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 6-8, 2010

Outline

- Introduction
- Short-range non-bonded forces
    - GPU kernel design considerations
    - NAMD GPU kernel
    - Kernel based on multilevel summation method
- Long-range electrostatics
    - Overview of multilevel summation method (MSM)
    - Kernel for MSM grid calculation (3D convolution)

Comparison with Particle-Grid Algorithms

- Particle-particle is more difficult 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

Loop unrolling (to reuse data)?

particle-grid	rx, ry, rz e1, ..., ek (positions, potentials)	3+k registers
particle-particle	r1x, r1y, r1z, ..., rkx, rky, rkz q1, ..., qk f1x, f1y, f1z, ..., fkx, fky, fkz u (positions, charges, forces, energy)	7k+1 registers

Particle-particle does not benefit from assigning multiple particles per thread

Non-bonded Potential Functions

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

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

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

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

Forces obtained from gradients of potential functions.

Outline

- Introduction
- Short-range non-bonded forces
  - GPU kernel design considerations
  - NAMD GPU kernel
  - Kernel based on multilevel summation method
- Long-range electrostatics
  - Overview of multilevel summation method (MSM)
  - Kernel for MSM grid calculation (3D convolution)

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

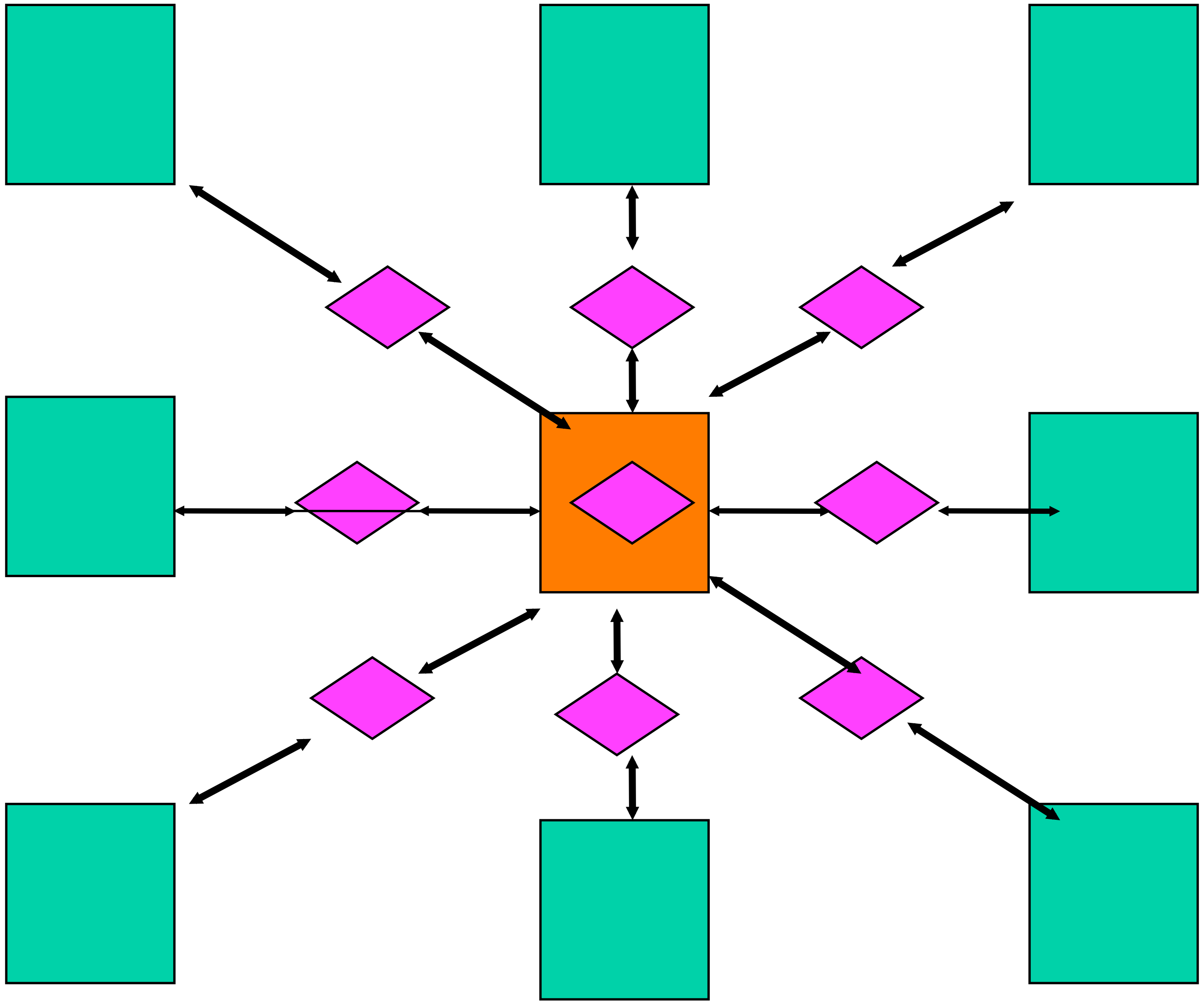
- How do we map work to the GPU threads?
  - Fine-grained: assign threads to sum forces on atoms
  - Extremely fine-grained: assign threads to pairwise interactions
- How do we decompose work into thread blocks?
  - Non-uniform: assign thread blocks to bins
  - Uniform: assign thread blocks to entries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

