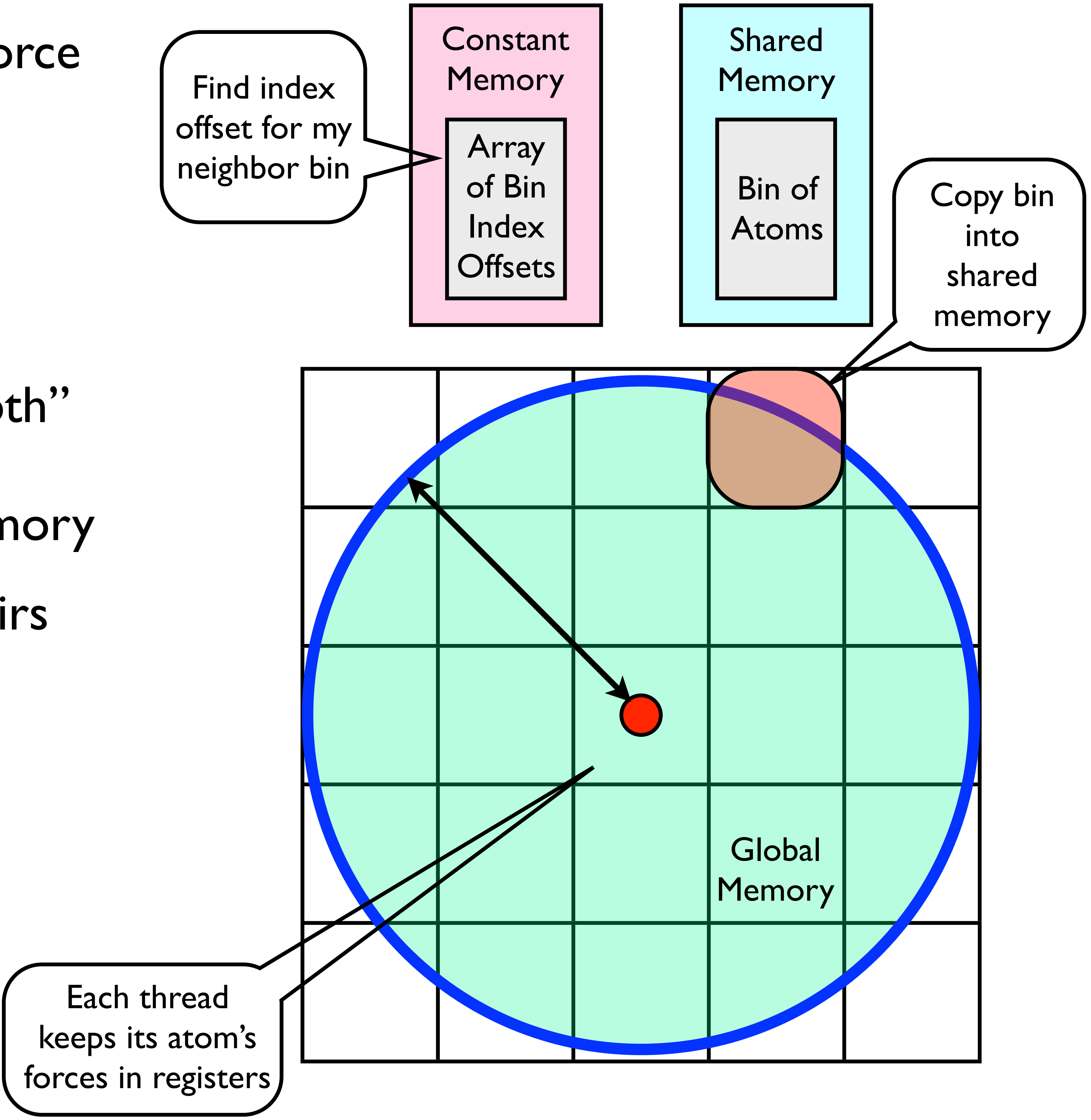
GPU Kernel for Short-range MSM

- CPU sorts atoms into bins, copies bins to GPU global memory
- Each bin is assigned to a thread block
- Threads are assigned to individual atoms
- Loop over surrounding neighborhood of bins, summing forces and energies from their atoms
- Calculation for MSM involves rsqrt() plus several multiplies and adds
- CPU copies forces and energies back from GPU global memory


