Summer School

e-Science with Many-core CPU/GPU Processors

Lecture 10: Case Study 3

Electrostatic Potential Calculation using Direct Coulomb Summation

Acknowledgement

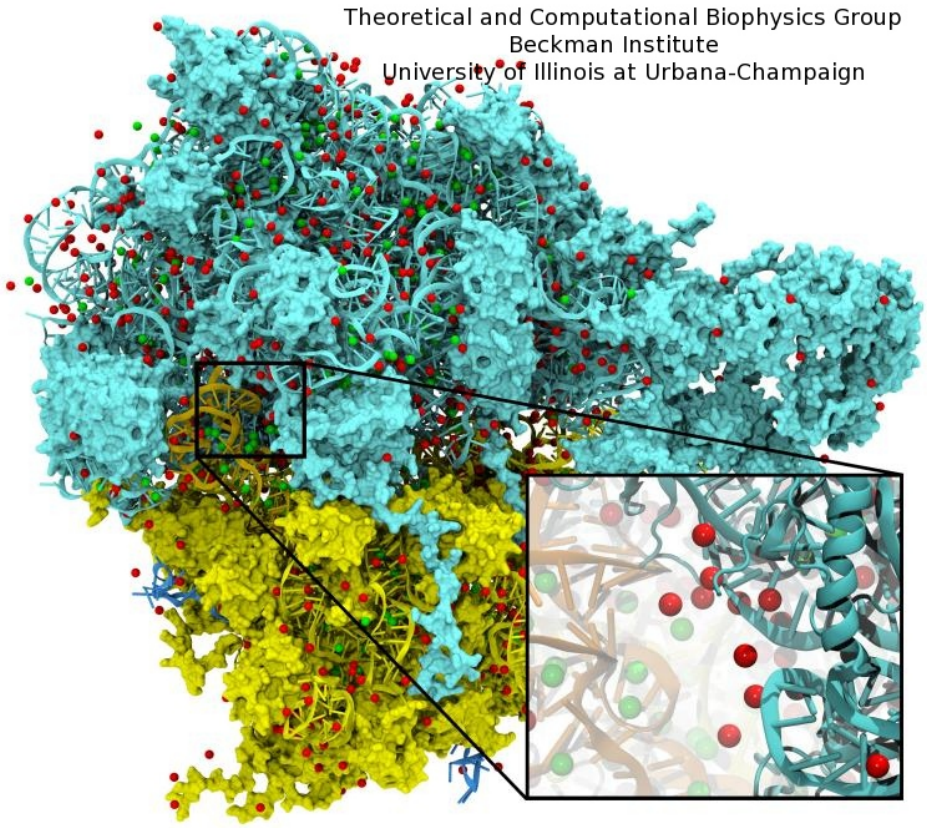
- Additional Information and References:
  - http://www.ks.uiuc.edu/Research/gpu/
  - http://www.ks.uiuc.edu/Research/vmd/
- Acknowledgement, questions, source code requests:
  - John Stone johns@ks.uiuc.edu
  - Klaus Schulten
  - Theoretical and Computational Biophysics Group, NIH Resource for Macromolecular Modeling and Bioinformatics Beckman Institute for Advanced Science and Technology
- NIH support: P41-RR05969

Outline

- 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

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



Overview of Ion Placement Process

- Calculate initial electrostatic potential map around the simulated structure considering the contributions of all atoms
- 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