NAMD Hybrid Decomposition

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



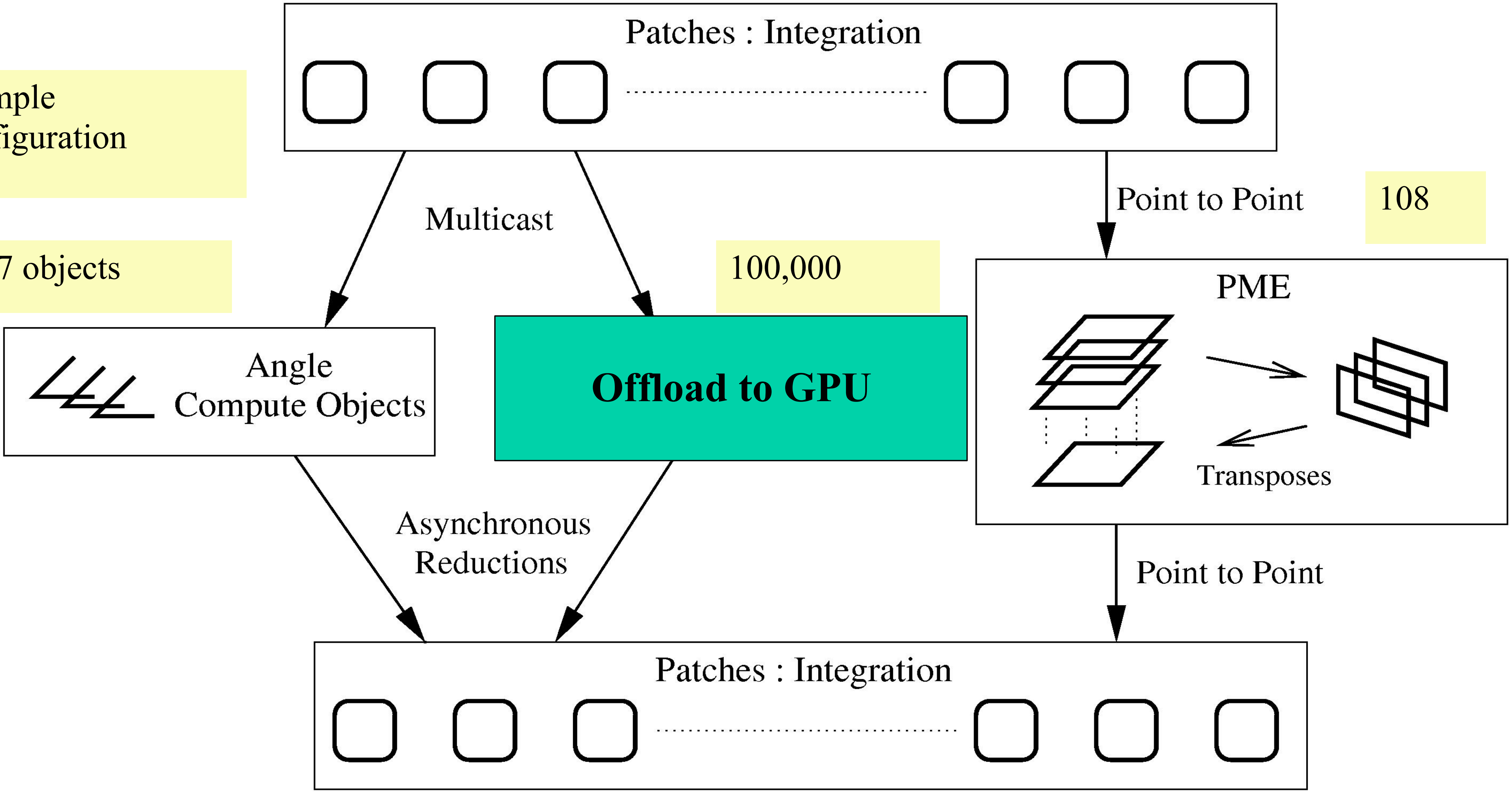
- Spatially decompose data and communication.
- Separate but related work decomposition.
- “Compute objects” facilitate iterative, measurement-based load balancing system.

NAMD Overlapping Execution

Phillips *et al.*, *SC2002*.

Example Configuration

847 objects



Patches : Integration

Multicast

Point to Point

108

100,000

Angle Compute Objects

Offload to GPU

PME

Transposes

Asynchronous Reductions

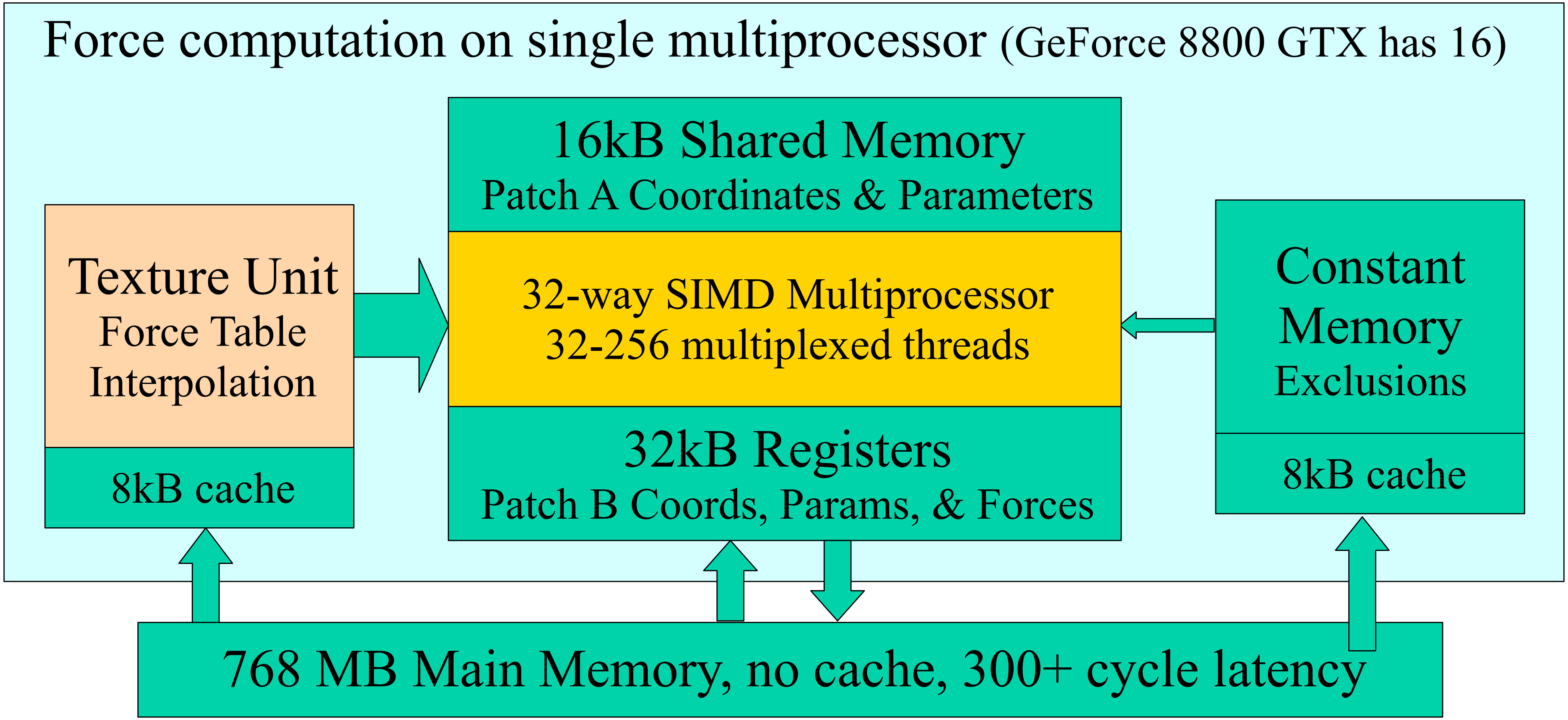
Point to Point

Patches : Integration

Objects are assigned to processors and queued as data arrives.

