GPU Particle-Particle Algorithms: Non-bonded Force Calculation

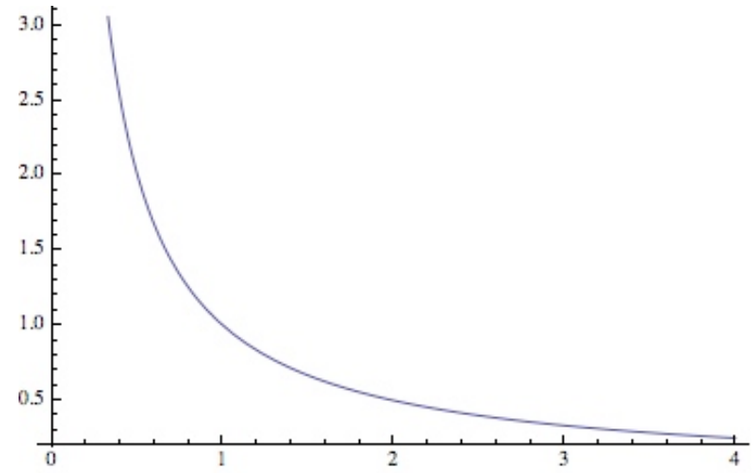
David J. Hardy

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/

GPU Programming for Molecular Modeling Workshop,
University of Illinois at Urbana-Champaign, August 2-4, 2013

Non-bonded Potential Functions

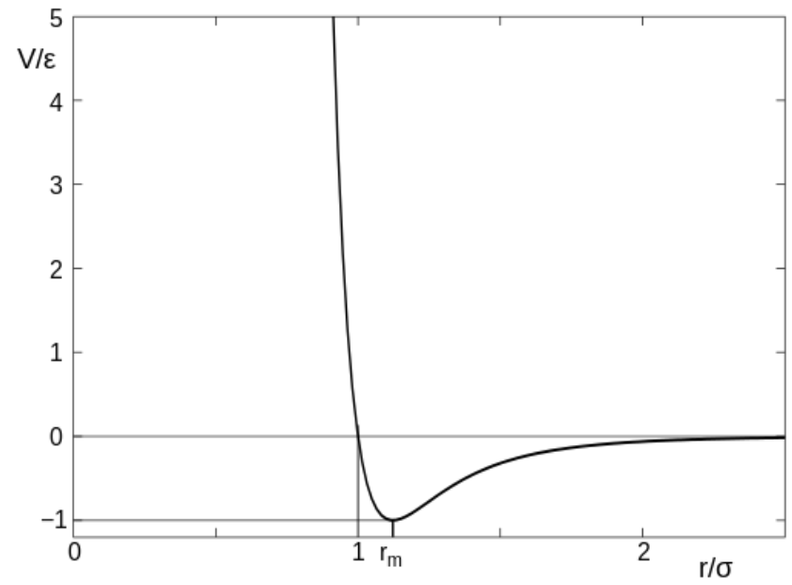
electrostatics: $$U_{\mathrm{elec}} = \sum_{i<j} C \frac{q_i q_j}{|\mathbf{r}_j - \mathbf{r}_i|}$$



van der Waals: $$U_{\mathrm{LJ}} = \sum_{i<j} \epsilon_{ij} \left[\left(\frac{r_{ij}^{\min}}{|\mathbf{r}_j - \mathbf{r}_i|} \right)^{12} - 2 \left(\frac{r_{ij}^{\min}}{|\mathbf{r}_j - \mathbf{r}_i|} \right)^{6} \right]$$

$$r_{ij}^{\min} = \frac{1}{2} \left(r_i^{\min} + r_j^{\min} \right)$$

$$\epsilon_{ij} = \sqrt{\epsilon_i \epsilon_j}$$



Forces obtained from gradients of potential functions.

Limit Interaction Range

Cutoff distance a with smoothing $S(r) = 0,\ r > a$: $\mathcal{O}(a^3 N)$

van der Waals: $$U_{\mathrm{LJ}} = \sum_{i<j} \epsilon_{ij} \left[\left(\frac{r_{ij}^{\min}}{r_{ij}} \right)^{12} - 2 \left(\frac{r_{ij}^{\min}}{r_{ij}} \right)^{6} \right] S_{\mathrm{LJ}}(r_{ij})$$

electrostatics: $$U_{\mathrm{elec}} = \sum_{i<j} C q_i q_j \left(\frac{S(r_{ij})}{r_{ij}} + \frac{1 - S(r_{ij})}{r_{ij}} \right)$$

Short-range part, calculate exactly

Long-range part, calculate approximately

Use methods like PME (particle-mesh Ewald)
or MSM (multilevel summation method)