Overview of Direct Coulomb Summation (DCS) Algorithm

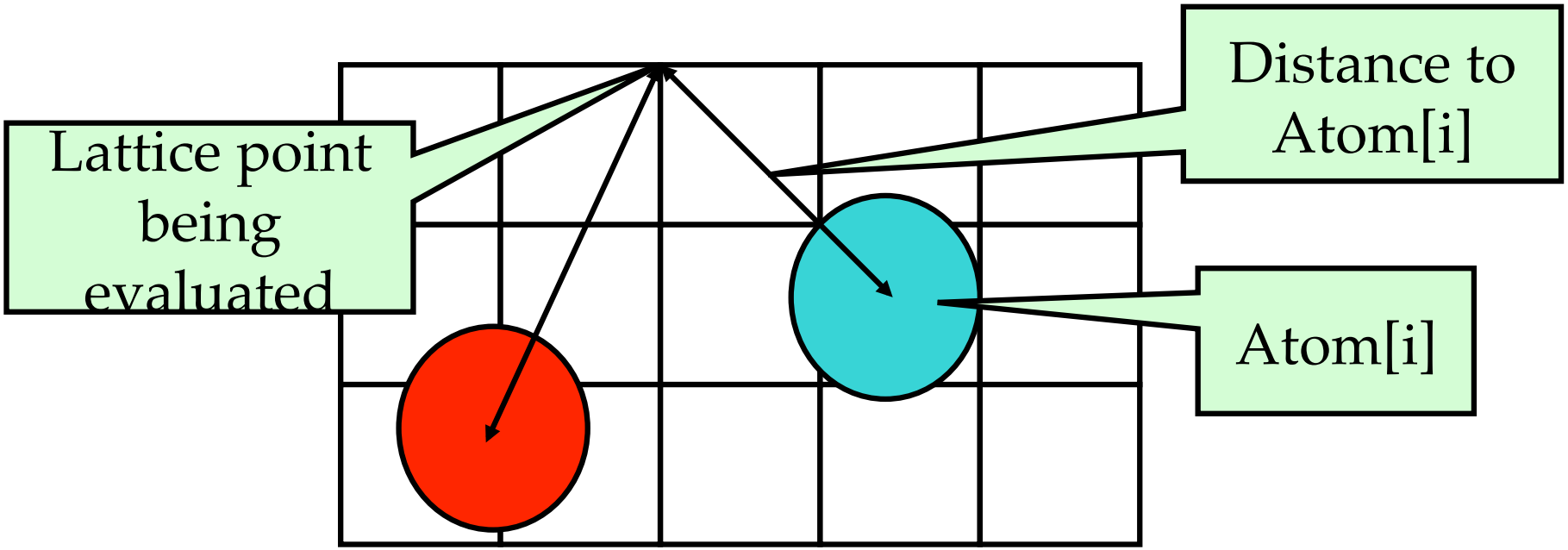
- One of several ways to compute the electrostatic potentials on a grid, ideally suited for the GPU
- Approximation-based methods such as multilevel summation can achieve much higher performance at the cost of some numerical accuracy and flexibility
- For today's talk, we'll only discuss DCS, as it is a conceptually simple algorithm that is easy to fully explore, and it requires very little background knowledge
- DCS: for each lattice point, sum potential contributions for all atoms in the simulated structure:

  potential += charge[i] / (distance to atom[i])

Direct Coulomb Summation (DCS) Algorithm Detail

- At each lattice point, sum potential contributions for all atoms in the simulated structure:

  potential += charge[i] / (distance to atom[i])



