GPU Teaching Kit

Accelerated Computing

Module 16 - Application Case Study – Electrostatic Potential Calculation

Lecture 16.2 - Electrostatic Potential Calculation - Part 2

Objective

- To learn how to apply parallel programming techniques to an application
  - A fast gather kernel
  - Thread coarsening for more work efficiency
  - Data structure padding for reduced divergence
  - Memory access locality and pre-computation techniques

A Slower Sequential C Version

```
void cenergy(float *energygrid, dim3 grid, float gridspacing, float z, const float *atoms, int
numatoms) {

  int atomarrdim = numatoms * 4;
  int k = z / gridspacing;
  for (int j=0; j<grid.y; j++) {
    float y = gridspacing * (float) j;
    for (int i=0; i<grid.x; i++) {
      float x = gridspacing * (float) i;
      float energy = 0.0f;
      for (int 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;
    }
  }
}
```

Output oriented.

A Slower Sequential C Version

```c
void cenergy(float *energygrid, dim3 grid, float gridspacing, float z, const
float *atoms, int numatoms) {

  int atomarrdim = numatoms * 4;
  int k = z / gridspacing;
  for (int j=0; j<grid.y; j++) {
    float y = gridspacing * (float) j;
    for (int i=0; i<grid.x; i++) {
      float x = gridspacing * (float) i;
      float energy = 0.0f
      for (int 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;
    }
  }
}
```

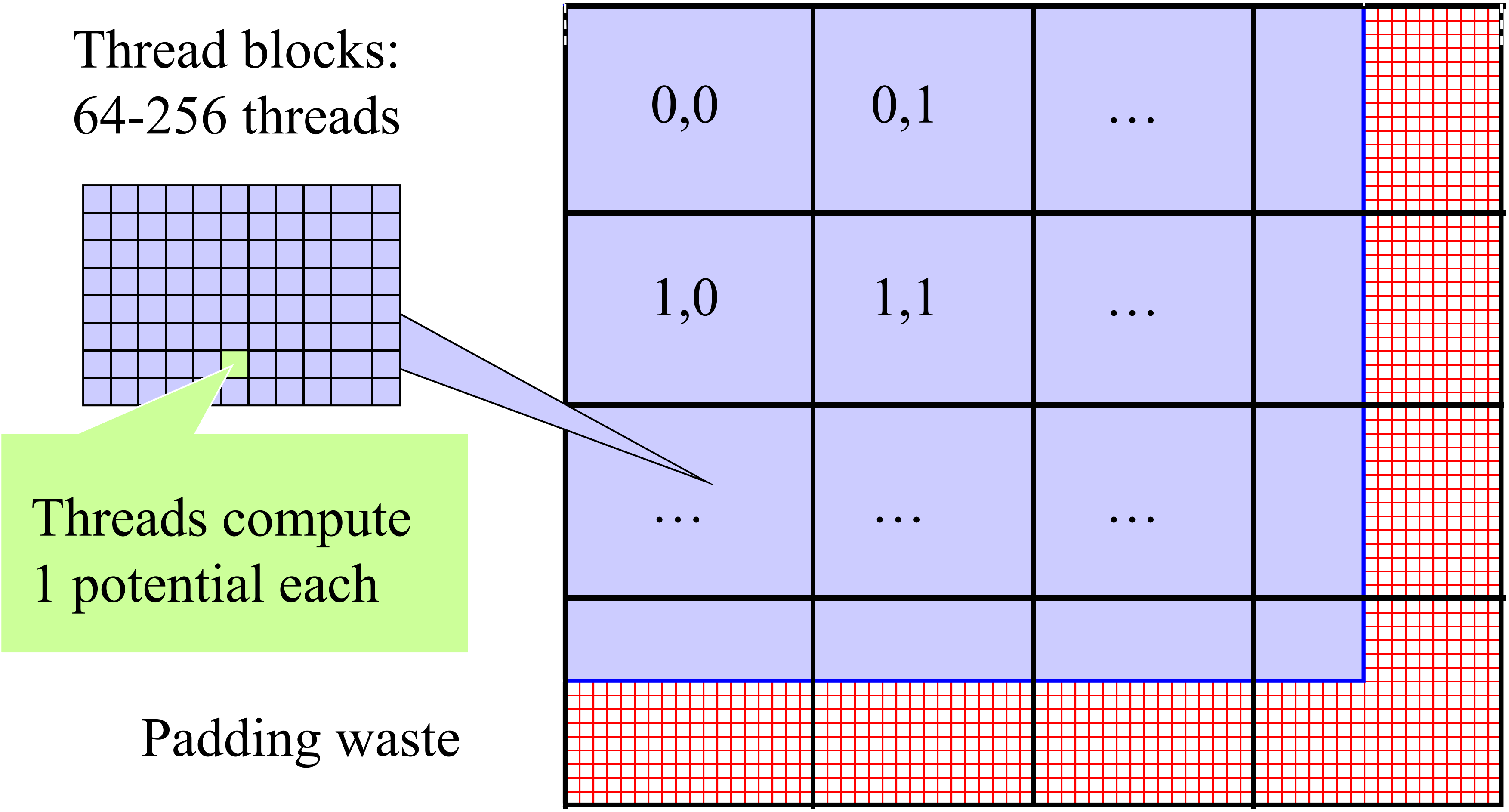
More redundant work.