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
Constant
Memory
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.

Each Block Gets a Pair of Patches

- Block-level constants in shared memory to save registers.
- patch_pair array 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[8]; } 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
```

Loading Atoms Is Not Trivial

- Want to copy two 16-byte structs per thread from global to shared memory.
- Global memory access should be aligned on 16-byte boundaries to be coalesced.
- 16-byte structs in shared memory cause bank conflicts, 36-byte structs do not.

Right-Sized Atom Data Structures

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

struct __align__(16) atom_param {  // must be multiple of 16!
  float sqrt_epsilon;
  float half_sigma;
  unsigned int index;
  unsigned short excl_index;
  unsigned short excl_maxdiff;
};

struct shared_atom {  // do not align, size 36 to avoid bank conflicts
  float3 position;
  float charge;
  float sqrt_epsilon;
  float half_sigma;
  unsigned int index;
  unsigned int excl_index;
  unsigned int excl_maxdiff;
};
```

More Problems Loading Atoms

- Global access to mixed-type atom_param struct won't coalesce! (Only built-in vector types will.)
- Fix it by casting global atom_param* to uint4*.
- Can't take pointer to struct in registers, so copy integers to shared memory.
- Use alias of shared_atom and uint arrays to finally read patch B into usable struct in registers.
- Use same trick to load patch A, but this time leave the data in shared memory.

Hack to Coalesce atom_params

```
extern __shared__ shared_atom jas[];   // atom jas[max_atoms_per_patch]
extern __shared__ unsigned int sh_uint[];  // aliased to jas[]
atom ipq;
atom_param iap;

if ( threadIdx.x < myPatchPair.patch1_size ) {
   int i = myPatchPair.patch1_atom_start + threadIdx.x;
   uint4 tmpa = ((uint4*)atoms)[i];  // coalesced reads from global memory
   uint4 tmpap = ((uint4*)atom_params)[i];
   i = 9*threadIdx.x;
   sh_uint[i]   = tmpa.x;  // copy to aliased ints in shared memory
   sh_uint[i+1] = tmpa.y;
   sh_uint[i+2] = tmpa.z;
   sh_uint[i+3] = tmpa.w;
   sh_uint[i+4] = tmpap.x;
   sh_uint[i+5] = tmpap.y;
   sh_uint[i+6] = tmpap.z;
   sh_uint[i+7] = ((tmpap.w << 16) >> 16);  // split two shorts into shared_atom ints
   sh_uint[i+8] = (tmpap.w >> 16);
   COPY_ATOM(ipq, jas[threadIdx.x])  // macros to copy structs element by element
   COPY_PARAM(iap, jas[threadIdx.x])
}
```

CPU Force Interpolation

- Want to avoid calculating erfc(), sqrt(), branches for switching functions.
- $\mathbf{U}(\mathbf{r}^2) = \varepsilon(\sigma^{12}\mathbf{A}(\mathbf{r}^2) + \sigma^6\mathbf{B}(\mathbf{r}^2)) + \mathbf{qqC}(\mathbf{r}^2)$
- $F = \mathbf{-2}\ r\ \mathbf{U}'(\mathbf{r}^2)$
- Piecewise cubic interpolation of A,B,C.
- Need more windows at small $\mathbf{r}^2$, 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.
- $\mathbf{F(r^{-1})/r} = \varepsilon(\sigma^{12}\mathbf{A(r^{-1})} + \sigma^{6}\mathbf{B(r^{-1})}) + \mathbf{qqC(r^{-1})}$
- $F = r\ \mathbf{F(r^{-1})/r}$
- Piecewise linear interpolation of A,B,C.
  - F(r) is linear since $\mathbf{r\ (a\ r^{-1} + b) = a + r\ b}$
- Texture unit hardware is a perfect match.

Const 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.
- All exclusion data fits in constant cache.

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).