DCS Computational Considerations

- 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 ion placement 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, even for the CPU!)

```
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

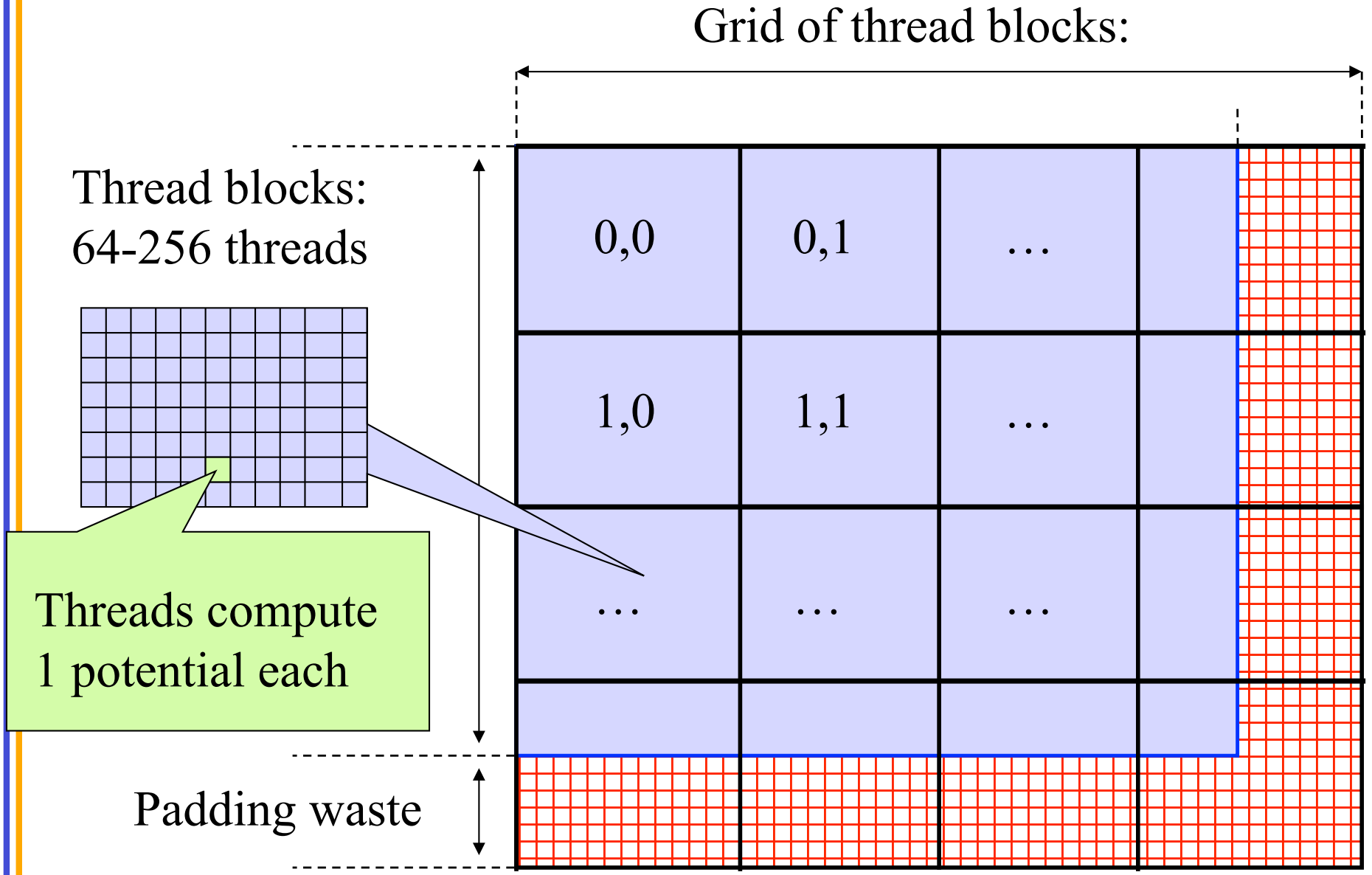
- Straightforward implementation has a low ratio of floating point arithmetic operations to memory transactions (at least for a GPU…)
- The innermost loop will consume operands VERY quickly
- Since atoms are read-only, they are ideal candidates for texture memory or constant memory
- GPU implementations 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

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];  // 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 *
                          (1.0f / sqrtf(dx*dx + dy*dy + atominfo[atomid].z));
  }
  energygrid[outaddr] = curenergy + energyval;
```