Find index offset for my neighbor bin
Constant Memory
Array of Bin Index Offsets
Shared Memory
Bin of Atoms
Copy bin into shared memory
Global Memory
Each thread keeps its atom's forces in registers

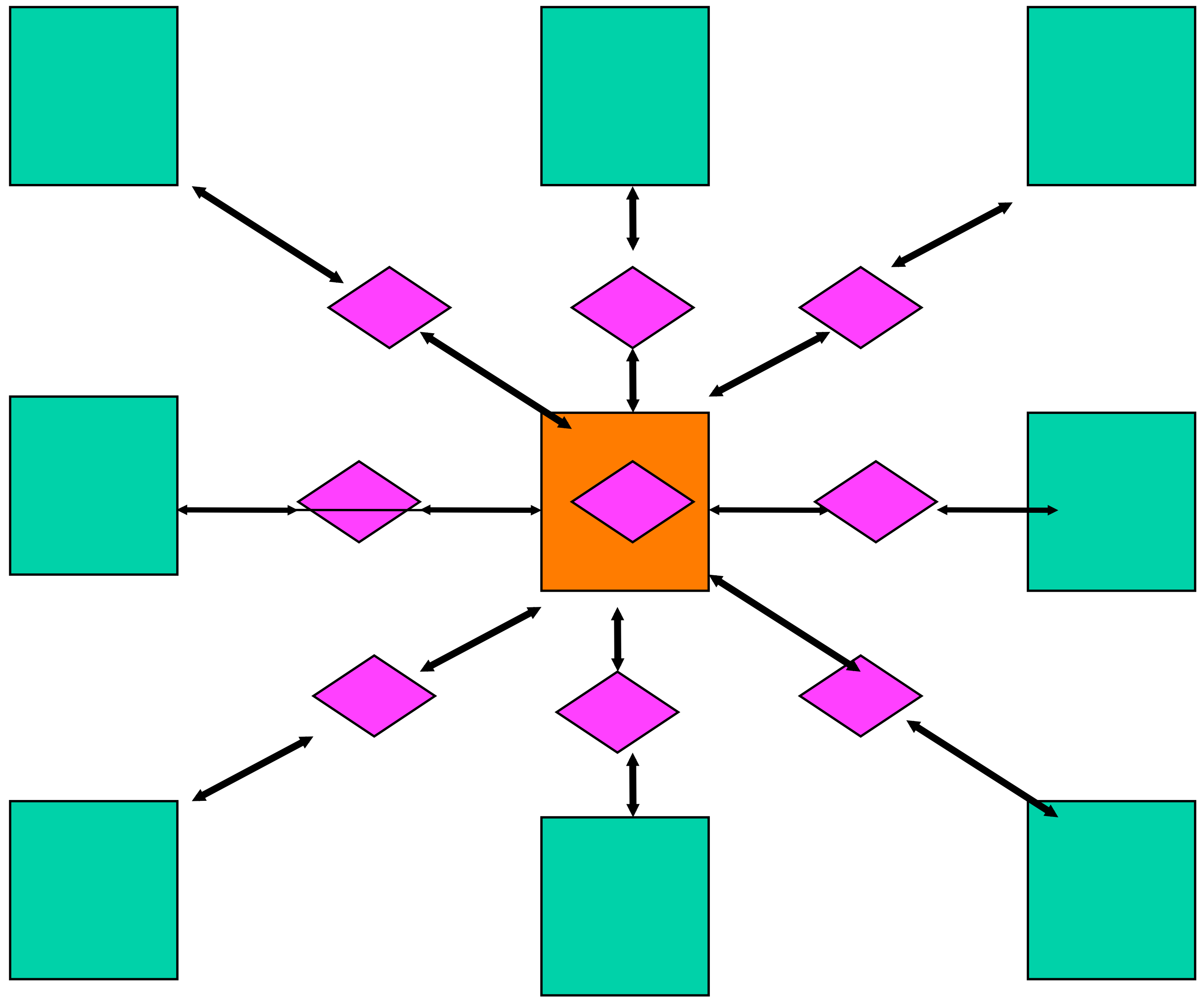
GPU Kernel for Short-range MSM

- Each thread accumulates atom force and energies in registers
- Bin neighborhood index offsets stored in constant memory
- Load atom bin data into shared memory; atom data and bin "depth" are carefully chosen to permit coalesced reads from global memory