Pros and Cons of the Slower Sequential Code

- Pros
  - Fewer access to the energygrid array
  - Simpler code structure

- Cons
  - Many more calculations on the coordinates
  - More access to the atom array
  - Overall, much slower sequential execution due to the sheer number of calculations performed

Simple DCS CUDA Block/Grid Decomposition

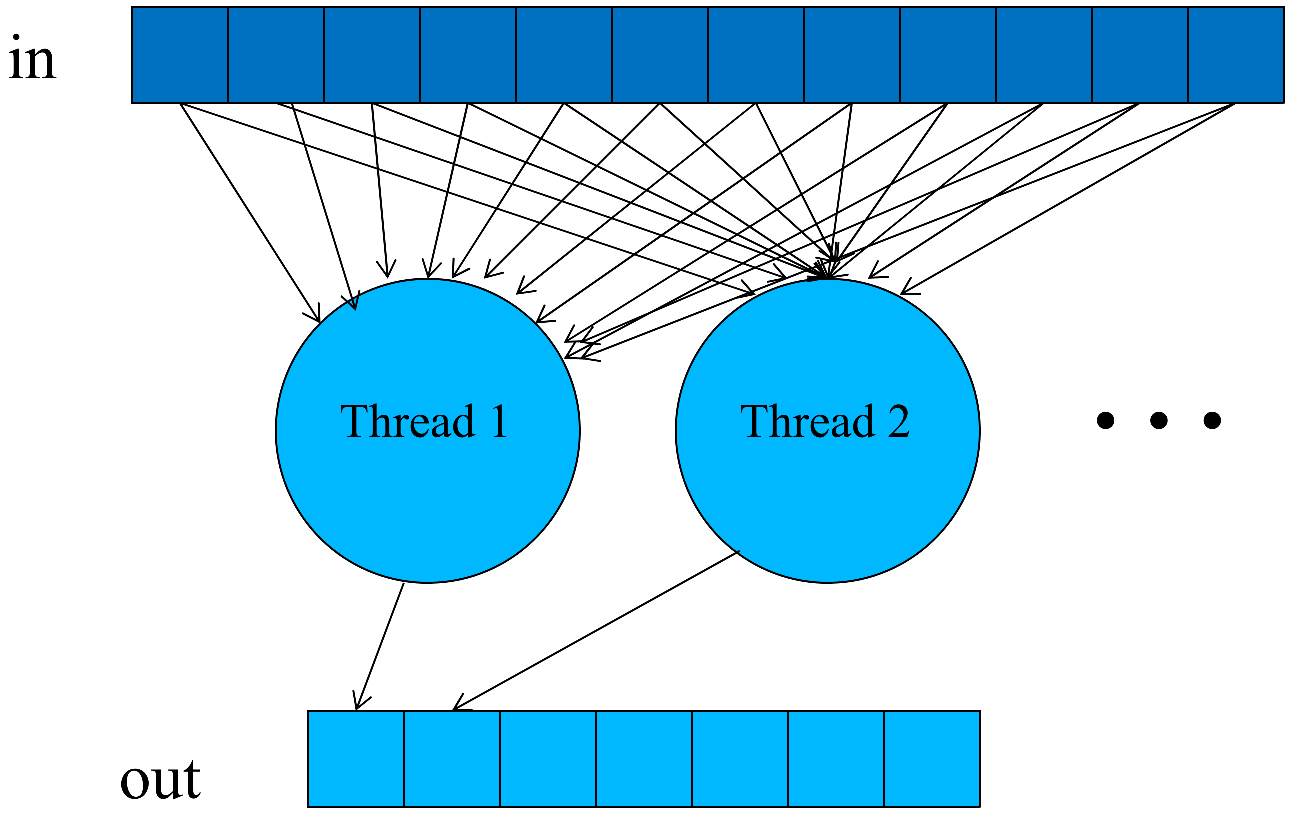


Thread blocks:
64-256 threads

Threads compute
1 potential each

Padding waste

Gather Parallelization



A Fast DCS CUDA Gather Kernel

```
void __global__ cenergy(float *energygrid, dim3 grid, float gridspacing, float z, float *atoms,
int numatoms) {

  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int atomarrdim = numatoms * 4;
  int k = z / gridspacing;
  float y = gridspacing * (float) j;
  float x = gridspacing * (float) i;
  float energy = 0.0f;
  for (int 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;
}
```

One thread per grid point

A Fast DCS CUDA Gather Kernel

```
void __global__ cenergy(float *energygrid, dim3 grid, float gridspacing, float z, float *atoms,
int numatoms) {

  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int atomarrdim = numatoms * 4;
  int k = z / gridspacing;
  float y = gridspacing * (float) j;