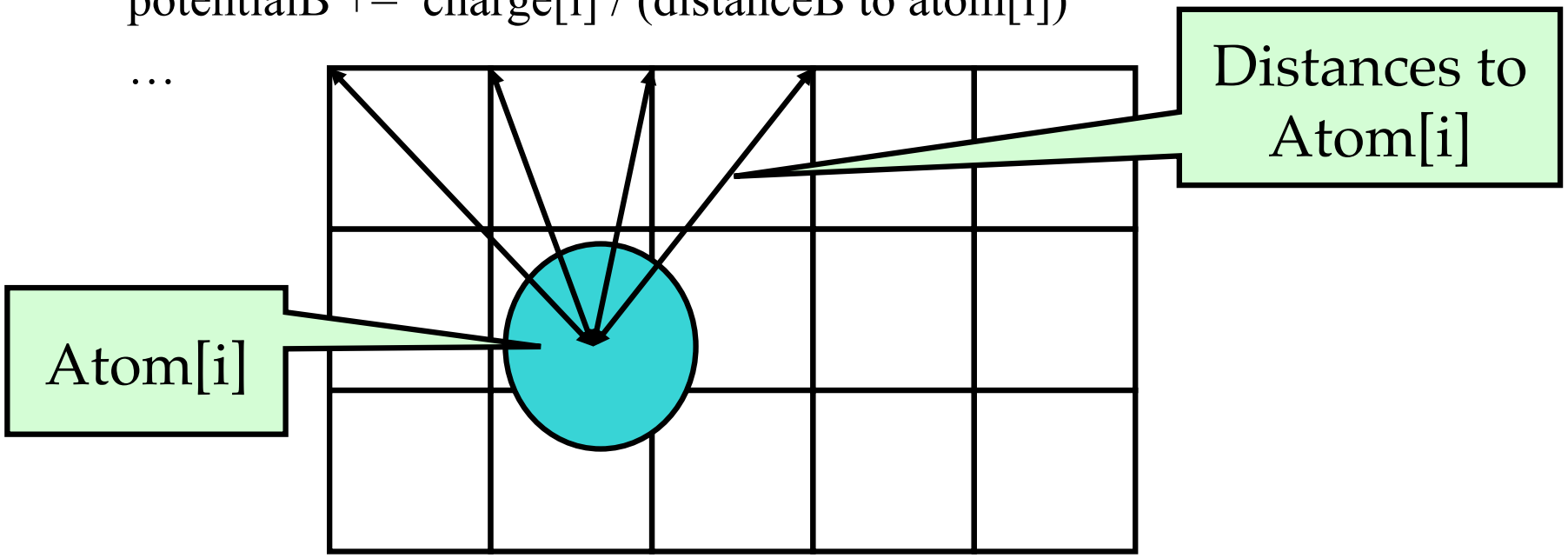
DCS CUDA Block/Grid Decomposition (unrolled)

- 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 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