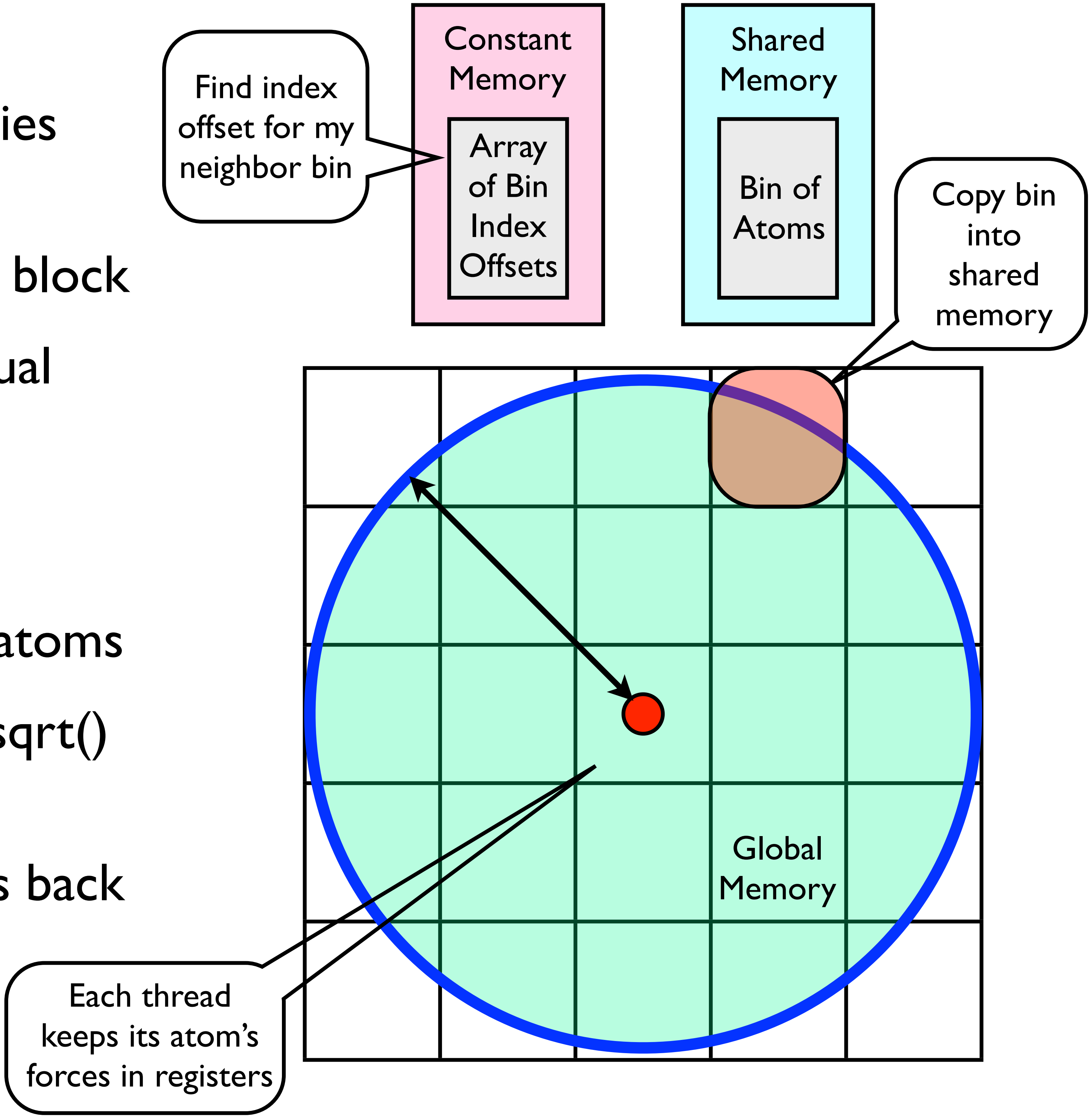
Nonbonded Forces CUDA Code

```
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                                                      Parameters
    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;
    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.

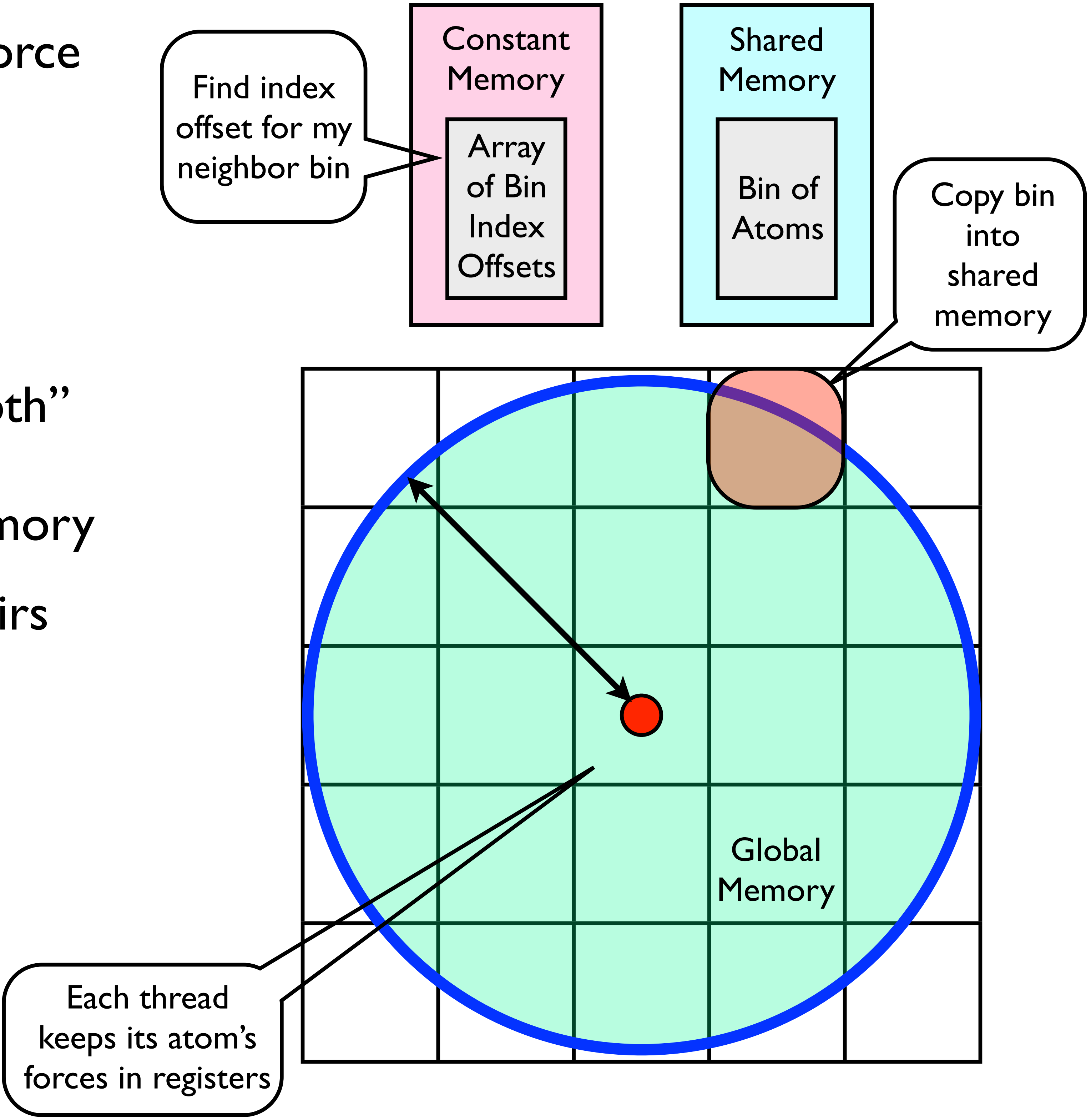
GPU Kernel for Short-range MSM

- 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

Atom Bin Storage

```
typedef struct BinSlot_t {
  int index;
  float rx, ry, rz;           /* position */
  float q, emin, rmin;    /* charge, sqrt(emin), (1/2)(rmin) */
  unsigned int excl;
} BinSlot;

union flint {
  float f;
  int i;
  unsigned int u;
};

density = 1.f/10;     /* 1 atom per 10 A^3 */
bindepth = 64;       /* thread block size */
binfill = 0.5;          /* bins are not allowed to be overfilled */

