VSCSE Summer School

Proven Algorithmic Techniques for Many-core Processors

Lecture 2: Parallelism Scalability Transformations
A Common Sequential Computation Pattern

Iterate over out

Iterate over in

Double Nested Loop

In

Out
A Simple Code Example

```c
for (m = 0; m < M; m++) {
    for (n = 0; n < N; n++) {
        out[n] += f(in[m], m, n);
    }
}
```

- **Input data in**
  - $M = \# \text{ scan points}$

- **Output data out**
  - $N = \# \text{ regularized scan points}$

- **Complexity is** $O(MN)$

- **Output tends to be more regular than input**

©Wen-mei W. Hwu and David Kirk/NVIDIA Urbana, Illinois, August 2-5, 2010
Scatter Parallelization

Thread 1

Thread 2

...
Scatter can be very slow.

- All threads have conflicting updates to the same `out` elements
  - Serialized with atomic operations
  - Very costly (slow) for large number of threads

©Wen-mei W. Hwu and David Kirk/NVIDIA Urbana, Illinois, August 2-5, 2010
Gather Parallelization

Thread 1

Thread 2

• • •
Gather can be very fast.

- All threads can read the same in elements
  - No serialization
  - Can even be efficiently consolidated through caches or local memories

All threads update their own out elements
Why is scatter parallelization often used rather than gather?

- In practice, each in does not affect all out elements
- Out tend to be much more regular than in
- It is easy to calculate all out elements affected by an in element
  - Harder to calculate all in elements to affect an out
  - Easy thread kernel code if written in scatter

©Wen-mei W. Hwu and David Kirk/NVIDIA Urbana, Illinois, August 2-5, 2010
Challenges in Gather Parallelization

• Regularize input elements so that it is easier to find all in elements that affects an out element
  – Cut-off Binning Lecture

• Can be even more challenging if data is highly non-uniform
  – Cut-off Binning for Non-Uniform Data Lecture

• For this lecture, we assume that all in elements affect all out elements
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
  – Most time consuming, focus of our example.

• 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
- All atoms affect all map lattice points, most accurate
- Approximation-based methods such as multilevel summation can achieve much higher performance at the cost of some numerical accuracy and flexibility
  - Will cover these later
- DCS: for each lattice point, sum potential contributions for all atoms in the simulated structure:
  
  \[
  \text{potential} += \frac{\text{charge}[i]}{\text{(distance to atom}[i])}
  \]
Direct Coulomb Summation (DCS) Algorithm Detail

- At each lattice point, sum potential contributions for all atoms in the simulated structure:
  \[
  \text{potential} \,=\, \text{charge}[i] \, / \, (\text{distance to atom}[i])
  \]
Electrostatic Potential Map Calculation Function Overview

• Each call calculates an x-y slice of the energy map
  – energygrid – pointer to the entire potential map
  – grid – the x, y, z dimensions of the potential map
  – gridspacing – modeled physical distance between grid points
  – atoms – array of x, y, z coordinates and charge of atoms
  – numatoms – number of atoms in atoms array

void cenergy(float *energygrid, dim3 grid, float gridspacing, float z, const float *atoms, int numatoms) {}
An Intuitive Sequential C Version

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

    int atomarrdim = numatoms * 4; // x, y, z, and charge info for each atom
    for (int n=0; n<atomarrdim; n+=4) {  // calculate potential contribution of each atom
        float dz = z - atoms[n+2]; // all grid points in a slice have the same z value
        float dz2 = dz*dz;
        int grid_slice_offset = (grid.x*grid.y*z) / grids spacing;
        float charge = atoms[n+3];
        for (int j=0; j<grid.y; j++) {
            float y = grids spacing * (float) j;
            float dy = y - atoms[n+1]; // all grid points in a row have the same y value
            float dy2 = dy*dy;
            int grid_row_offset = grid_slice_offset + grid.x*j;
            for (int i=0; i<grid.x; i++) {
                float x = grids spacing * (float) i;
                float dx = x - atoms[n ];
                energygrid[grid_row_offset + i] += charge / sqrtf(dx*dx + dy2+ dz2);
            }
        }
    }
}
```
Summary of Simple Sequential C Version

• Algorithm is input oriented
  – For each input atom, calculate its contribution to all grid points in an x-y slice

• Output (energygrid) is very regular
  – Simple linear mapping between grid point indices and modeled physical coordinates

• Input (atom) is irregular
  – Modeled x,y,z coordinate of each atom needs to be stored in the atom array

• The algorithm is efficient in performing minimal calculations on distances, coordinates, etc.
Irregular Input vs. Regular Output

- Atoms come from modeled molecular structures, solvent (water) and ions
  - Irregular by necessity

- Energy grid models the electrostatic potential value at regularly spaced points
  - Regular by design
Straightforward CUDA Parallelization

• Use each thread to compute the contribution of an atom to all grid points
  – Scatter parallelization

• Kernel code largely correspond to CPU version with outer loop stripped
  – Each thread corresponds to an outer loop iteration of CPU version
  – Numatoms used in kernel launch configuration host code
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:
    • Copy atom data to GPU
    • Run CUDA Kernel on atoms and slice resident on GPU
  – Copy slice from GPU to host
• Free resources
A Very Slow DCS Scatter Kernel!

