GPU Particle-Grid Methods: Electrostatics

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/

Workshop on GPU Programming for Molecular Modeling , Beckman Institute for Advanced Science and Technology, University of Illinois at Urbana-Champaign, August 7, 2010

Multiple Debye-Hückel Electrostatics

- Part of Poisson-Boltzmann solver in the popular APBS package
- Method: compute electrostatic potentials at grid points on boundary faces of box containing molecule
- Screening function:

$$S(r) = \frac{e^{-\kappa(r-\sigma_j)}}{1+\kappa\sigma_j}$$

MDH Kernel (CUDA)

```
extern shared f loat smem [ ] ;
int igrid = (blockIdx .x  blockDim.x ) + threadIdx .x ;      int lsize = blockDim.x ;    int lid= threadIdx .x ;
float lgx = gx [ igrid ] ;  float lgy = gy [ igrid ] ;  float lg z = gz [ igrid ] ;  float v = 0.0 f ;
for ( int jatom = 0 ; jatom < natoms ; jatom+=lsize ) {
   syncthreads ( ) ;
   i f ( ( jatom + l i d ) < natoms ) {
      smem[ lid                ] = ax [ jatom + lid] ;
      smem[ lid +       lsize ] = ay [ jatom + lid] ;
      smem[ lid + 2 * lsize ] = az [ jatom + lid] ;
      smem[ lid + 3 * lsize ] = charge [ jatom + lid] ;
      smem[ lid + 4 * lsize ] = size [ jatom + lid] ;
   }
   syncthreads ( ) ;
   i f ( ( jatom+l s i z e ) > natoms ) l s i z e = natoms − jatom ;
      for ( int i =0; i<l s i z e ; i++) {
         f loat dx = lgx − smem[ i              ] ;
         f loat dy = lgy − smem[ i +      lsize ] ;
         f loat dz = lgz − smem[ i + 2 * lsize ] ;
         f loat dist = sqrtf ( dxdx + dydy + dzdz ) ;
         v += smem[i+3*lsize] * expf(−xkappa ( dist − smem[ i+4*lsize ] ) ) / (1.0 f + xkappa  smem[ i+4*lsize ]) * dist) ;
      }
   }
   val [ igrid ] = pre1 * v;
```

Collectively load atoms from global memory into shared memory

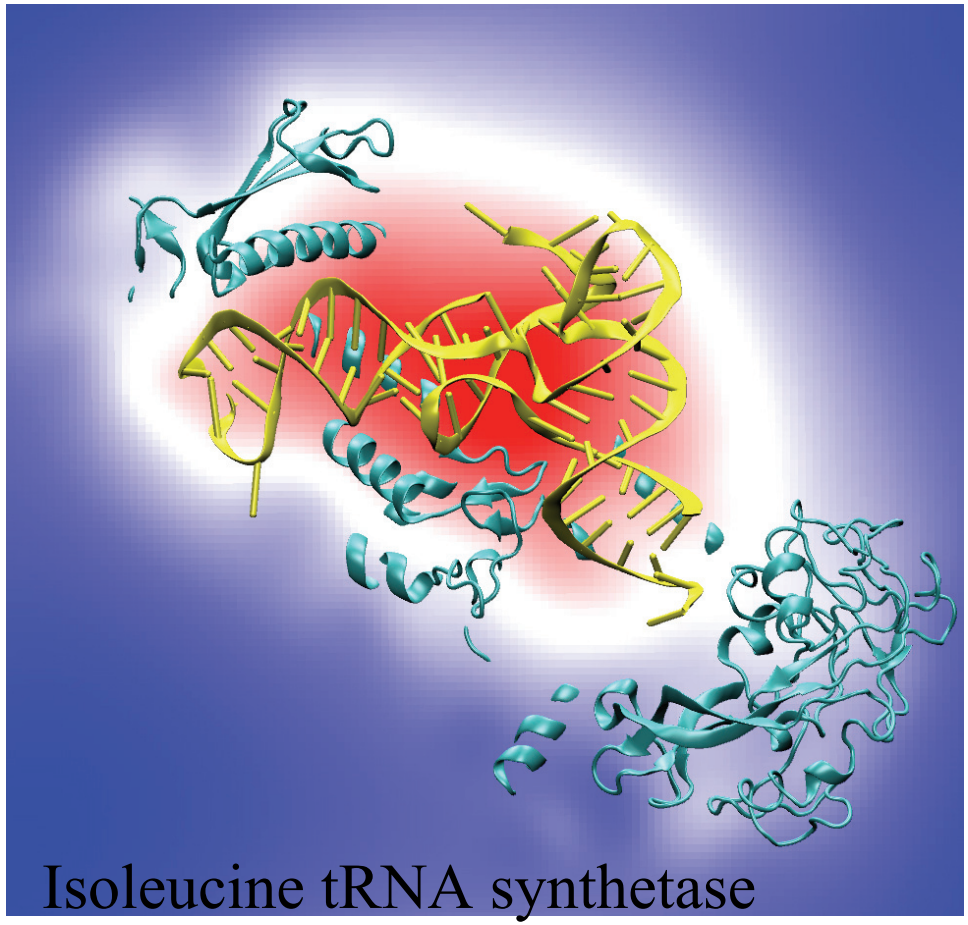
Loop over all all atoms in shared memory accumulating potential contributions into grid points