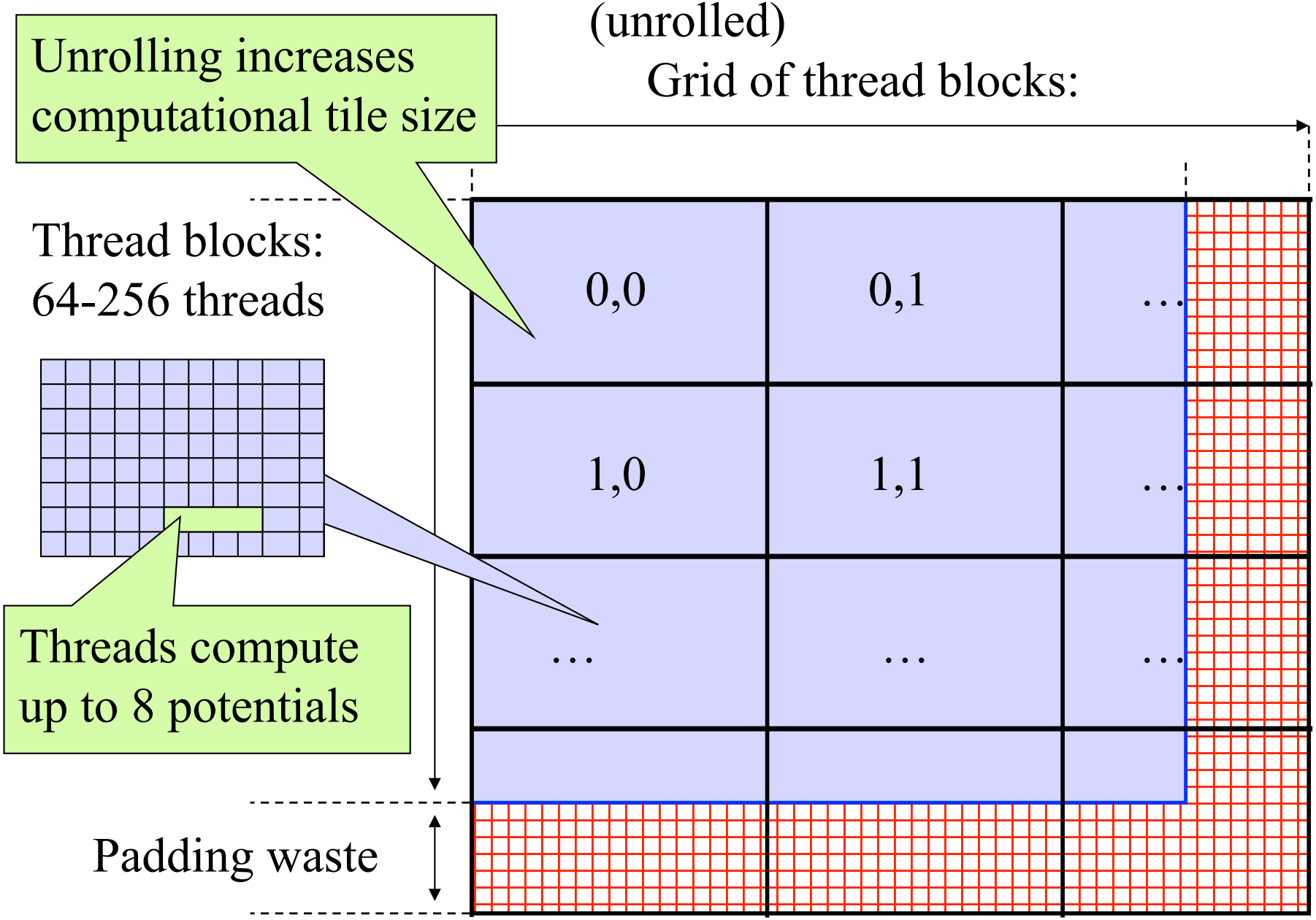
DCS CUDA Algorithm: Unrolling Loops

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

```
potentialA +=  charge[i] / (distanceA to atom[i])
potentialB +=  charge[i] / (distanceB to atom[i])
…
```



DCS CUDA Block/Grid Decomposition



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 * (1.0f / sqrtf(dx1*dx1 + dysqpdzsq));
    energyvalx2 += atominfo[atomid].w * (1.0f / sqrtf(dx2*dx2 + dysqpdzsq));
    energyvalx3 += atominfo[atomid].w * (1.0f / sqrtf(dx3*dx3 + dysqpdzsq));
    energyvalx4 += atominfo[atomid].w * (1.0f / sqrtf(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 is too long to show as a snippet due to the large amount of manual unrolling of loads into registers
- Source code is available by request