/* ideal bin volume, use to determine bin array dimensions */
binvolume = binfill * bindepth / density;
binlength = powf(binvolume, 1.f/3);  /* our ideal bin side length */
```

Loop Over Neighborhood of Bins

```
for (n = 0;  n < NbrhoodLen_C;  n++) {  /* loop over neighbor bins */
  int ib = Nbrhood_C[n].x;
  int jb = Nbrhood_C[n].y;
  int kb = Nbrhood_C[n].z;  /* these are offsets from my bin number */
  float cx = -ib * bx;
  float cy = -jb * by;
  float cz = -kb * bz;  /* (bin center) - (nbrbin center) */
  ib += i;
  jb += j;
  kb += k;  /* absolute indices of neighbor bin */
  /*** bin number adjusted for boundaries ***/

  __syncthreads();  /* read the next bin into the abin cache in shared memory */
  int bindex = (kb*nby + jb)*nbx + ib;
  int m;
  for (m = 0;  m < 8;  m++) {
    /* atom bin contains 8*bindepth flints */
    abinCache_S[m*bindepth + na] = abin_G[bindex*8*bindepth + m*bindepth + na];
  }
  __syncthreads();

  /*** loop over atoms in bin ***/

} /* end loop over neighborhood */
```

Inner Loop Over Atoms In Bin (1)

```
for (nb = 0;  nb < (8*bindepth);  nb += 8) {  // loop over nbr bin atoms
  // go through shared memory cache elements in order
  int bid = abinCache_S[nb].i;
  if (-1 == bid) break;  /* no more atoms in bin */
  float rx = rix - abinCache_S[nb+1].f + cx;
  float ry = riy - abinCache_S[nb+2].f + cy;
  float rz = riz - abinCache_S[nb+3].f + cz;
  float r2 = rx*rx + ry*ry + rz*rz;

  if (r2 < cutoff2) {  /* within cutoff */
    float qj = abinCache_S[nb+4].f;
    float eminj = abinCache_S[nb+5].f;
    float rminj = abinCache_S[nb+6].f;
    unsigned int exmask = abinCache_S[nb+7].u;
    int shift = aid - bid;
    if (shift < 0) {
      exmask = excli;
      shift = -shift;
    }
    int isexcl = (shift < 32 && (exmask & (1u<<(unsigned)shift)));

    /* exclusions also have to subtract off long-range elec part */
    float s = r2 * inv_cutoff2;
    float g = 1 + (s-1)*(-1.f/2 + (s-1)*(3.f/8)); /* Taylor 2 splitting */
    float dg = -1.f/2 + (s-1)*(3.f/4);

    float qq = qi * qj;
    float ue = qq * (-inv_cutoff * g);
    float due_r = qq * (-2*inv_cutoff2*inv_cutoff * dg);
    float uv = 0;
    float duv_r = 0;
```

Inner Loop Over Atoms In Bin (2)

```
    if ( ! isexcl ) {  /* not an exclusion, evaluate both elec and vdw */
      float inv_r = rsqrtf(r2);
      float inv_r2 = inv_r * inv_r;
      ue += qq * inv_r;
      due_r += qq * (-inv_r*inv_r2);
      float emin = emini * eminj;
      float rmin = rmini + rminj;
      float rmin_r2 = (rmin * rmin) * inv_r2;
      float rmin_r6 = rmin_r2 * rmin_r2 * rmin_r2;
      float rmin_r12 = rmin_r6 * rmin_r6;
      uv = emin * (rmin_r12 - 2 * rmin_r6);
      duv_r = -12 * emin * inv_r2 * (rmin_r12 - rmin_r6);

      if (r2 > swon2) {  /* switching function for vdw */
        float sw = (cutoff2 - r2) * (cutoff2 - r2) *
          (cutoff2 + 2*r2 - 3*swon2) * swdenom;
        float dsw_r = 12*(cutoff2 - r2) * (swon2 - r2) * swdenom;
        duv_r = uv * dsw_r + duv_r * sw;
        uv = uv * sw;
      }
    }

    fx += -rx * (due_r + duv_r);
    fy += -ry * (due_r + duv_r);
    fz += -rz * (due_r + duv_r);
    u_elec += 0.5f * ue;
    u_vdw += 0.5f * uv;
  } /* within cutoff */
} /* loop over nbr bin atoms */
```

Sum Reduction of Energies

```
/* sum reduce energies over thread block */
fbinCache_S[na] = u_elec;           // everybody writes their local values
fbinCache_S[bindepth + na] = u_vdw;
__syncthreads();
{
  int m;
  for (m = (bindepth >> 1);  m > 32;  m >>= 1) {  // sync threads across warps
    if (na < m) {
      fbinCache_S[na] += fbinCache_S[na + m];
      fbinCache_S[na+bindepth] += fbinCache_S[na+bindepth + m];
    }
    __syncthreads();
  }
}
if (na < 32) {  // no sync required within a warp
  fbinCache_S[na] += fbinCache_S[na + 32];
  fbinCache_S[na+bindepth] += fbinCache_S[na+bindepth + 32];

  fbinCache_S[na] += fbinCache_S[na + 16];
  fbinCache_S[na+bindepth] += fbinCache_S[na+bindepth + 16];

  fbinCache_S[na] += fbinCache_S[na + 8];
  fbinCache_S[na+bindepth] += fbinCache_S[na+bindepth + 8];

  fbinCache_S[na] += fbinCache_S[na + 4];
  fbinCache_S[na+bindepth] += fbinCache_S[na+bindepth + 4];

  fbinCache_S[na] += fbinCache_S[na + 2];
  fbinCache_S[na+bindepth] += fbinCache_S[na+bindepth + 2];