Effective Use of GPU

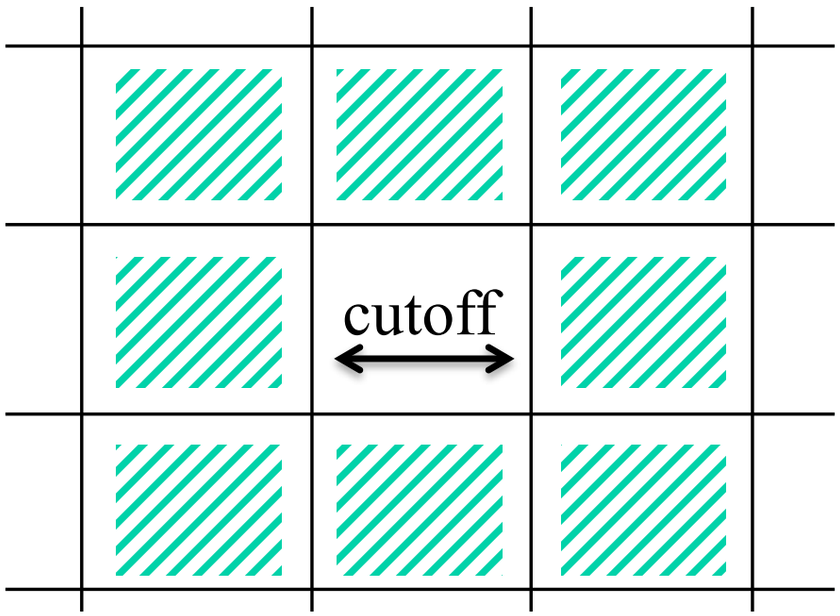
- Large amount of fine-grained parallelism
  - Keep 10,000+ threads busy with consistent calculations
- Efficient memory access patterns
  - Need good understanding of memory hierarchy
    - *Register file*, *shared memory cache* — tradeoff with thread block scheduling
    - *Constant memory*, *texture memory* — read-only, distinctive access patterns
    - *Main memory* — slow access, use coalesced reads and writes
  - Adapt data to memory constraints
- Using suitable control structures and operations
  - Reduce branch divergence, thread synchronization
  - Simplify loop control logic, indexing arithmetic
  - Appropriate use of special function unit

Comparison with Particle-Grid Algorithms

- Particle-particle is more challenging because:
  - Irregularity makes it harder to map work to threads
  - Less fine-grained parallelism available:
    - N particles, M grid points: $10N \leq M \leq 100N$
    - MN particle-grid vs. $\frac{1}{2}N^2$ particle-particle interactions
  - Amount of fine-grained parallelism reduced by factor of 20 to 200
- Particle-particle interactions require more data per interaction

Short-range Non-bonded Interactions

- Sum interactions within cutoff distance a
  - Perform spatial hashing of atoms into grid cells
  - For every grid cell, for each atom:
    - ✦ Loop over atoms in each neighboring cell
    - ✦ If $r_{ij}^2 < a^2$, sum potential energy, virial, and atomic forces
  - Use Newton's 3rd Law: $\mathbf{f}_{ji} = -\mathbf{f}_{ij}$



If cutoff distance is no bigger than cell, then loop over nearest neighbors

NAMD terminology:

grid cells are "patches"

spatial (re-)hashing is "atom migration"

Non-bonded Exclusions

- Self interactions are excluded
- Typically exclude pairs of atoms that are covalently bonded to each other or to a common atom
- Possible approaches:
  - Ignore and correct later
    - ✦ But this can cause large numerical errors!
  - Detect during evaluation and skip

Algorithmic Enhancements

- Maintain pair lists
  - For each atom i, keep list of atoms j within cutoff
  - Extend cutoff distance (a+δ), no update needed until an atom moves distance $\delta/2$
- Interpolation tables for interactions
  - Avoid erfc and exp functions needed for PME
  - Avoid rsqrt (on x86)
  - Avoid additional branching and calculation for van der Waals switching function

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?

Designing GPU Kernels for Short-range Non-bonded Forces

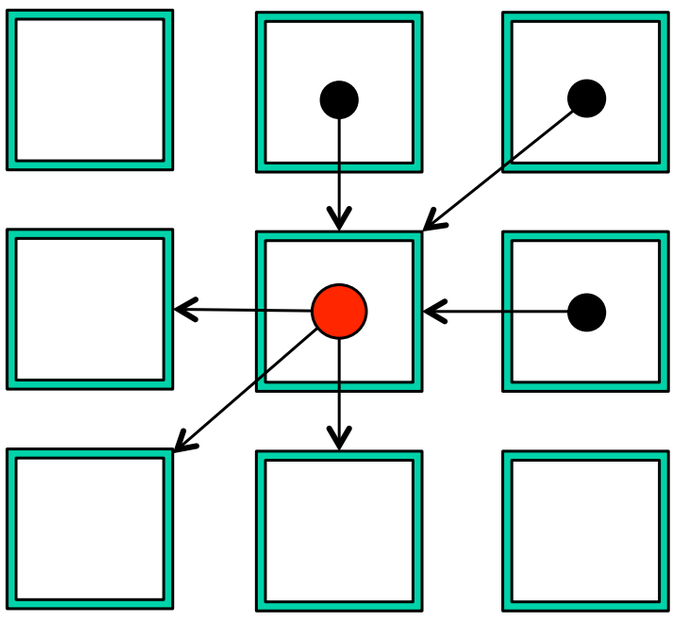
- How do we map work to the GPU threads?
  - Fine-grained: assign threads to sum forces on atoms
  - Extremely fine-grained: assign threads to pairwise interactions
- How do we decompose work into thread blocks?
  - Non-uniform: assign thread blocks to bins
  - Uniform: assign thread blocks to entries of the force matrix