- Check for and omit excluded pairs
- Thread block performs sum reduction of energies
- Coalesced writing of forces and energies (with padding) to GPU global memory
- CPU sums energies from bins


Find index offset for my neighbor bin
Constant Memory
Array of Bin Index Offsets
Shared Memory
Bin of Atoms
Copy bin into shared memory
Global Memory
Each thread keeps its atom's forces in registers

NAMD Hybrid Decomposition

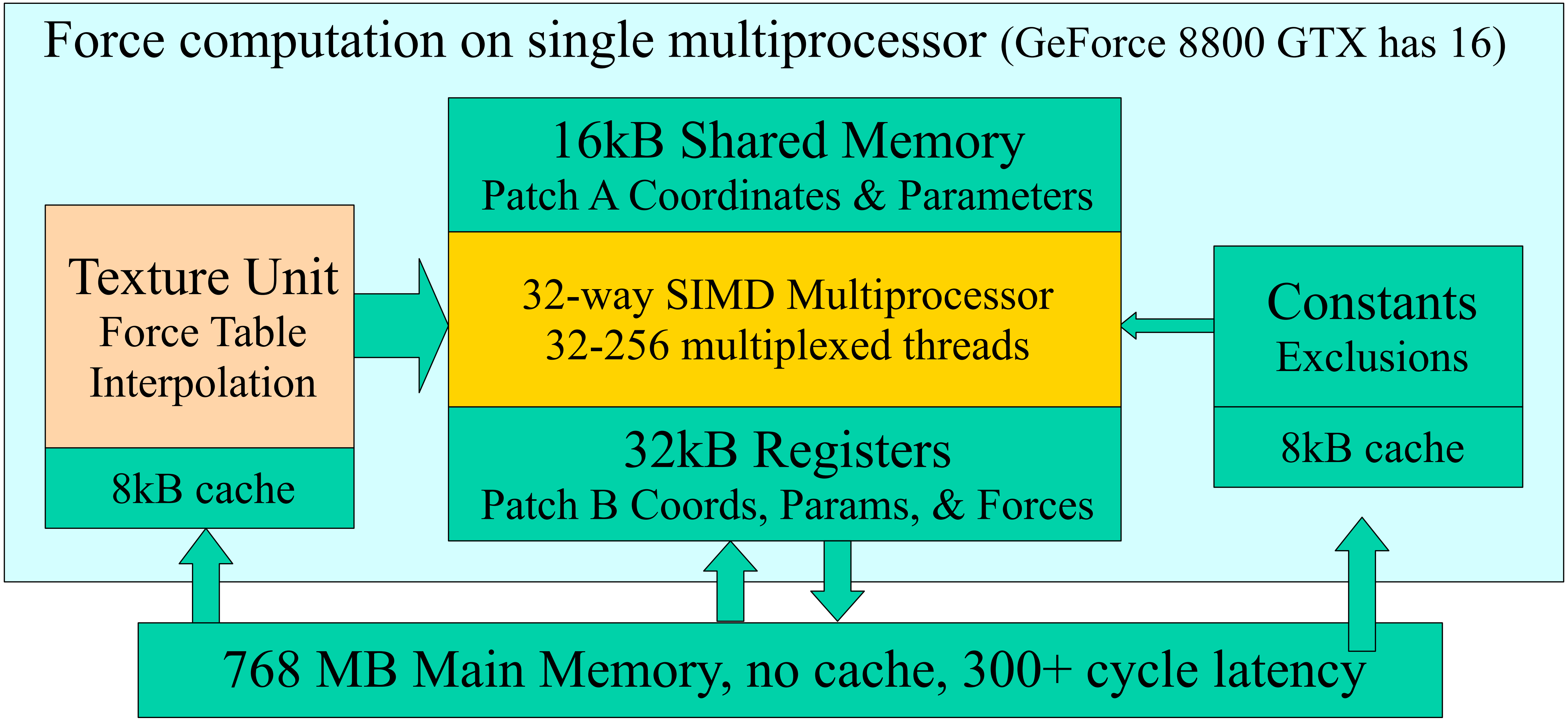
Kale *et al.*, *J. Comp. Phys.* **151**:283-312, 1999.