Electrostatic Potential Maps

- Electrostatic potentials evaluated on 3-D lattice:

$$V_i = \sum_j \frac{q_j}{4\pi\epsilon_0 |\mathbf{r}_j - \mathbf{r}_i|}$$

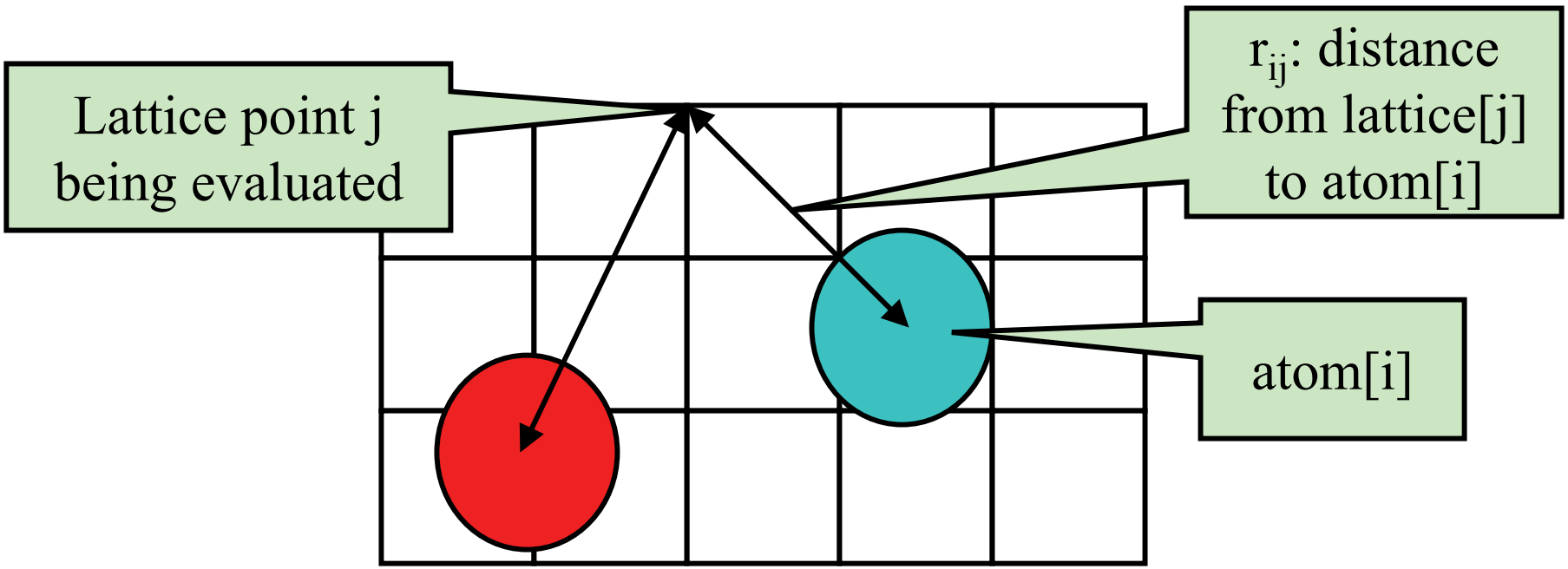
- Applications include:
  - Ion placement for structure building
  - Time-averaged potentials for simulation
  - Visualization and analysis



Direct Coulomb Summation (DCS) Algorithm Detail

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

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



DCS Computational Considerations

- Attributes of DCS algorithm for computing electrostatic maps:
  - Highly data parallel
  - Starting point for more sophisticated algorithms
  - Single-precision FP arithmetic is adequate for intended uses
  - Numerical accuracy can be further improved by compensated summation, spatially ordered summation groupings, or with the use of double-precision accumulation
  - Interesting test case since potential maps are useful for various visualization and analysis tasks
- Forms a template for related spatially evaluated function summation algorithms in CUDA