Designing GPU Kernels for Short-range Non-bonded Forces

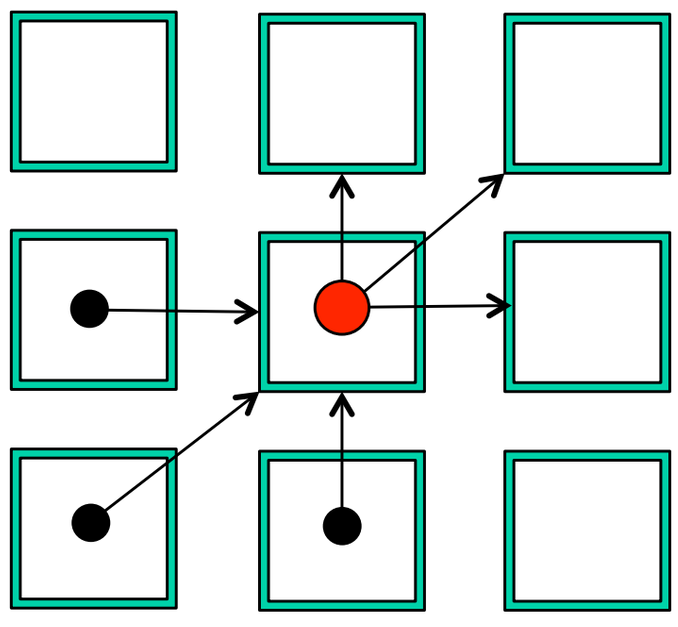
- Is single precision enough? Do we need double precision?
- How might we handle non-bonded exclusions?
  - Detect and omit excluded pairs (use bit masks)
  - Ignore, fix with CPU (use force clamping)
- How do we compute potential energies or the virial?
- How do we calculate expensive functional forms?
  - PME requires erfc(): is it faster to use an interpolation table?

Short-range Parallelization

- Spatial decomposition
- Assign grid cells to processors
- Maps naturally to 3D mesh topology
  - Communication with nearest neighbors