  float x = gridspacing * (float) i;
  float energy = 0.0f;
  for (int 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;
}
```

All threads access all atoms.

Consolidated writes to grid points

Additional Comments

- Gather kernel is much faster than a scatter kernel
  - No serialization due to atomic operations
- Compute efficient sequential algorithm does not translate into the fast parallel algorithm
  - Gather vs. scatter is a big factor
  - But we will come back to this point later!

Even More Comments

- In modern CPUs, cache effectiveness is often more important than compute efficiency
- The input oriented (scatter) sequential code actually has bad cache performance
  - energygrid[] is a very large array, typically 20X or more larger than atom[]
  - The input oriented sequential code sweeps through the large data structure for each atom, trashing cache.

Outline of A Fast Sequential Code

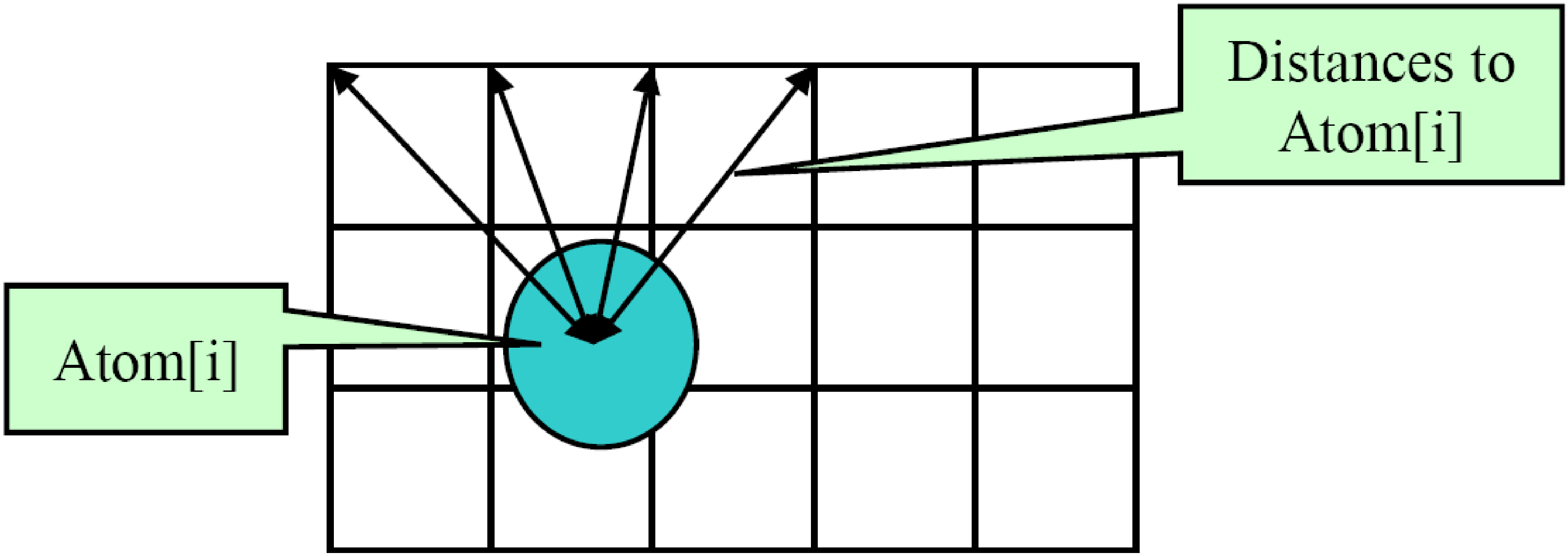
```
for all z {
  for all atoms {pre-compute dz2 }
    for all y {
      for all atoms {pre-compute dy2 (+ dz2) }
        for all x {
          for all atoms {
            compute contribution to current x,y,z point
            using pre-computed dy2 + dz2
          }
        }
      }
    }
  }
}
```