Single Slice DCS: Simple (Slow) C Version

```
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

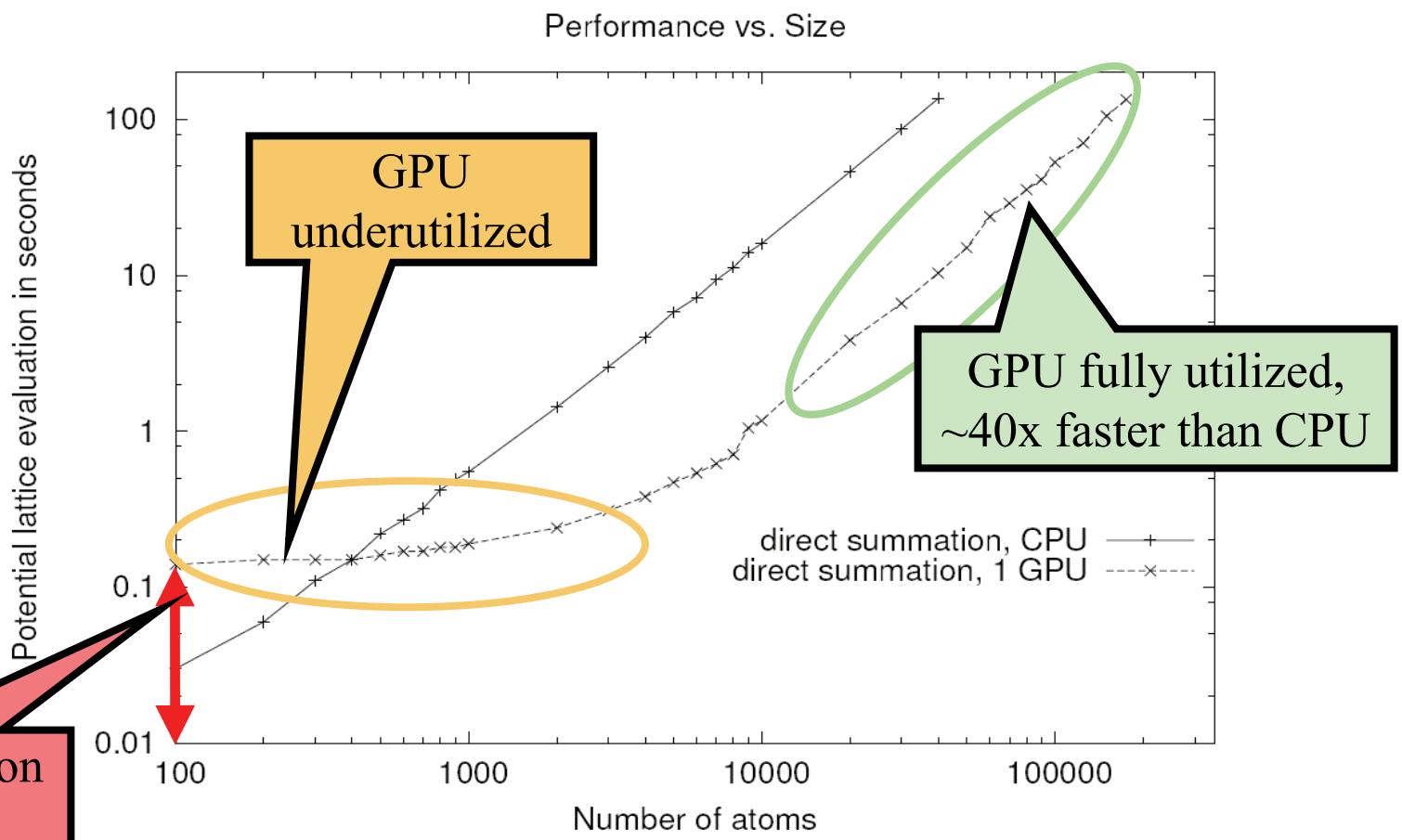
- Electrostatic maps used for ion placement require evaluation of ~20 potential lattice points per atom for a typical biological structure
- Atom list has the smallest memory footprint, best choice for the inner loop (both CPU and GPU)
- Lattice point coordinates are computed on-the-fly
- Atom coordinates 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

An Approach to Writing CUDA Kernels

- Find an algorithm that can expose substantial parallelism, we'll ultimately need thousands of independent threads…
- Identify appropriate GPU memory or texture subsystems used to store data used by kernel
- Are there trade-offs that can be made to exchange computation for more parallelism?
  - Though counterintuitive, past successes resulted from this strategy
  - "Brute force" methods that expose significant parallelism do surprisingly well on current GPUs
- Analyze the real-world use case for the problem and select the kernel for the problem sizes that will be heavily used