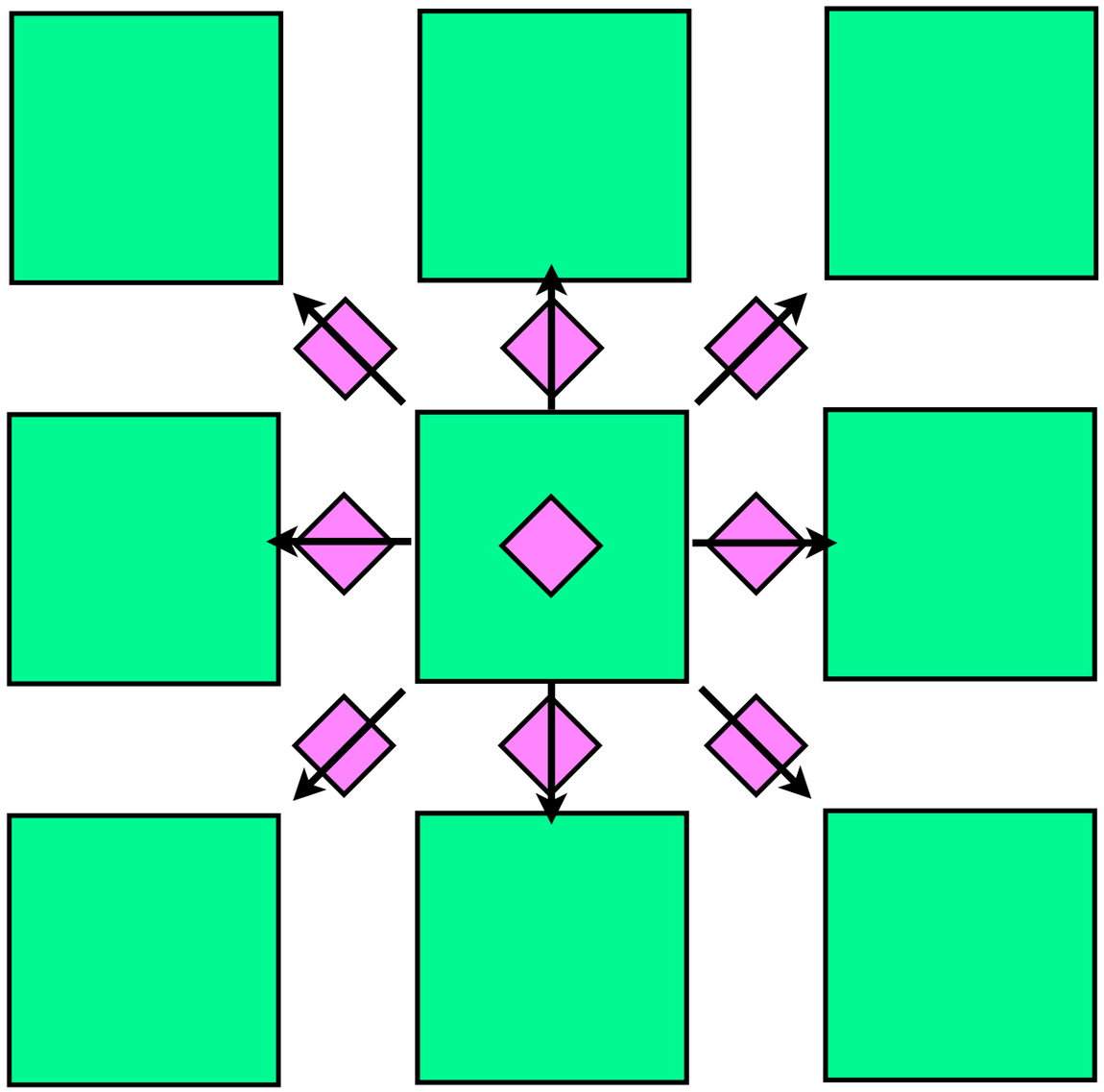


NAMD sends positions downstream, then sends forces upstream



NAMD hybrid decomposition

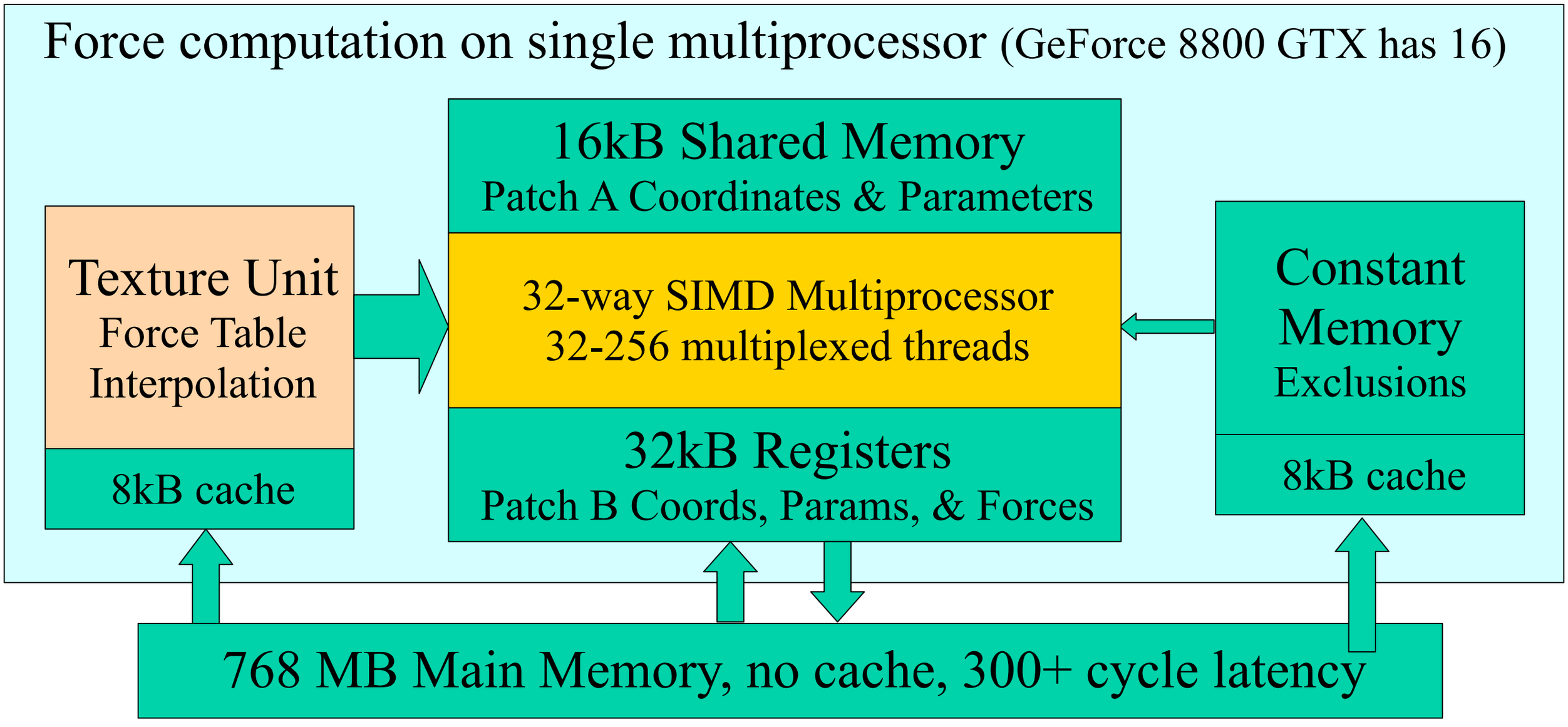
Kale, *et al., J. Comp. Phys.* **151**:283-312, 1999



- Decompose data spatially into *patches*
- Decompose work into concurrent *compute objects*
- Compute objects facilitate iterative, measurement-based load balancing

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).



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

Each Block Gets a Pair of Patches

- Store block-level constants in shared memory to save registers.
- Structure patch_pair is 16-byte aligned.
- To coalesce read have each thread load one int from global memory and write it into a union in shared memory.

```
#define myPatchPair pp.pp
 __shared__ union { patch_pair pp; unsigned int i[PPSIZE]; } pp;
 __shared__ bool same_patch;
 __shared__ bool self_force;

if ( threadIdx.x < (sizeof(patch_pair)>>2) ) {
  unsigned int tmp = ((unsigned int*)patch_pairs)[
                        (sizeof(patch_pair)>>2)*blockIdx.x+threadIdx.x];
  pp.i[threadIdx.x] = tmp;
}
__syncthreads();
// now all threads can access myPatchPair safely
```

Right-Sized Atom Data Structures

```
#define BLOCK_SIZE 128
#define SHARED_SIZE 32

struct __align__(16) atom {  // must be multiple of 16!
  float3 position;
  float charge;
};

struct __align__(16) atom_param {  // must be multiple of 16!
  int vdw_type;
  int index;
  int excl_index;
  int excl_maxdiff;  // maxdiff == 0 -> only excluded from self
};

__shared__ union {
  atom d[SHARED_SIZE];
  unsigned int i[4*SHARED_SIZE];
  float f[4*SHARED_SIZE];
} jpqu;

__shared__ union {
  atom_param d[SHARED_SIZE];
  unsigned int i[4*SHARED_SIZE];
} japu;
```

CPU Force Interpolation

- Want to avoid calculating erfc(), sqrt(), branches for switching functions.
- $U(r^2) = \varepsilon(\sigma^{12}A(r^2) + \sigma^6B(r^2)) + qqC(r^2)$
- $\mathbf{F} = -2\,\mathbf{r}\,U'(r^2)$
- Piecewise cubic interpolation of A,B,C.
- Need more windows at small r^2, so use exponent and high-order mantissa bits in floating point format to determine window.