- Spatially decompose data and communication.
- Separate but related work decomposition.
- “Compute objects” facilitate iterative, measurement-based load balancing system.

NAMD Non-bonded Forces on GPU

- Decompose work into pairs of “patches” (bins), identical to NAMD structure.
- Each patch-pair is calculated by an SM (thread block).

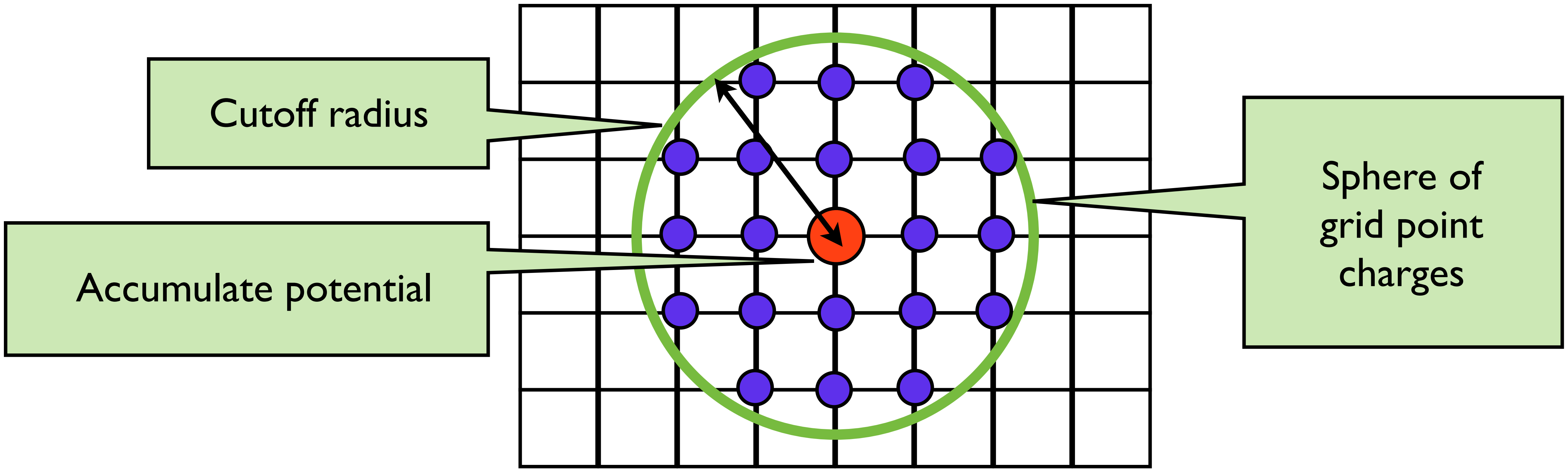


Force computation on single multiprocessor (GeForce 8800 GTX has 16)
16kB Shared Memory
Patch A Coordinates & Parameters
Texture Unit
Force Table
Interpolation
8kB cache
32-way SIMD Multiprocessor
32-256 multiplexed threads
Constants
Exclusions
8kB cache
32kB Registers
Patch B Coords, Params, & Forces
768 MB Main Memory, no cache, 300+ cycle latency

Stone *et al.*, *J. Comp. Chem.* **28**:2618-2640, 2007.