```c
void __global__ cenergy(float *energygrid, float *atoms, dim3 grid, float gridspacing, float z) {
    int n = blockIdx.x * blockDim.x + threadIdx.x;
    float dz = z - atoms[n+2];  // all grid points in a slice have the same z value
    float dz2 = dz*dz;
    int grid_slice_offset = (grid.x*grid.y*z) / gridspacing;
    float charge = atoms[n+3];
    for (int j=0; j<grid.y; j++) {
        float y = gridspacing * (float) j;
        float dy = y - atoms[n+1];  // all grid points in a row have the same y value
        float dy2 = dy*dy;
        int grid_row_offset = grid_slice_offset + grid.x*j;
        for (int i=0; i<grid.x; i++) {
            float x = gridspacing * (float) i;
            float dx = x - atoms[n ];
            energygrid[grid_row_offset + i] += charge / sqrtf(dx*dx + dy2 + dz2));
        }
    }
}
```
Pros and Cons of the Scatter Kernel

• Pros
  – Follows closely the CPU version
  – Good for software engineering and code maintenance
  – Preserves computation efficiency (coordinates, distances, offsets) of sequential code

• Cons
  – The atomic add serializes the execution, very slow!
  – Not even worth trying this yourself.
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;
        }
    }
}
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
DCS CUDA Block/Grid Decomposition
(no register tiling)

Grid of thread blocks:

Thread blocks: 64-256 threads

Threads compute 1 potential each

Padding waste
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;
}

©Wen-mei W. Hwu and David Kirk/NVIDIA Urbana, Illinois, August 2-5, 2010
Additional Comments

• Further optimizations
  – $dz^2$ can be pre-calculated and sent in place of $z$

• Gather kernel is much faster than a scatter kernel
  – Whereas the CPU sequential code prefers scatter style code

• 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 sequential code actually has very 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.
• The fastest sequential code is actually an optimized output oriented code
Outline of A Fast Sequential Code

for all z {
    for all atoms {precompute $dz^2$ }
    for all y {
        for all atoms {precompute $dy^2 (+ dz^2)$ }
        for all x {
            for all atoms {
                compute contribution to current $x,y,z$ point
                using precomputed $dy^2$ and $dz^2$
            }
        }
    }
}
REGISTER TILING FOR EFFICIENCY
Basic Idea

- Parallel execution sometime requires doing redundant work
  - Merging multiple threads into one allows re-use of result, avoiding redundant work
Outline of Technique

• Merge multiple threads so each resulting thread calculates multiple output elements
  – Perform the redundant work once and save result into registers
  – Use register result for calculating all output elements

• Merged kernel code will use more registers
  – May reduce the number of threads allowed on an SM
  – Increased efficiency may outweigh reduced parallelism

• Also referred to as thread coarsening
For DCS Kernel

- merge threads to 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 per thread work 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
• Add each atom’s contribution to several lattice points at a time, where distances only differ in one component:

\[
\text{potential}_A \ += \ \text{charge}[i] / (\text{distance}_A \text{ to } \text{atom}[i]) \\
\text{potential}_B \ += \ \text{charge}[i] / (\text{distance}_B \text{ to } \text{atom}[i]) \\
\ldots
\]
DCS CUDA Block/Grid Decomposition

(Coarsened, coalesced)

Coarsening increases computational tile size

Threads compute up to 8 potentials, skipping by half-warps
DCS Coarsened 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 the summation, avoiding register consumption
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));
}
More Comments on Coarsened Kernel

• Pros:
  – We can reduce the number of loads by reusing atom coordinate values for multiple voxels, by storing in regs
  – By merging multiple points into each thread, we can compute $dy^2+dz^2$ once and use it multiple times, much like the fast CPU version of the code
  – A good balance between efficiency, locality and parallelism

• Cons:
  – Uses more registers, one of several limited resources
  – Increases effective tile size, or decreases thread count in a block, though not a problem at this level
ANY MORE QUESTIONS?