Texture Unit Force Interpolation

- Bit manipulation of floats is not possible.
- But rsqrt() is implemented in hardware.
- $F(r^{-1})/r = \varepsilon(\sigma^{12}A(r^{-1}) + \sigma^{6}B(r^{-1})) + qqC(r^{-1})$
- $\mathbf{F} = \mathbf{r}\ F(r^{-1})/r$
- Piecewise linear interpolation of A,B,C is equivalent to linear interpolation of force **F**
  - since $r\ (a\ r^{-1} + b) = a + r\ b$
- Texture unit hardware is a perfect match.

Constant Memory Exclusion Tables

- Need to exclude bonded pairs of atoms.
  - Also apply correction for PME electrostatics.
- Exclusions determined by using atom indices to bit flags in exclusion arrays.
- Repetitive molecular structures limit unique exclusion arrays.
- If table is too large for constant memory, overflow is handled by reading from main memory.

Overview of Inner Loop

- Calculate forces on atoms in registers due to atoms in shared memory.
  - Ignore Newton's 3rd law (reciprocal forces).
  - Do not sum forces for atoms in shared memory.
- All threads access the same shared memory atom, allowing shared memory broadcast.
- Only calculate forces for atoms within cutoff distance (roughly 10% of pairs).

Nonbonded Forces CUDA Code

```
texture<float4> force_table;
__constant__ unsigned int exclusions[];
__shared__ atom jatom[];
atom iatom;      // per-thread atom, stored in registers
float4 iforce;   // per-thread force, stored in registers
for ( int j = 0; j < jatom_count; ++j ) {
  float dx = jatom[j].x - iatom.x;   float dy = jatom[j].y - iatom.y;  float dz = jatom[j].z - iatom.z;
  float r2 = dx*dx + dy*dy + dz*dz;
  if ( r2 < cutoff2 ) {
    float4 ft = texfetch(force_table, 1.f/sqrt(r2));                    Force Interpolation
    bool excluded = false;
    int indexdiff = iatom.index - jatom[j].index;                       Exclusions
    if ( abs(indexdiff) <= (int) jatom[j].excl_maxdiff ) {
      indexdiff += jatom[j].excl_index;
      excluded = ((exclusions[indexdiff>>5] & (1<<(indexdiff&31))) != 0);
    }
    float f = iatom.half_sigma + jatom[j].half_sigma;  // sigma
    f *= f*f;  // sigma^3
    f *= f;  // sigma^6
    f *= ( f * ft.x + ft.y );  // sigma^12 * fi.x - sigma^6 * fi.y
    f *= iatom.sqrt_epsilon * jatom[j].sqrt_epsilon;                    Parameters
    float qq = iatom.charge * jatom[j].charge;
    if ( excluded ) { f = qq * ft.w; }  // PME correction
    else { f += qq * ft.z; }  // Coulomb
    iforce.x += dx * f;   iforce.y += dy * f;    iforce.z += dz * f;
    iforce.w += 1.f;  // interaction count or energy                    Accumulation
  }
}
```

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

NAMD CUDA Kernel Evolution

- Original — minimize main memory access
  - Enough threads to load all atoms in patch
  - Needed two atoms per thread to fit
  - Swap atoms between shared memory and registers
- Revised — multiple blocks for concurrency
  - 64 threads/atoms per block (now 128 for Fermi)
  - Loop over shared memory atoms in sets of 16
  - Two blocks for each patch pair

Further Kernel Developments

- Production features in NAMD 2.7b3 release (7/6/2010):
  - Full electrostatics with PME
  - 1-4 exclusions
  - Constant-pressure simulation
  - Improved force accuracy:
    - ✦ Patch-centered atom coordinates
    - ✦ Increased precision of force interpolation
- Performance enhancements in NAMD 2.7b4 release (9/17/2010):
  - Sort blocks in order of decreasing work
  - Recursive bisection within patch on 32-atom boundaries
  - Warp-based pair lists based on sorted atoms

Sorting Blocks

- Sort patch pairs by increasing distance
- Equivalent to sort by decreasing work
- Slower blocks start first, fast blocks last
- Reduces idle time, total runtime of grid

Sorting Atoms

- Reduce warp divergence on cutoff tests
- Group nearby atoms in the same warp
- One option is space-filling curve
- Used recursive bisection instead
  - Split only on 32-atom boundaries
  - Find major axis, sort, split, repeat…

Warp-based Pairlists

- List generation
  - Load 16 atoms into shared memory
  - Any atoms in this warp within pairlist distance?
  - Combine all (4) warps as bits in char and save
- List use
  - Load set of 16 atoms if any bit is set in list
  - Only calculate if this warp's bit is set
  - Cuts kernel runtime by 50%