  fbinCache_S[na] += fbinCache_S[na + 1];
  fbinCache_S[na+bindepth] += fbinCache_S[na+bindepth + 1];
}
if (na < 1) {    fbinCache_S[3*bindepth+1] = fbinCache_S[bindepth];  // summed u_vdw
  fbinCache_S[3*bindepth  ] = fbinCache_S[0];         // summed u_elec
}
__syncthreads();  // sync here before writing local forces
```

Outline

- Introduction
- Short-range non-bonded forces
  - GPU kernel design considerations
  - NAMD GPU kernel
  - Kernel based on multilevel summation method
- Long-range electrostatics
  - Overview of multilevel summation method (MSM)
  - Kernel for MSM grid calculation (3D convolution)

Multilevel Summation Method

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

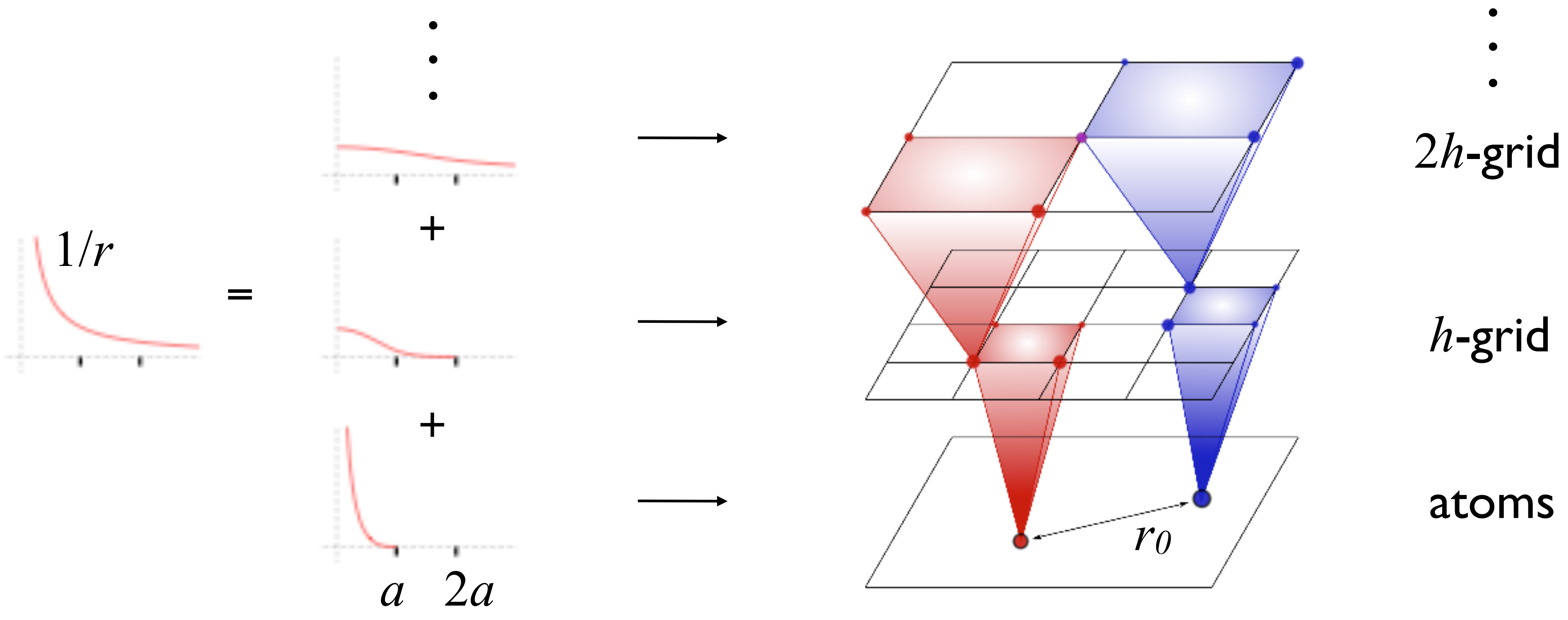
- Fast algorithm for N-body electrostatics
- Calculates sum of smoothed pairwise potentials interpolated from a hierarchal nesting of grids