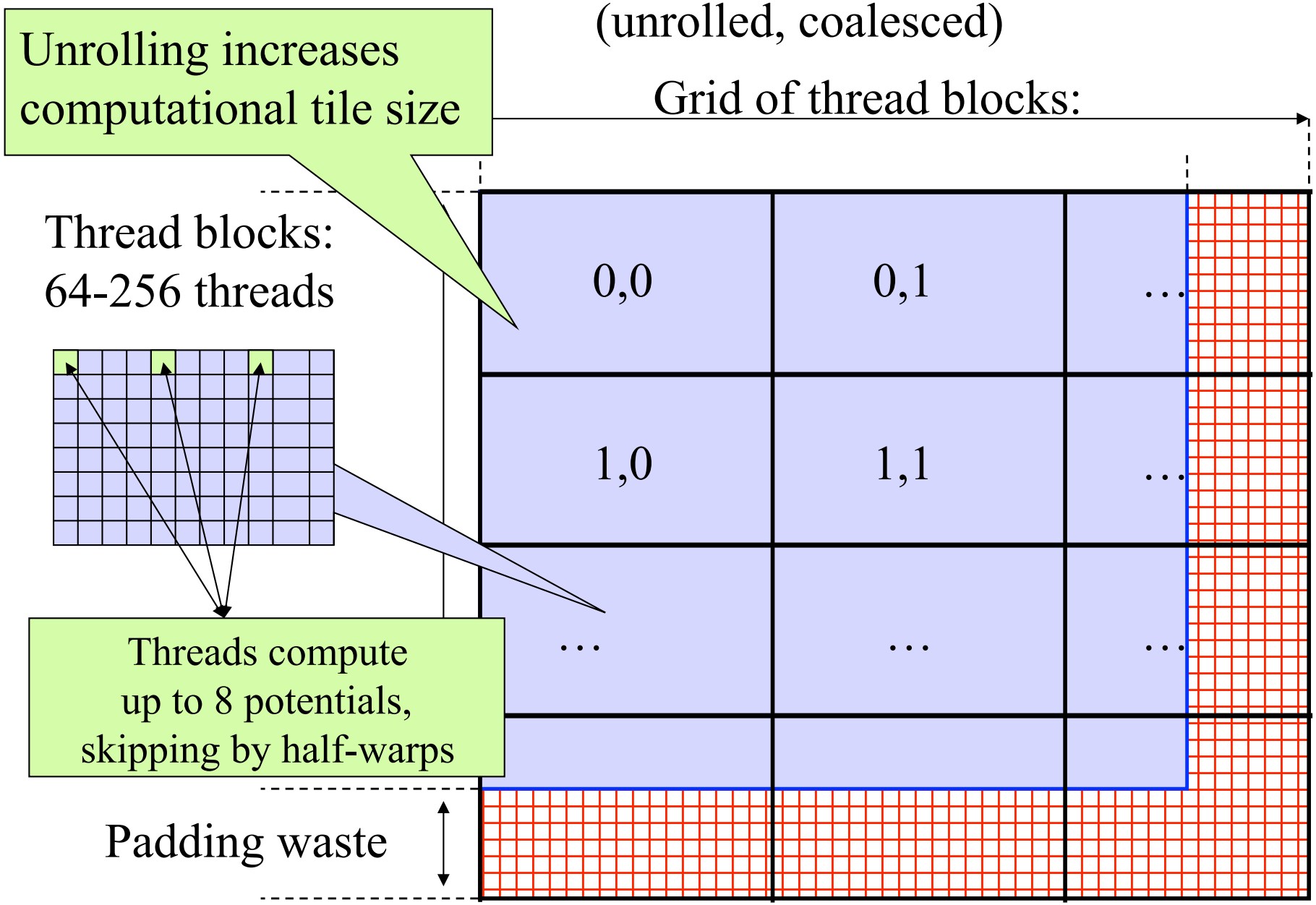
DCS Version 4: Const+Loop Unrolling+Coalescing 291.5 GFLOPS, 39.5 Billion Atom Evals/Sec

- Christopher Rodrigues found an even better formulation!
- 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
- Code is too long to show as a snippet
- Source code is available by request

DCS CUDA Block/Grid Decomposition



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