Direct Coulomb Summation Runtime

Lower is better

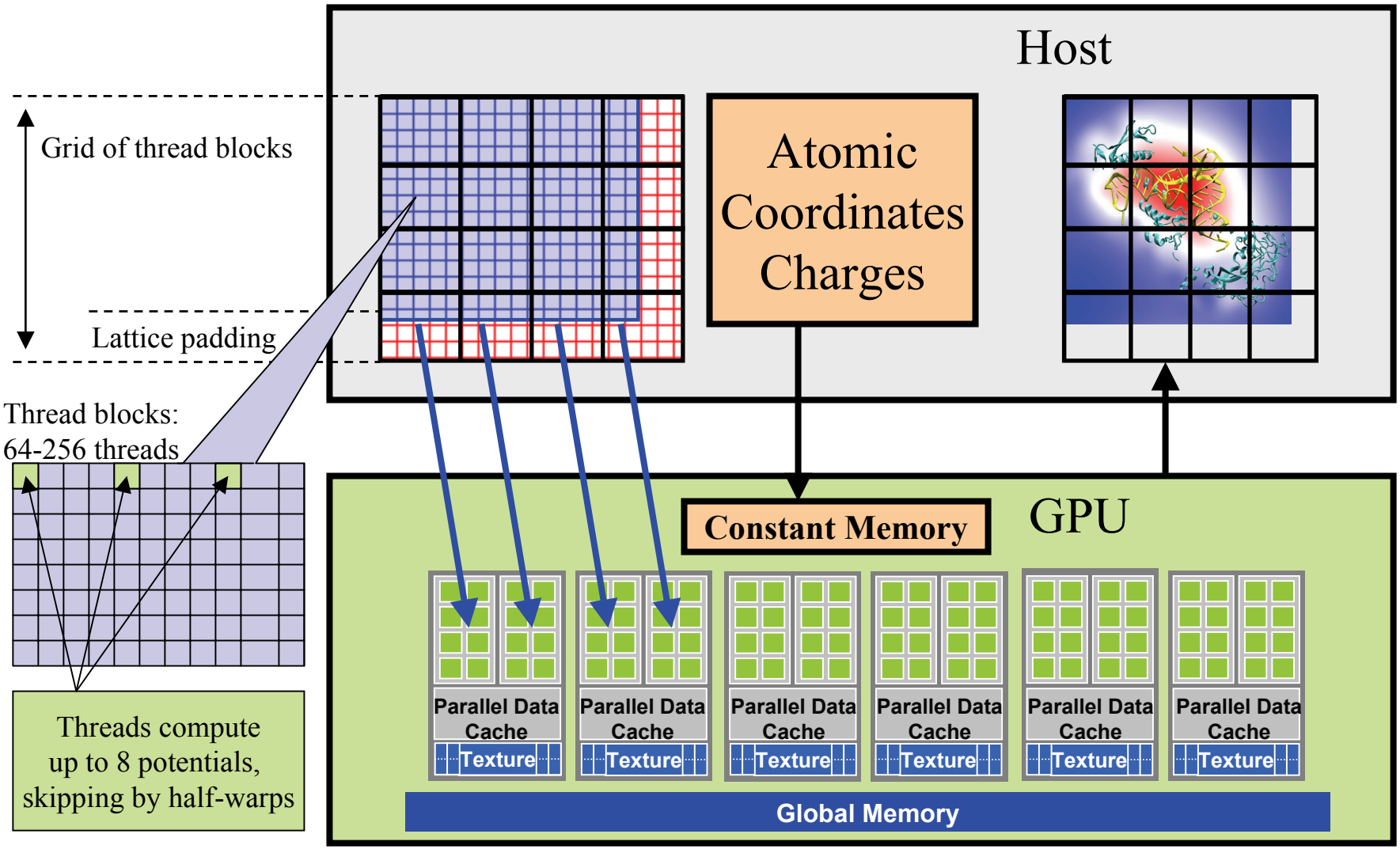


Accelerating molecular modeling applications with graphics processors. J. Stone, J. Phillips, P. Freddolino, D. Hardy, L. Trabuco, K. Schulten. *J. Comp. Chem.*, 28:2618-2640, 2007.

DCS Observations for GPU Implementation

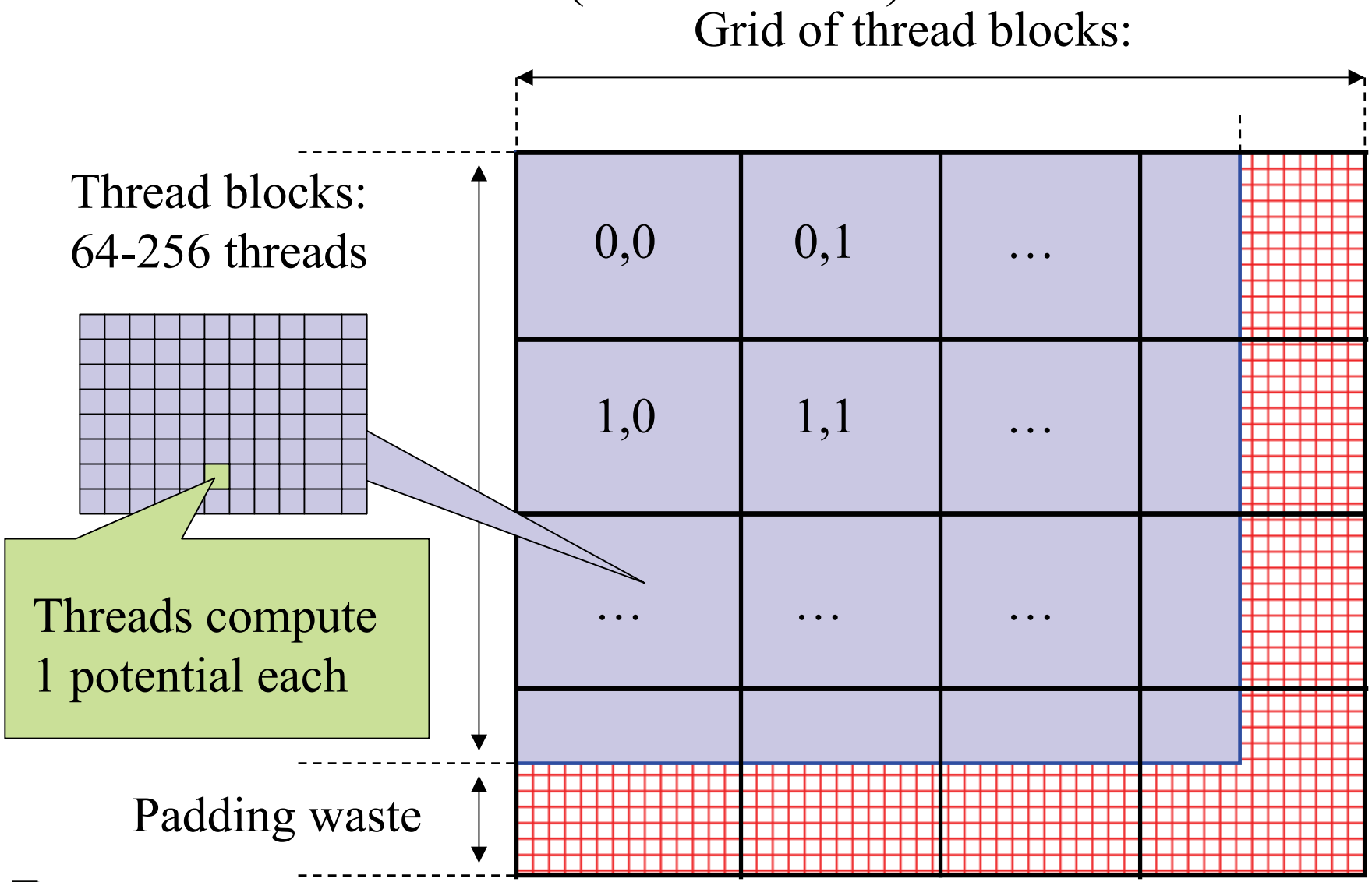
- Naive implementation has a low ratio of FP arithmetic operations to memory transactions (at least for a GPU…)
- The innermost loop will consume operands VERY quickly
- Since atoms are read-only, they are ideal candidates for texture memory or constant memory
- GPU implementations must access constant memory efficiently, avoid shared memory bank conflicts, coalesce global memory accesses, and overlap arithmetic 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

Direct Coulomb Summation on the GPU



DCS CUDA Block/Grid Decomposition

(non-unrolled)



DCS CUDA Block/Grid Decomposition (non-unrolled)

- 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 at the edges

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];
 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;
```