- Advantages over PME (particle-mesh Ewald) and/or FMM (fast multipole method):
  - Algorithm has linear time complexity
  - Allows non-periodic or periodic boundaries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

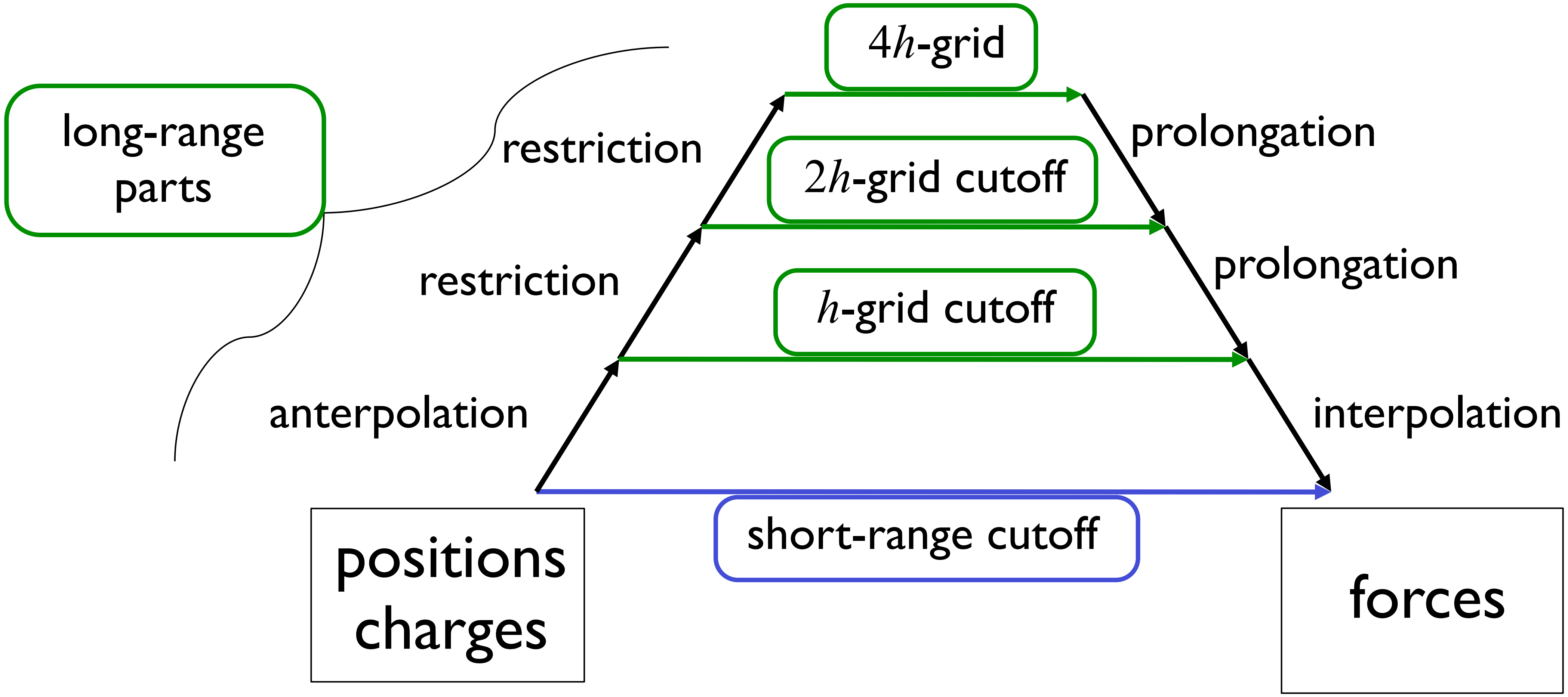
Interpolate the smoothed potentials



MSM Calculation

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

Computational Steps



long-range parts

restriction

restriction

anterpolation

4h-grid

2h-grid cutoff

h-grid cutoff

short-range cutoff

prolongation

prolongation

interpolation

positions charges

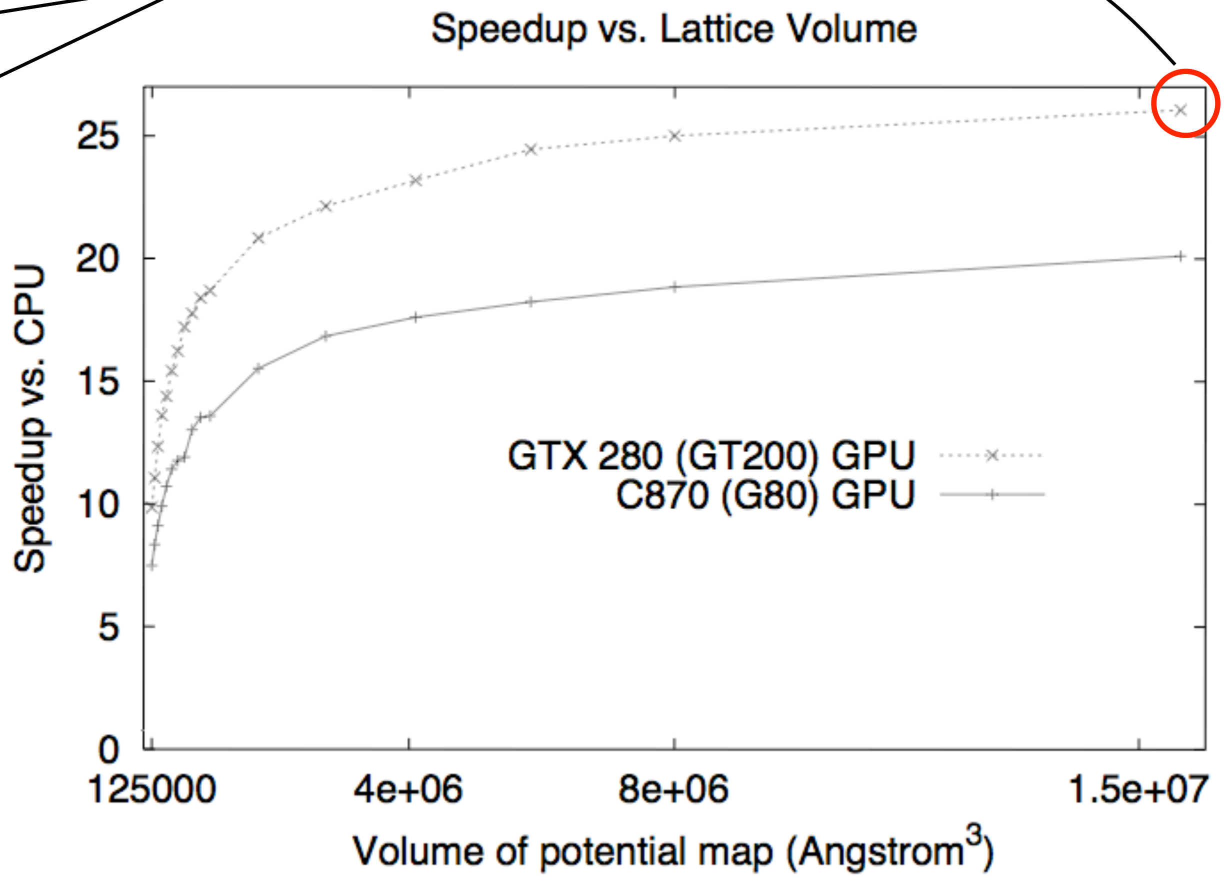
forces

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.

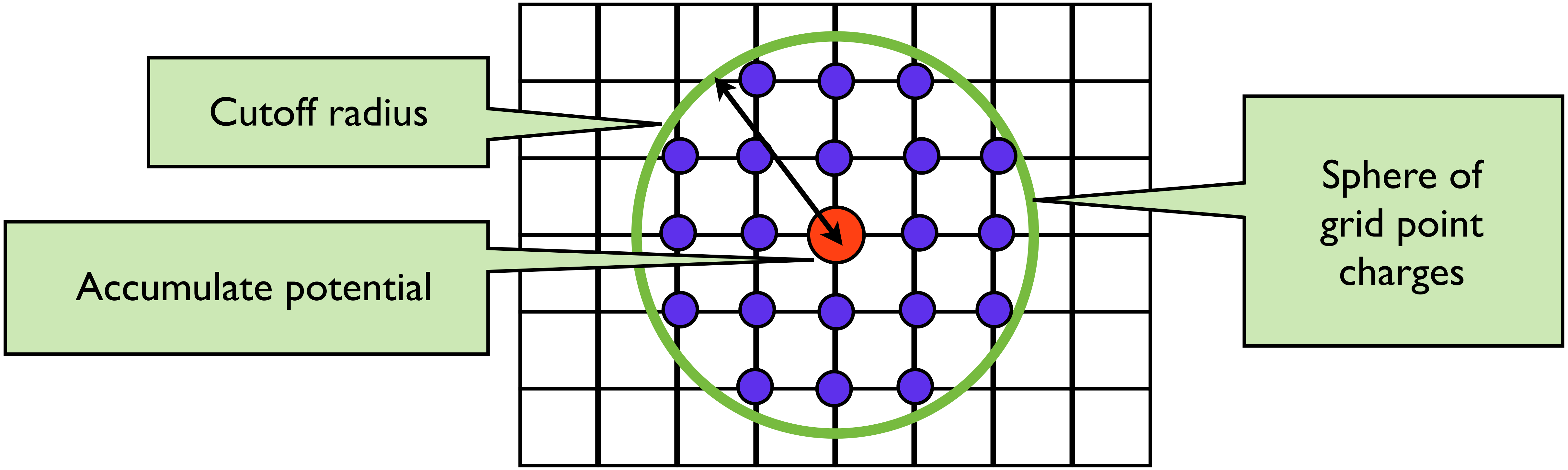
Computational steps	CPU (s)	w/ GPU (s)	Speedup
Short-range cutoff	480.07	14.87	32.3