Multilevel Summation Method

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

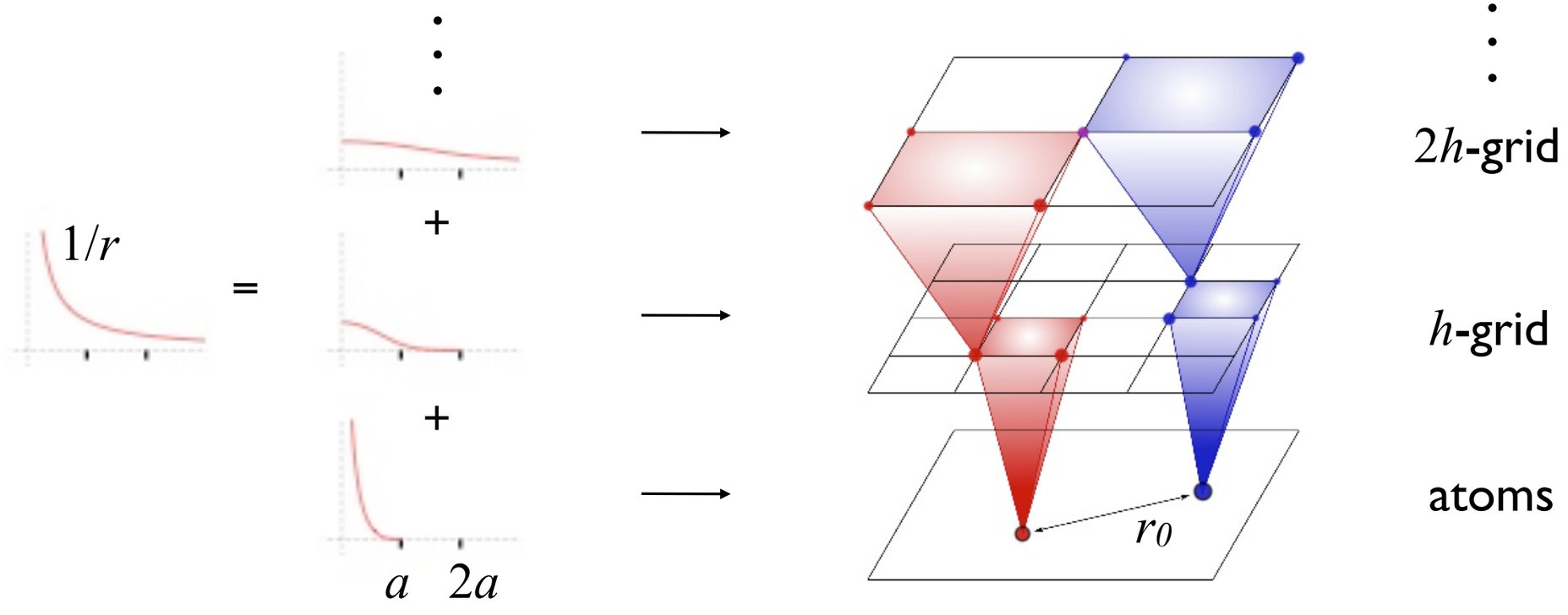
- Fast algorithm for N-body electrostatics
- Calculates sum of smoothed pairwise potentials interpolated from a hierarchal nesting of grids
- Advantages over PME (particle-mesh Ewald) and/or FMM (fast multipole method):
  - Algorithm has linear time complexity
  - Allows non-periodic or periodic boundaries
  - 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