MSM Grid Interactions

- Potential summed from grid point charges within cutoff
- Uniform spacing enables distance-based interactions to be precomputed as stencil of "weights"
- Weights at each level are identical up to scaling factor (!)
- Calculate as 3D convolution of weights
  - stencil size up to 23x23x23

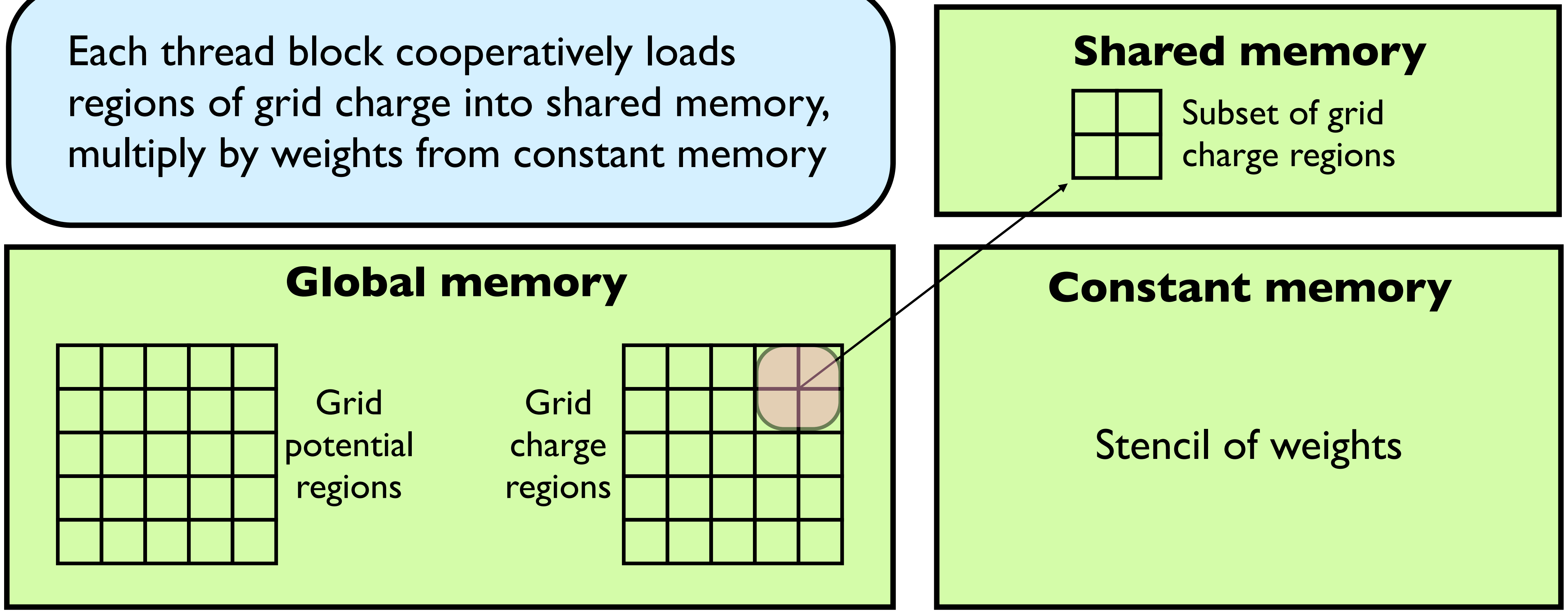


MSM Grid Interactions on GPU

- Store weights in constant memory (padded up to next multiple of 4)
- Thread block calculates 4x4x4 region of potentials (stored contiguously)
- Pack all regions over all levels into 1D array (each level padded with zero-charge region)
- Store map of level array offsets in constant memory
- Kernel has thread block loop over surrounding regions of charge (load into shared memory)
- All grid levels are calculated concurrently, scaled by level factor (keeps GPU from running out of work at upper grid levels)