Long-range anterpolation	0.18	1.36	36.4
restriction	0.16		
lattice cutoff	49.47		
prolongation	0.17		
interpolation	3.47		
Total	533.52	20.21	26.4



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

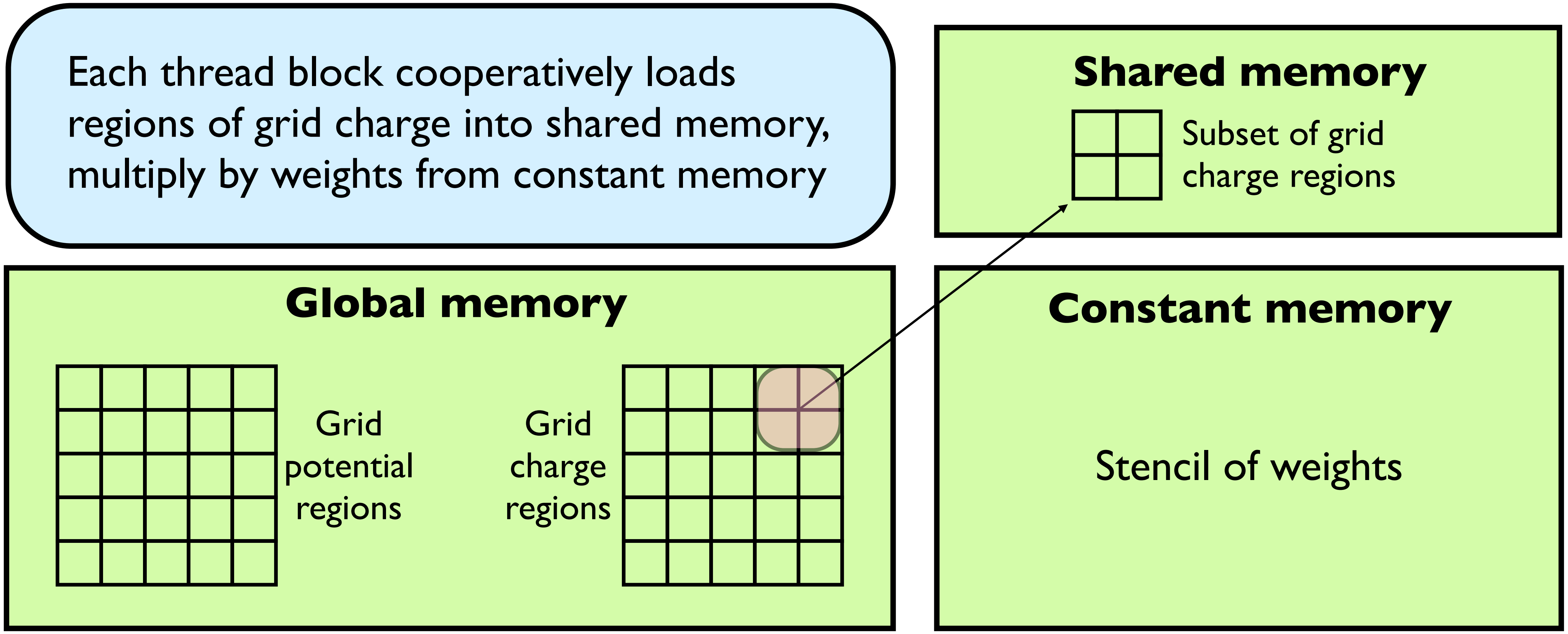
- Potential summed from grid point charges within cutoff
- Uniform spacing enables distance-based interactions to be precomputed as stencil of "weights"
- Weights at each level are identical up to scaling factor (!)
- Calculate as 3D convolution of weights
  - stencil size up to 23x23x23



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)

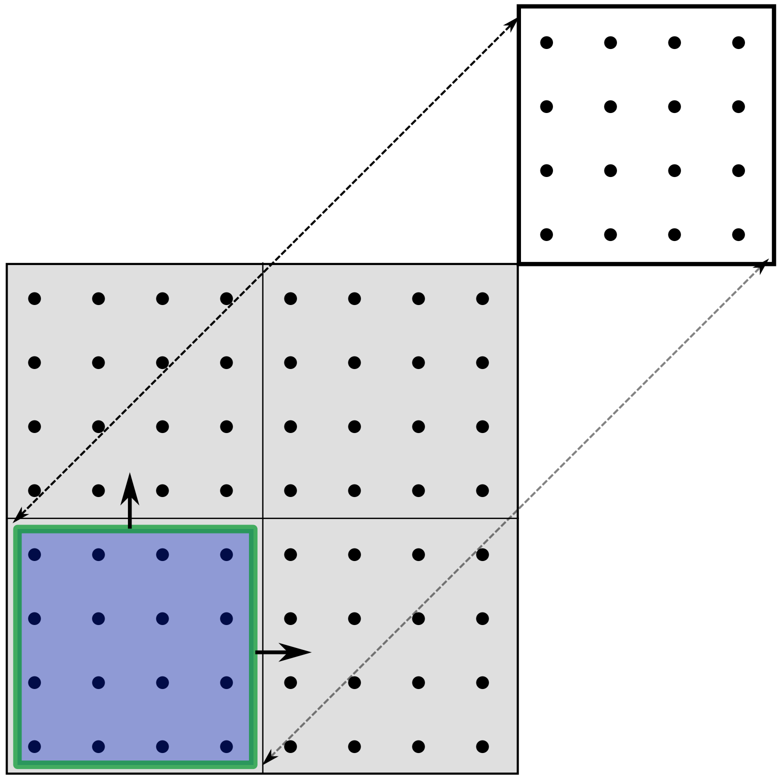
Hardy, *et al.*, *J. Paral. Comp.* **35**:164-177, 2009.



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 of GPU-MSM for MD

(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)