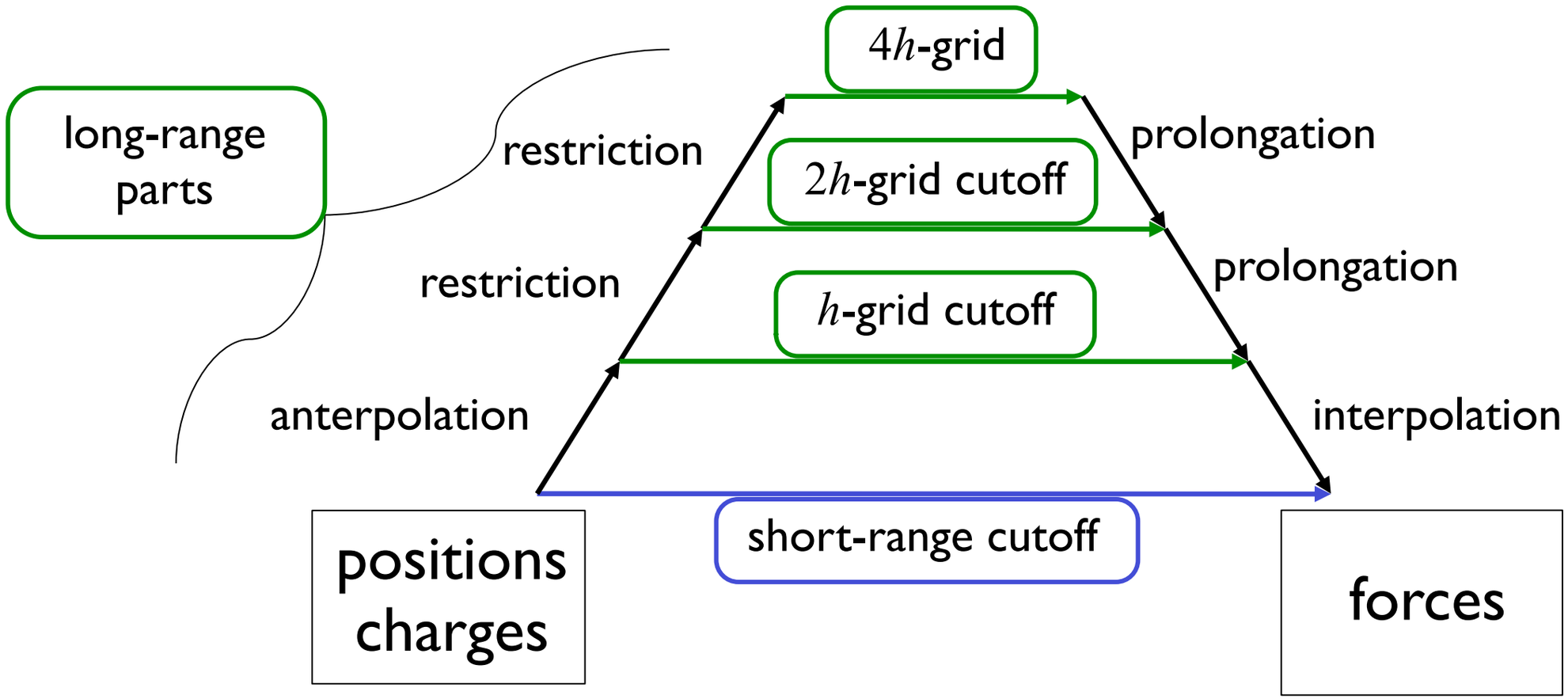
Interpolate the smoothed potentials



MSM Calculation

force	=	exact short-range part	+	interpolated long-range part

Computational Steps

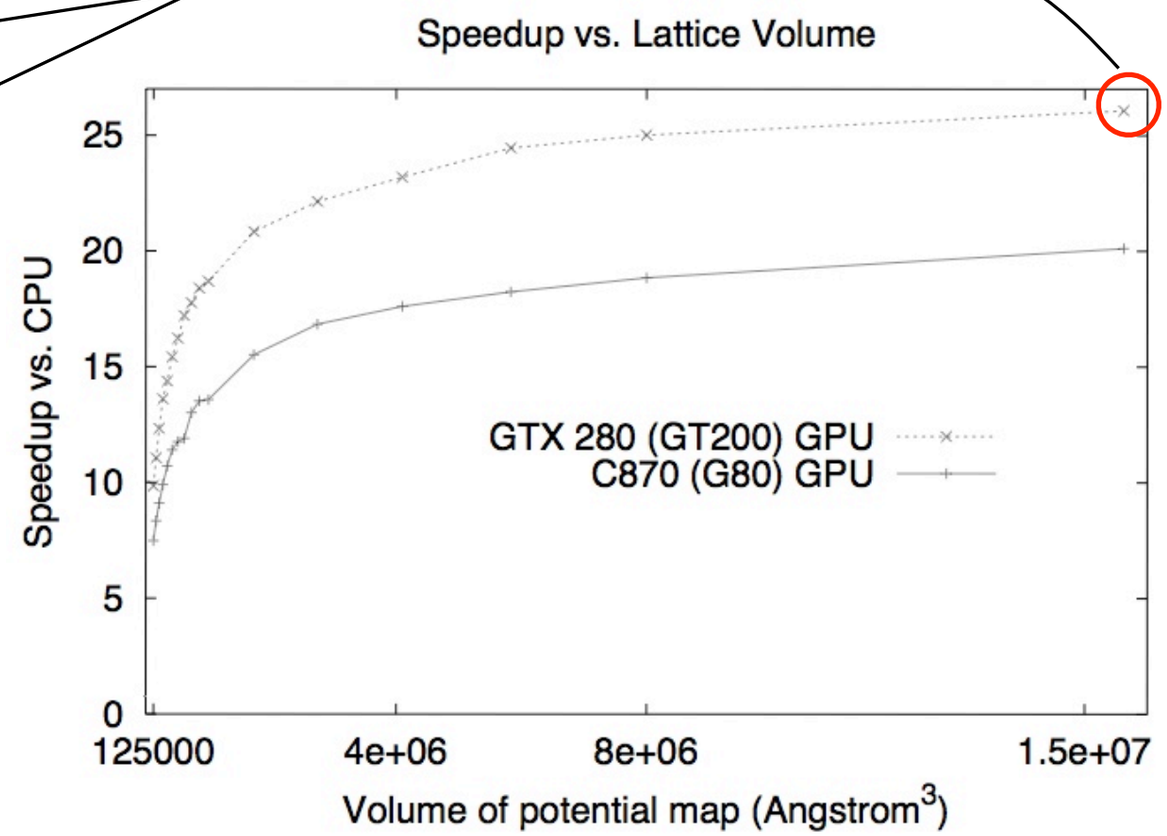


Multilevel Summation on the GPU (for electrostatic potential maps)

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.