More Thoughts on Fast Sequential Code

- Need temporary arrays for pre-calculated dz2 and dy2 + dz2 values
- So, why does this code has better cache behaior on CPUs?

Reuse Distance Calculation for More Computation Efficiency

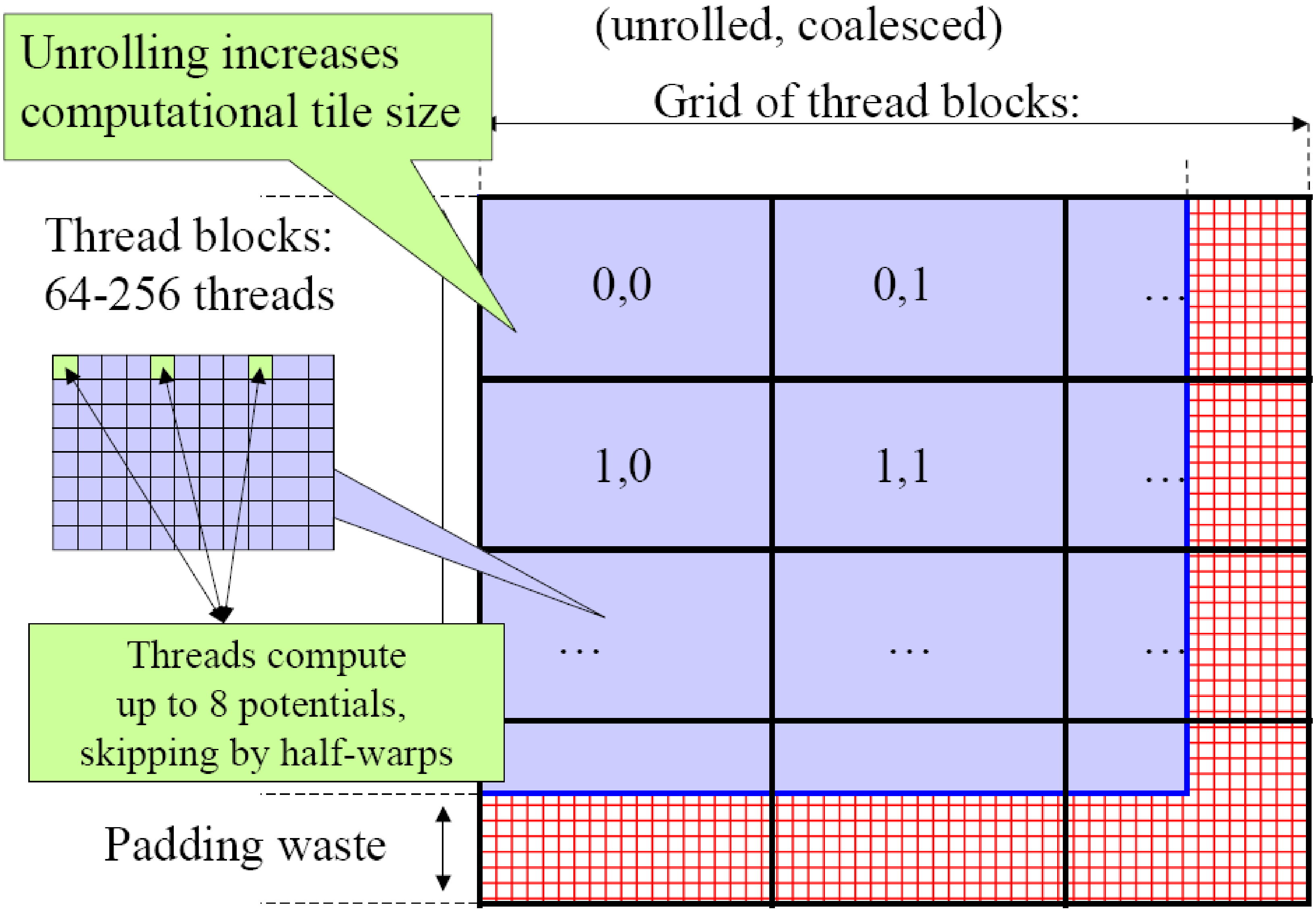
. . .