Start global memory reads early. Kernel hides some of its own latency.

Only dependency on global memory read is at the end of the kernel…

DCS CUDA Block/Grid Decomposition (unrolled, thread coarsening)

- Reuse atom data and partial distance components multiple times
- Use “unroll and jam” to unroll the outer loop into the inner loop
- Uses more registers, but increases arithmetic intensity significantly
- Kernels that unroll the inner loop calculate more than one lattice point per thread result in larger computational tiles:
  - Thread count per block must be decreased to reduce 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

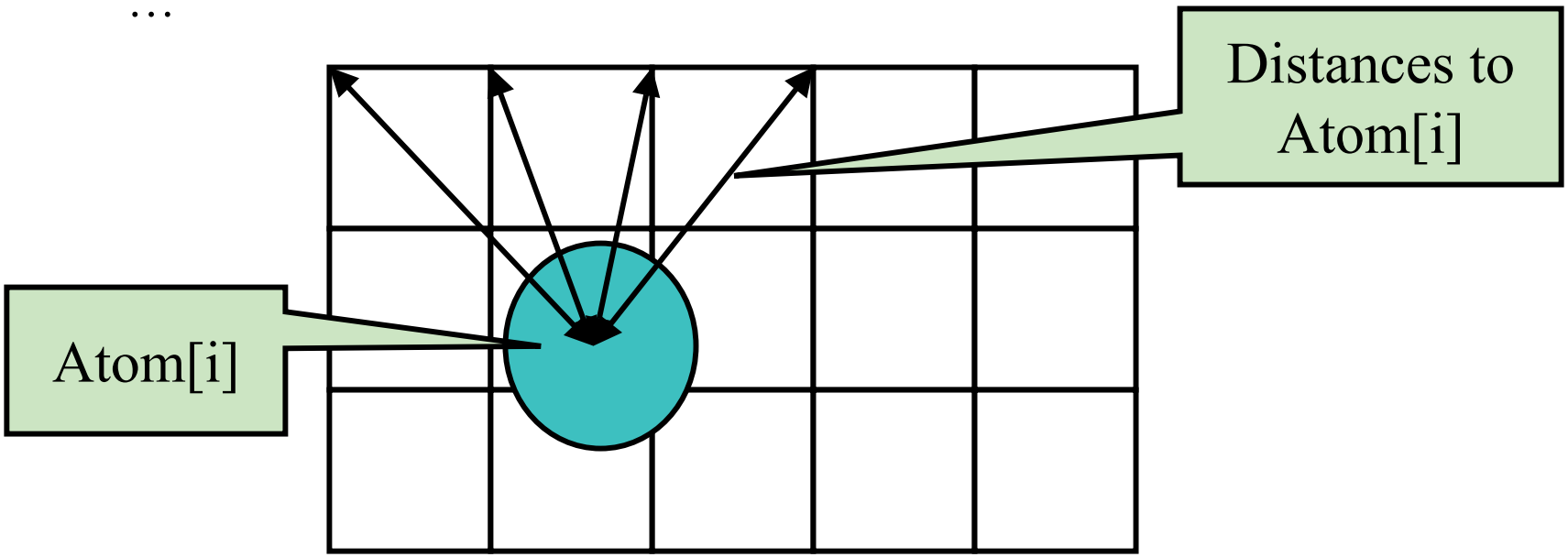
DCS CUDA Algorithm: Unrolling Loops

- Add each atom's contribution to several lattice points at a time, distances only differ in one component:

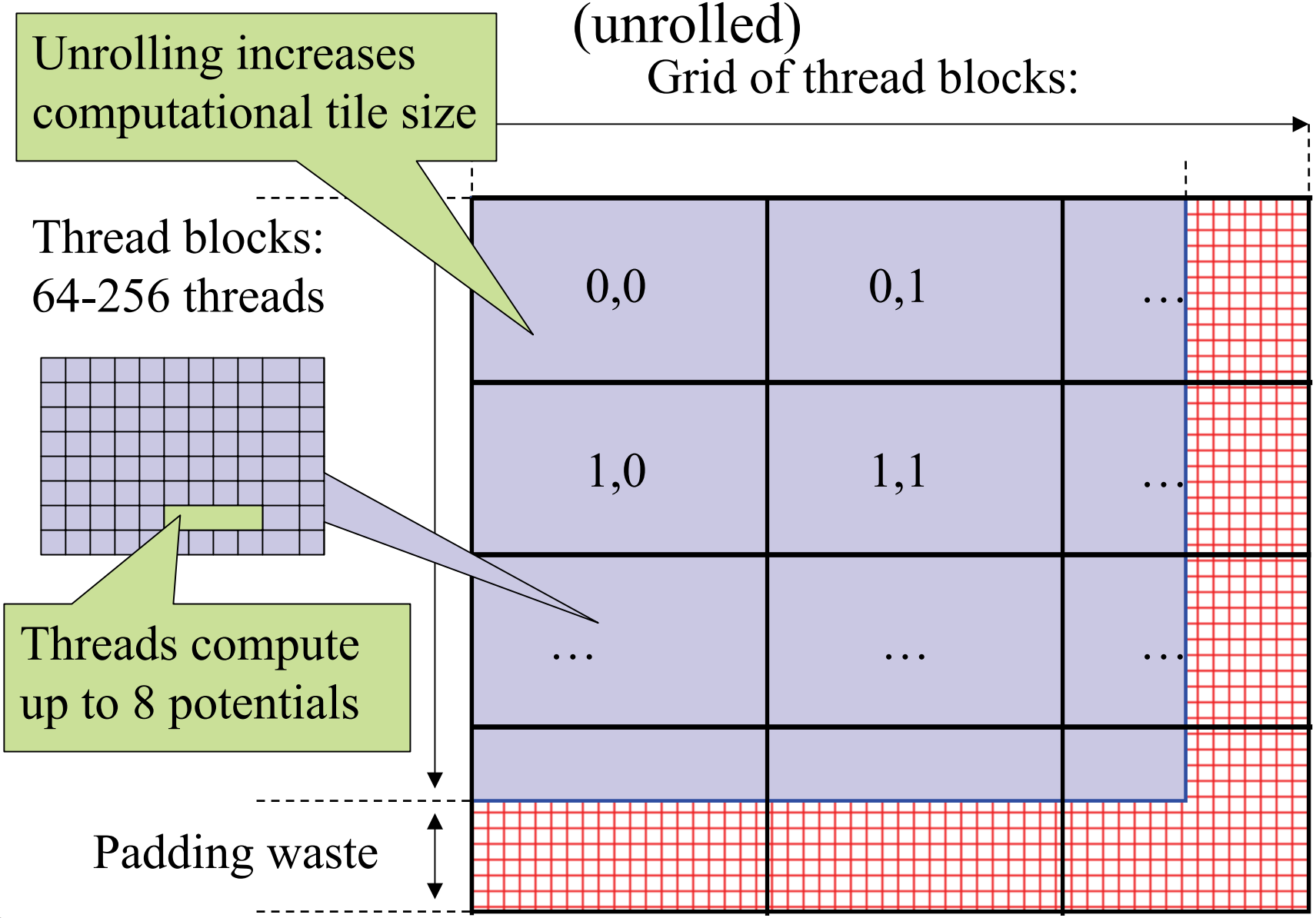
potential[j] += atom[i].charge / r_{ij}

potential[j+1] += atom[i].charge / $r_{i(j+1)}$

…

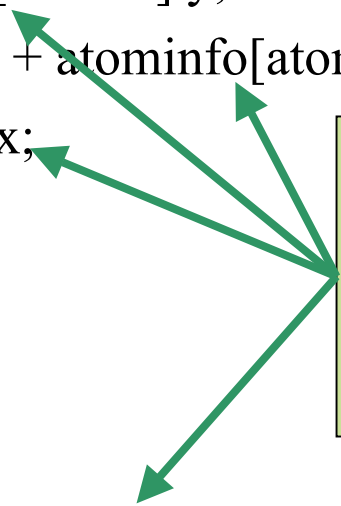


DCS CUDA Block/Grid Decomposition



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 x = atominfo[atomid].x;
    float dx1 = coorx1 - x;
    float dx2 = coorx2 - x;
    float dx3 = coorx3 - x;
    float dx4 = coorx4 - x;
    float charge = atominfo[atomid].w;
    energyvalx1 += charge * rsqrtf(dx1*dx1 + dysqpdzsq);
    energyvalx2 += charge * rsqrtf(dx2*dx2 + dysqpdzsq);
    energyvalx3 += charge * rsqrtf(dx3*dx3 + dysqpdzsq);
    energyvalx4 += charge * rsqrtf(dx4*dx4 + dysqpdzsq);
  }
```



Compared to non-unrolled kernel: memory loads are decreased by 4x, and FLOPS per evaluation are reduced, but register use is increased…

DCS Version 4: Const+Loop Unrolling+Coalescing 291.5 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 co-scheduling of 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 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
- Source code is available by request

DCS Version 4: Inner Loop