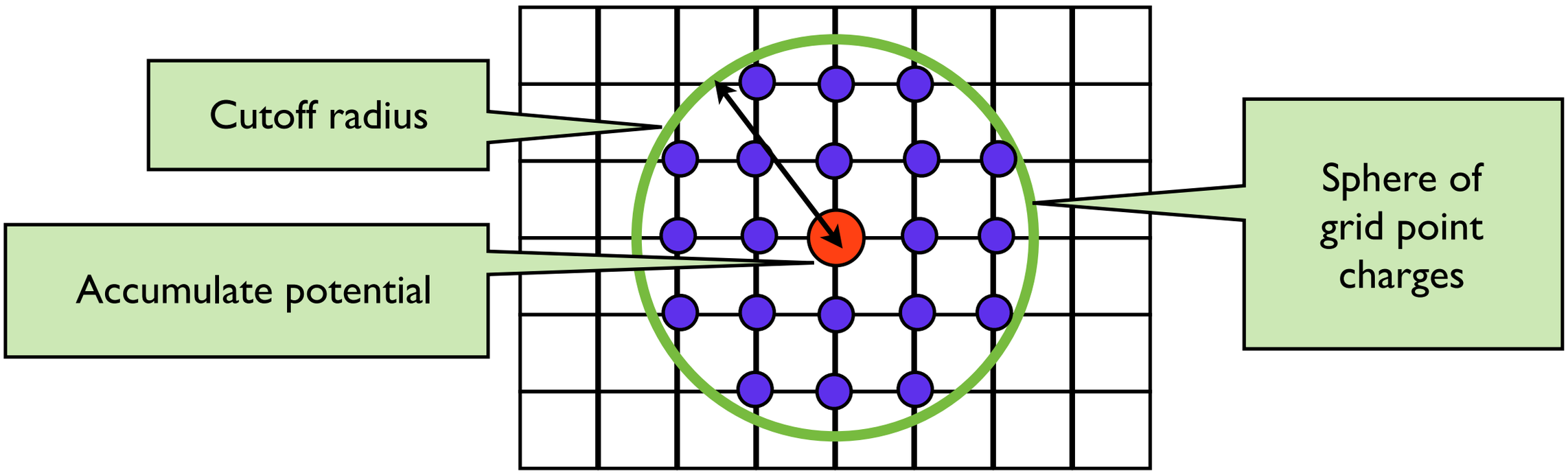
Computational steps	CPU (s)	w/ GPU (s)	Speedup
Short-range cutoff	480.07	14.87	32.3
Long-range anterpolation	0.18		
restriction	0.16		
lattice cutoff	49.47	1.36	36.4
prolongation	0.17		
interpolation	3.47		
Total	533.52	20.21	26.4



Multilevel summation of electrostatic potentials using graphics processing units.
D. Hardy, J. Stone, K. Schulten. *J. Parallel Computing*, 35:164-177, 2009.

MSM Grid Interactions

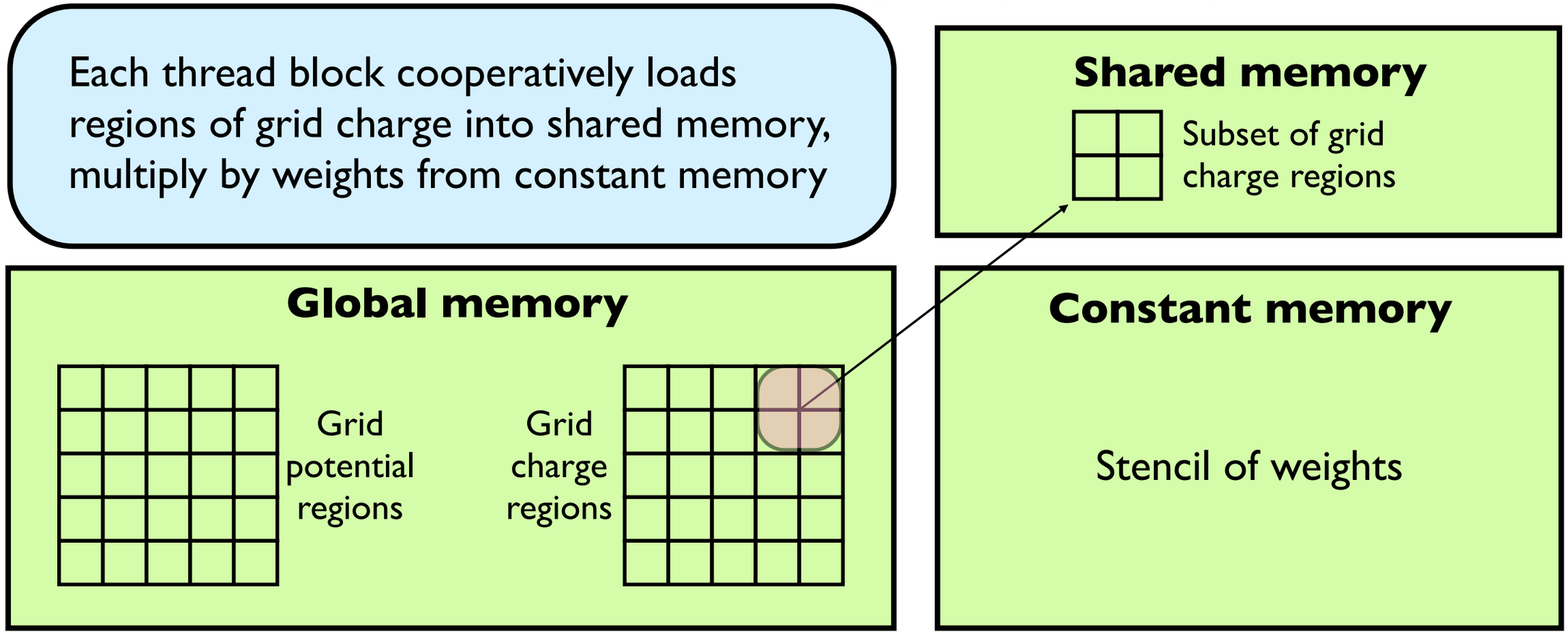
- Potential summed from grid point charges within cutoff
- Uniform spacing enables distance-based interactions to be precomputed as stencil of "weights"
- Weights at each level are identical up to scaling factor (!)
- Calculate grid potential as 3D convolution of weights with charges
  - stencil size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



MSM Grid Interactions on GPU

- Store weights in constant memory (padded up to next multiple of 4)
- Block of 64 threads calculates 4x4x4 ***region*** of potentials (stored contiguously)
- Pack all charge regions over all levels into 1D array (grid padded with zero charge)
- Store map of offsets to each level in constant memory
- Each thread block loops over surrounding charge regions (load into shared memory)
- Calculate all grid levels concurrently (avoid running out of work at upper grid levels)

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



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