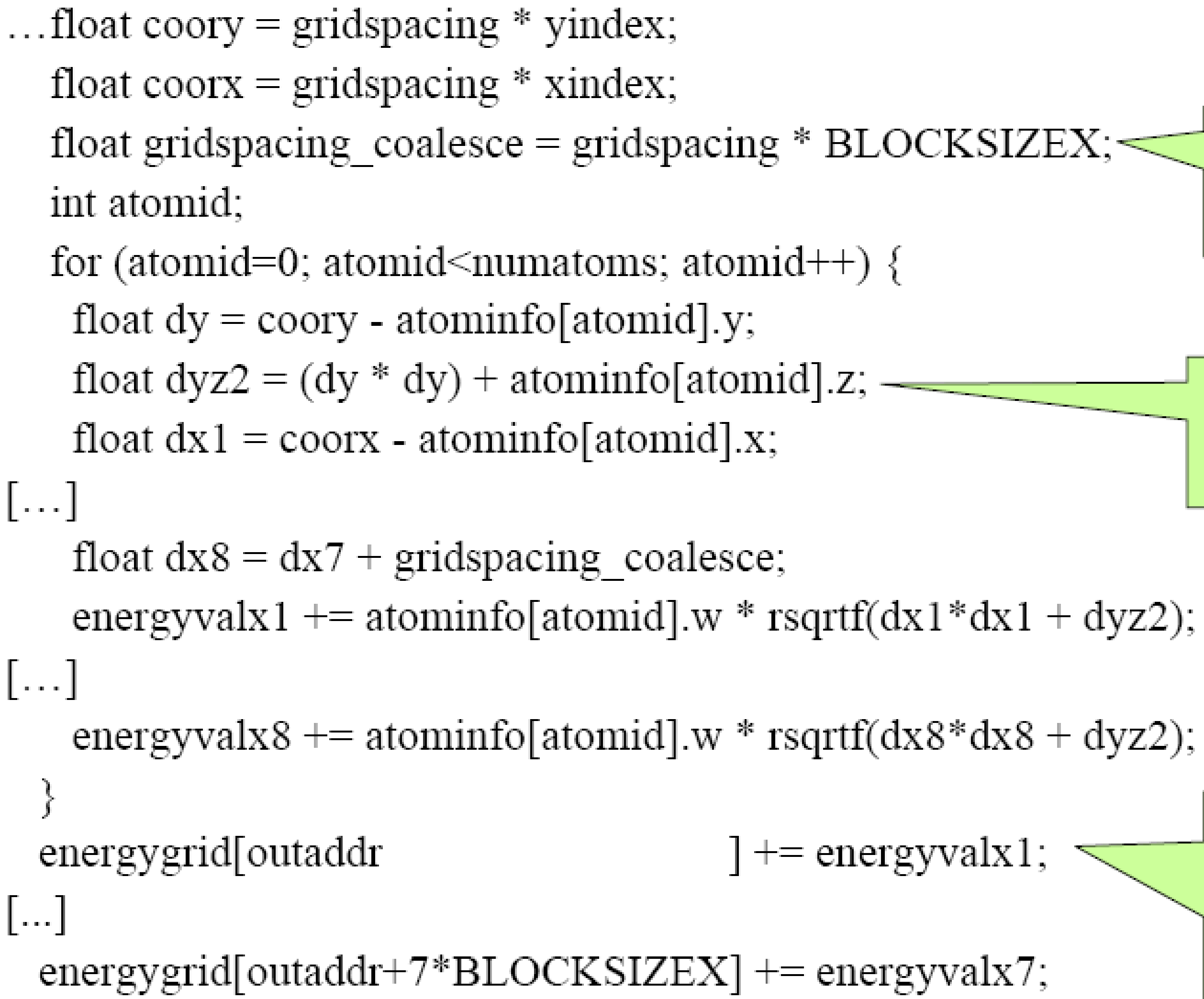
Distances to Atom[i]