```
…float coory = gridspacing * yindex;
   float coorx = gridspacing * xindex;
   float gridspacing_coalesce = gridspacing * BLOCKSIZEX;
   int atomid;
   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 dx8 = dx7 + gridspacing_coalesce;
     energyvalx1 += atominfo[atomid].w * rsqrtf(dx1*dx1 + dyz2);
[…]
     energyvalx8 += atominfo[atomid].w * rsqrtf(dx8*dx8 + dyz2);
   }
  energygrid[outaddr                          ] += energyvalx1;
[...]
  energygrid[outaddr+7*BLOCKSIZEX] += energyvalx7;
```

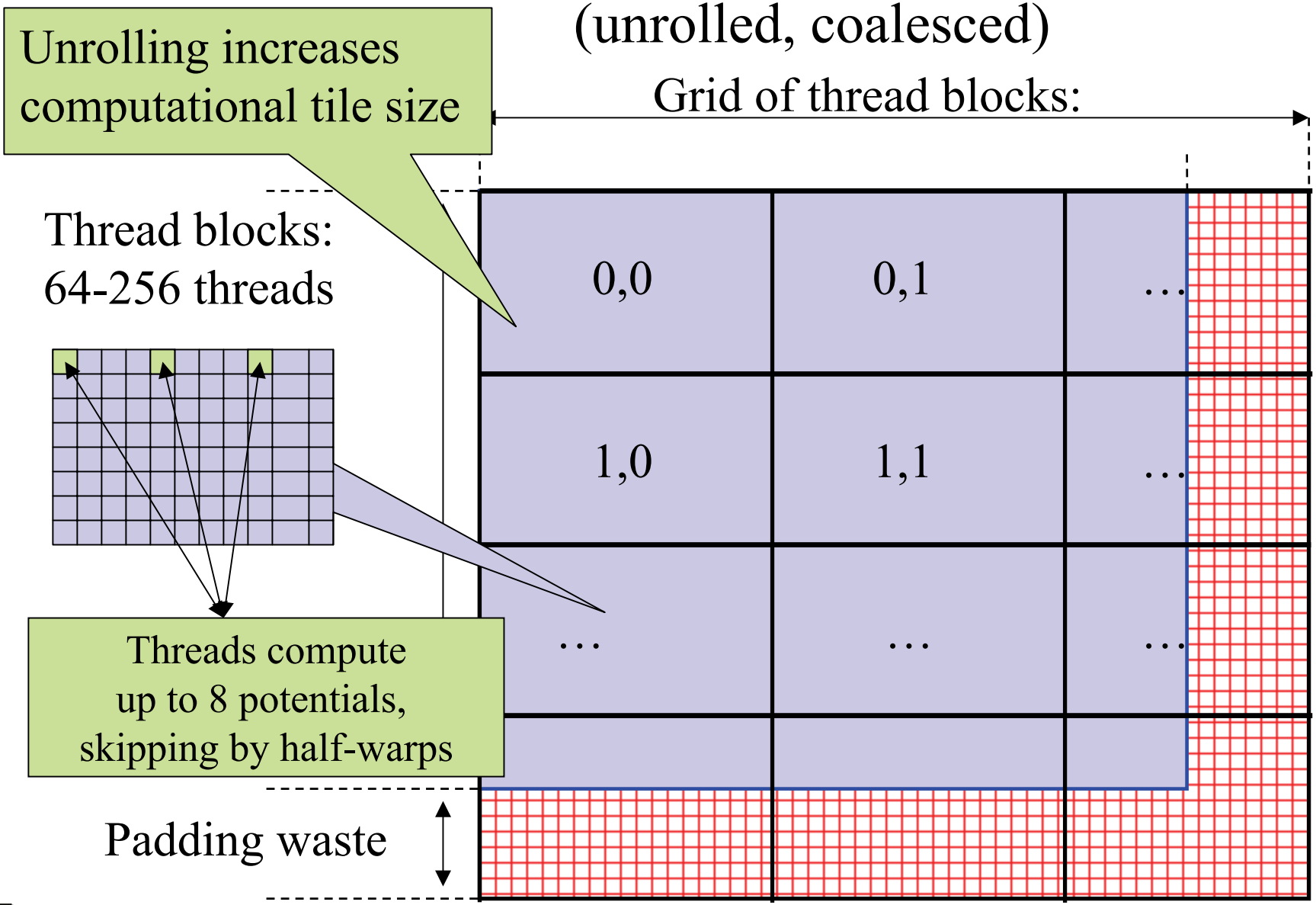
Points spaced for memory coalescing

Reuse partial distance components dy^2 + dz^2

Global memory ops occur only at the end of the kernel, decreases register use

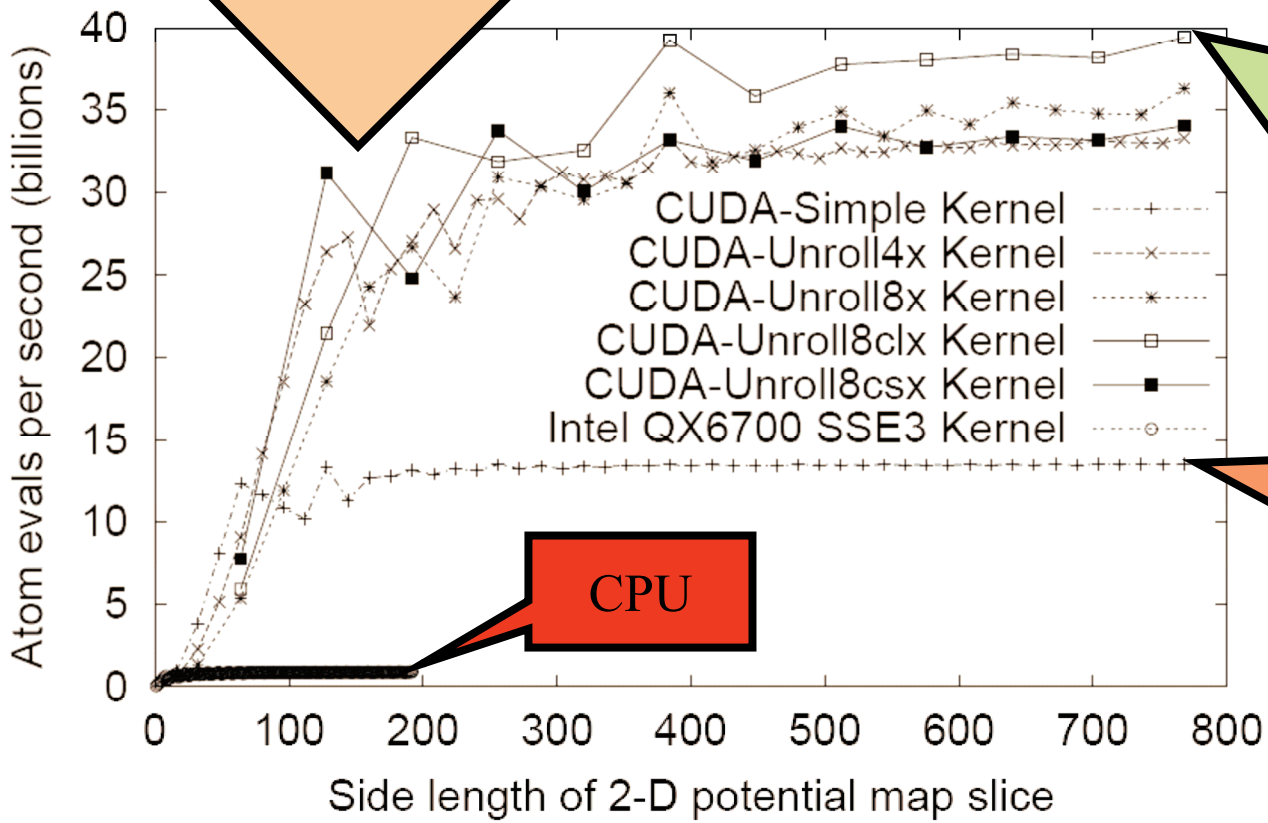
DCS CUDA Block/Grid Decomposition

(unrolled, coalesced)



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

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

DCS Version 4 Inner Loop, Scalar OpenCL