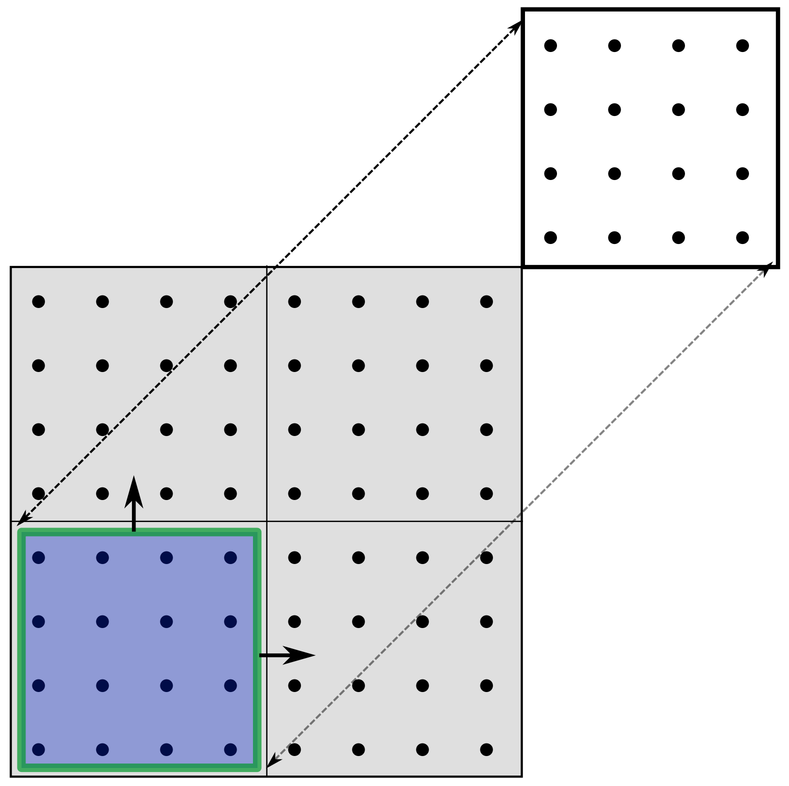
Hardy, *et al.*, *J. Paral. Comp.* **35**:164-177, 2009.

Each thread block cooperatively loads regions of grid charge into shared memory, multiply by weights from constant memory



Apply Weights Using Sliding Window

- Thread block must collectively use same value from constant memory
- Read 8x8x8 grid charges (8 regions) into shared memory
- Window of size 4x4x4 maintains same relative distances
- Slide window by 4 shifts along each dimension



Initial Results

(GPU: NVIDIA GTX-285, using CUDA 3.0; CPU: 2.4 GHz Intel Core 2 Q6600 quad core)

Box of 21950 flexible waters, 12 A cutoff, 1ps	CPU only	with GPU	Speedup vs. NAMD/CPU
NAMD with PME	1199.8 s	210.5 s	5.7 x
NAMD-Lite with MSM	5183.3 s (4598.6 short, 572.23 long)	176.6 s (93.9 short, 63.1 long)	6.8 x (19% over NAMD/GPU)

Concluding Remarks

- Redesign of MD algorithms for GPUs is important:
  - Commodity GPUs offer great speedups for well-constructed codes
  - CPU single core performance is not improving
  - Multicore CPUs benefit from redesign efforts (e.g., OpenCL can target multicore CPUs)
- MSM offers advantages that benefit GPUs:
  - Short-range splitting using polynomial of r^2
  - Calculation on uniform grids
- More investigation into alternative designs for MD algorithms

Acknowledgments

- Bob Skeel (Purdue University)
- John Stone (University of Illinois at Urbana-Champaign)
- Jim Phillips (University of Illinois at Urbana-Champaign)
- Klaus Schulten (University of Illinois at Urbana-Champaign)
- Funding from NSF and NIH