Atom[i]

Thread Coarsening



A Compute Efficient Gather Kernel



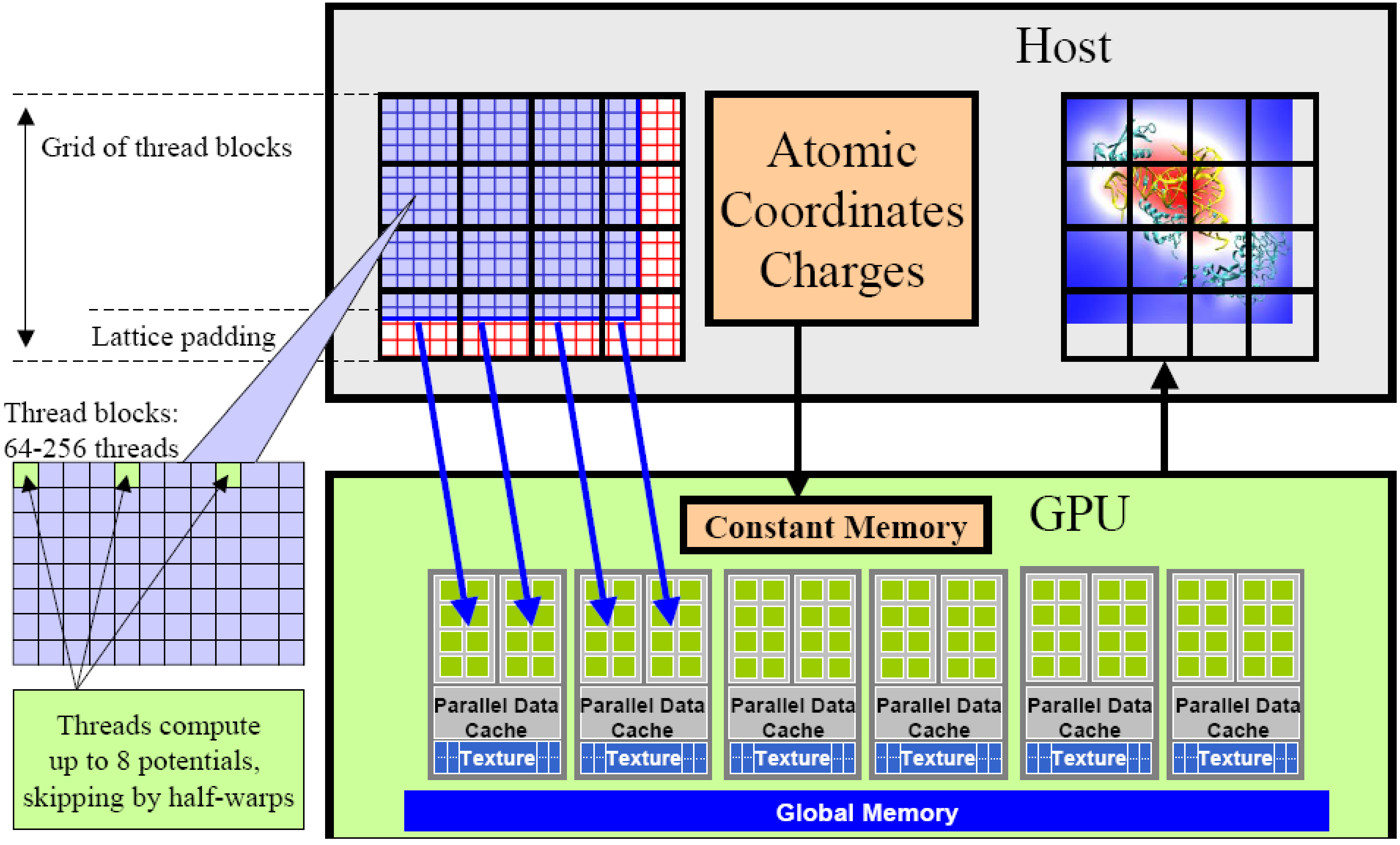
```
…float coory = gridspacing * yindex;
  float coorx = gridspacing * xindex;
  float gridspacing_coalesce = gridspacing * BLOCKSIZEX;
  int atomid;
  for (atomid=0; atomid<numatoms; atomid++) {
    float dy = coory - atominfo[atomid].y;
    float dyz2 = (dy * dy) + atominfo[atomid].z;
    float dx1 = 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