```
…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 charge = atominfo[atomid].w;
    energyvalx1 += charge * native_rsqrt(dx1*dx1 + dyz2);
    energyvalx2 += charge * native_rsqrt(dx2*dx2 + dyz2);
    energyvalx3 += charge * native_rsqrt(dx3*dx3 + dyz2);
    energyvalx4 += charge * native_rsqrt(dx4*dx4 + dyz2);
  }
```

Well-written CUDA code can often be easily ported to OpenCL if C++ features and pointer arithmetic aren't used in kernels.

DCS Version 4 Inner Loop (CUDA)

(only 4-way unrolling for conciseness to compare OpenCL)

```
…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 charge = atominfo[atomid].w;
  energyvalx1 += charge * rsqrtf(dx1*dx1 + dyz2);
  energyvalx2 += charge * rsqrtf(dx2*dx2 + dyz2);
  energyvalx3 += charge * rsqrtf(dx3*dx3 + dyz2);
  energyvalx4 += charge * rsqrtf(dx4*dx4 + dyz2);
}
```

DCS Version 4 Inner Loop, Vectorized OpenCL

```
 float4 gridspacing_u4 = { 0.f, 1.f, 2.f, 3.f };
 gridspacing_u4 *= gridspacing_coalesce;
 float4 energyvalx=0.0f;
…
 for (atomid=0; atomid<numatoms; atomid++)
    float dy = coory - atominfo[atomid].y
    float dyz2 = (dy * dy) + atominfo[atomid].z;
    float4 dx = gridspacing_u4 + (coorx – atominfo[atomid].x);
    float charge = atominfo[atomid].w;
    energyvalx1 += charge * native_rsqrt(dx1*dx1 + dyz2);
 }
```

CPUs, AMD GPUs, and Cell often perform better with vectorized kernels.
Use of vector types may increase register pressure; sometimes a delicate balance…

Acknowledgements

- Theoretical and Computational Biophysics Group, University of Illinois at Urbana-Champaign
- Wen-mei Hwu and the IMPACT group at University of Illinois at Urbana-Champaign
- NVIDIA CUDA Center of Excellence, University of Illinois at Urbana-Champaign
- NCSA Innovative Systems Lab
- The CUDA team at NVIDIA
- NIH support: P41-RR05969