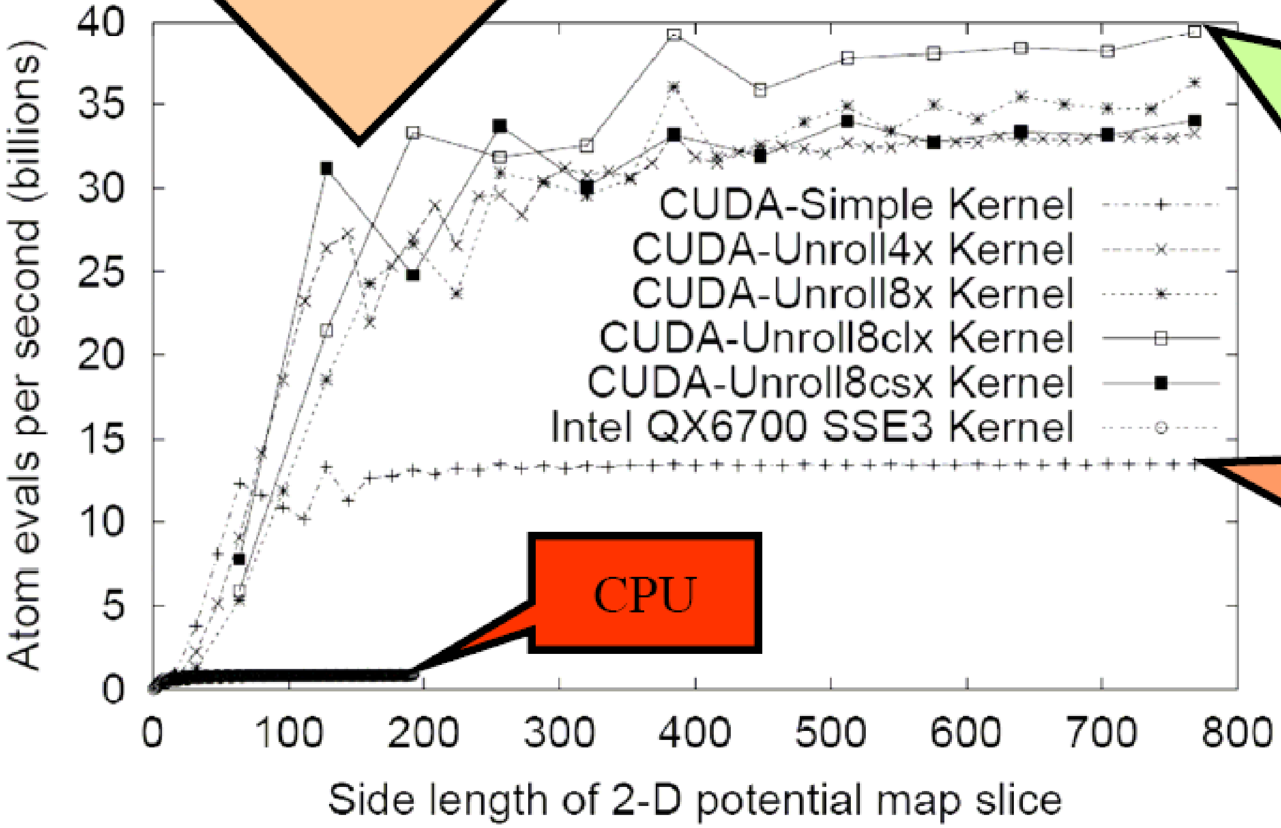
Global memory ops occur only at the end of the kernel, decreases register use

Thread Coarsening for More Computation Efficiency



Performance Comparison

Number of thread blocks modulo number of SMs results in significant performance variation for small workloads



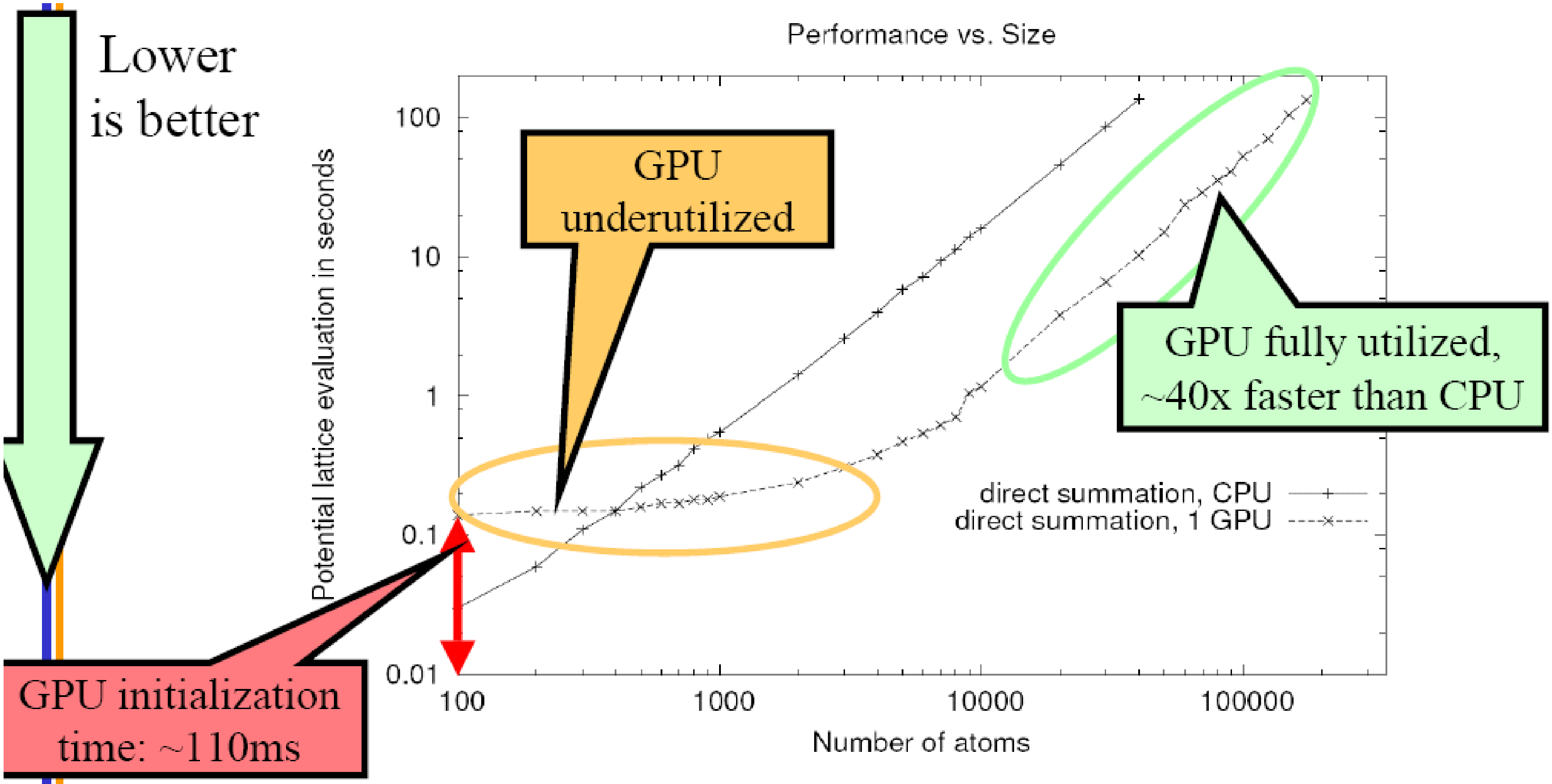
CUDA-Unroll8clx: fastest GPU kernel, 44x faster than CPU, 291 GFLOPS on GeForce 8800GTX

CUDA-Simple: 14.8x faster, 33% of fastest GPU kernel

CPU

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

More Work is Needed to Feed a GPU



Lower is better

GPU underutilized

GPU fully utilized, ~40x faster than CPU

GPU initialization time: ~110ms

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.

NVIDIA.

GPU Teaching Kit

Accelerated Computing

ILLINOIS

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN