CS 677: Parallel Programming for Many-core Processors
Lecture 9

Instructor: Philipppos Mordohai
Webpage: www.cs.stevens.edu/~mordohai
E-mail: Philipppos.Mordohai@stevens.edu
Final Projects

• 31 days left to final presentations…
  – 34 days to report deadline
• Have you started?
• Will your timeline work?
Overview

• Case Study – Electrostatic Potential Calculation
  – A class project at UIUC also resulting in publications
  – Chapter 12 in K&H

• Input Binning
  – From NVIDIA and University of Houston

• Computational Thinking
  – Chapter 13 in K&H
Electrostatic potential map is used in building stable structures for molecular dynamics simulation
The contribution of atom[i] to the electrostatic potential at lattice point j is atom[i].charge / r_{ij}.

The total potential at lattice point j is the sum of contributions from all atoms in the system.
Sequential CPU Code

```c
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;
        }
    }
}
```
GPU Implementation

• Option 1: each thread calculates the contribution of one atom to all grid points
  – “Scatter”
• Option 2: each thread calculates the accumulated contributions of all atoms to one grid point
  – “Gather”
• Pros/cons?
Loop Transformation

• Need perfectly nested loops
  – as in MRI example
  – Move calculation of y into inner loop
  – Pros/cons?

```c
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) {
            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 Kernel Design Overview

Grid of thread blocks

Lattice padding

Thread blocks: 64-256 threads

Threads compute up to 8 potentials, skipping by half-warps

Atomic Coordinates Charges

Constant Memory

Global Memory

Host

GPU
DCS Kernel Version 1

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;

qsqrtf(): reciprocal square root

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 Kernel Version 1

... 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 *
                 rsnrqtf(dx*dx + dy*dy + atominfo[atomid].z);
}
energygrid[outaddr] = curenergy + energyval;

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

ILP vs. TLP

atominfo[].z is already squared

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

qsqrtf(): reciprocal square root
Information Reuse

Atom[i]

Distances to Atom[i]
...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 * rsqrts(dx1*dx1 + dysqpdzsq);
    energyvalx2 += charge * rsqrts(dx2*dx2 + dysqpdzsq);
    energyvalx3 += charge * rsqrts(dx3*dx3 + dysqpdzsq);
    energyvalx4 += charge * rsqrts(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...
Memory Coalescing

• Two issues:
  – Each thread calculates potentials of four adjacent grid points
  
  – If grid width is not multiple of tile width, boundary management becomes complicated
Memory Layout for Coalescing

Grid of thread blocks:

Thread blocks: 64-256 threads

0,0 0,1 ...

1,0 1,1 ...

... ...

Threads compute up to 8 potentials, skipping by half-warps

Unrolling increases computational tile size

Padding waste
...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

ILP vs. TLP
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 vs. CPU-GPU Comparison

Lower is better

GPU initialization time: ~110ms

Accelerating molecular modeling applications with graphics processors.
Input Binning
Objective

• To understand how data scalability problems in gather parallel execution motivate input binning
• To learn basic input binning techniques
• To understand common tradeoffs in input binning
Scatter to Gather Transformation
However

• Input tends to be much less regular than output
  – It may be difficult for each thread to efficiently locate all inputs relevant to its output
  – Or, to efficiently exclude all inputs irrelevant to its output
• In a naïve arrangement, all threads may have to process all inputs to decide if each input is relevant to its output
  – This makes execution time scale poorly with data set size
  – Important problem for many-cores designed to process large data sets
DCS Algorithm for Electrostatic Potentials Revisited

- At each grid point, sum the electrostatic potential from all atoms
  - All threads read all inputs
- Highly data-parallel
- But has quadratic complexity
  - Number of grid points $\times$ number of atoms
  - Both proportional to volume
  - Poor data scalability

$q/r$ added to the potential here

Distance $r$

Atom with charge $q$
Algorithm for Electrostatic Potentials With a Cutoff

- Ignore atoms beyond a *cutoff distance*, $r_c$
  - Typically 8Å-12Å
  - Long-range potential may be computed separately
- Number of atoms within cutoff distance is roughly constant (uniform atom density)
  - 200 to 700 atoms within 8Å-12Å cutoff sphere for typical biomolecular structures
Implementation Challenge

• For each tile of grid points, we need to identify the set of atoms that need to be examined
  – One could naively examine all atoms and only use the ones whose distance is within the given range
  – But this examination still takes time, and brings the time complexity right back to
    • number of atoms × number of grid points
  – Each thread needs to avoid examining the atoms outside the range of its grid point(s)
Binning

• A process that groups data to form a chunk called *bin*
• Helps problem solving due to data coarsening
• Uniform bin arrays, Variable bins, KD Trees, ...
Binning for Cut-Off Potential

• Divide the simulation volume with non-overlapping uniform cubes
• Every atom in the simulation volume falls into a cube based on its spatial location
  – Bins represent location property of atoms
• After binning, each cube has a unique index in the simulation space for easy parallel access

(a) Simulation volume

(b) Simulation volume with eight bins
Spatial Sorting Using Binning

- Presort atoms into bins by location in space
- Each bin holds several atoms
- Cutoff potential only uses bins within $r_c$
  - Yields a linear complexity cutoff potential algorithm

©Wen-mei W. Hwu and David Kirk/NVIDIA 2010
Bin Size Considerations

• Capacity of atom bins needs to be balanced
  – Too large - many dummy atoms in bins
  – Too small - some atoms will not fit into bins
  – Target bin capacity to cover more than 95% or atoms

• CPU places all atoms that do not fit into bins into an overflow bin
  – Use a CPU sequential algorithm to calculate their contributions to the energy grid lattice points.
  – CPU and GPU can do potential calculations in parallel
Bin Design

- Uniform sized/capacity bins allow array implementation
  - And the relative offset list approach

- Bin capacity should be big enough to contain all the atoms that fall into a bin
  - Cut-off will screen away atoms that weren’t processed
  - Performance penalty if too many are screened away
Going from DCS Kernel to Large Bin Cut-off Kernel

• Adaptation of techniques from the direct Coulomb summation kernel for a cutoff kernel
• Atoms are stored in constant memory as with DCS kernel
• CPU loops over potential map regions that are \((24\text{Å})^3\) in volume (cube containing cutoff sphere)
• Large bins of atoms are appended to the constant memory atom buffer until it is full, then GPU kernel is launched
• Host loops over map regions reloading constant memory and launching GPU kernels until completion
Large Bin Design Concept

- Map regions are \((24\AA)^3\) in volume
- Regions are sized large enough to provide the GPU enough work in a single kernel launch
  - \((48\text{ lattice points})^3\) for lattice with 0.5\AA spacing
  - Small bins don’t provide the GPU enough work to utilize all SMs, to amortize constant memory update time, or kernel launch overhead
Large-bin Cutoff Kernel Evaluation

• 6× speedup relative to fast CPU version
• Work-inefficient
  – Coarse spatial hashing into (24Å)³ bins
  – Only 6.5% of the atoms a thread tests are within the cutoff distance
• Better adaptation of the algorithm to the GPU will gain another 2.5×
Improving Work Efficiency

• Thread block examines atom bins up to the cutoff distance
  – Use a sphere of bins
  – All threads in a block scan the same bins and atoms
    • No hardware penalty for multiple simultaneous reads of the same address
    • Simplifies fetching of data
  – The sphere has to be big enough to cover all grid point at corners
  – There will be a small level of divergence
    • Not all grid points processed by a thread block relate to all atoms in a bin the same way
    • (A within cut-off distance of N but outside cut-off of M)
The Neighborhood is a volume

- Calculating and specifying all bin indexes of the sphere can be quite complex
  - Rough approximations reduce efficiency
Neighborhood Offset List (Pre-calculated)

• A list of relative offsets enumerating the bins that are located within the cutoff distance for a given location in the simulation volume

• Detection of surrounding atoms becomes realistic for output grid points
  – By visiting bins in the neighborhood offset list and iterating over the atoms they contain
Pseudo Code

// 1. binning
for each atom in the simulation volume,
    index_of_bin = atom.addr / BIN_SIZE
    bin[index_of_bin] <= atom

// 2. generate the neighborhood offset list
for each c from -cutoff to cutoff,
    if distance(0, c) < cutoff,
        nlist <= c

// 3. do the computation
for each point in the output grid,
    index_of_bin := point.addr / BIN_SIZE
    for each offset in nlist,
        for each atom in bin[index_of_bin + offset],
            point.potential += atom.charge / (distance from point to atom)
Performance

• $O(MN')$ where $M$ and $N'$ are the number of output grid points and atoms in the neighborhood offset list, respectively
  – In general, $N'$ is small compared to the number of all atoms

• Works well if the distribution of atoms is uniform
Details on Small Bin Design

- For 0.5Å lattice spacing, a $(4\AA)^3$ cube of the potential map is computed by each thread block
  - $8\times8\times8$ potential map points
  - 128 threads per block (4 points/thread)
  - 34% of examined atoms are within cutoff distance
More Design Considerations for the Cutoff Kernel

• High memory throughput to atom data essential
  – Group threads together for locality
  – Fetch bins of data into shared memory
  – Structure atom data to allow fetching

• After taking care of memory demand, optimize to reduce instruction count
  – Loop and instruction-level optimization
Tiling Atom Data

- Shared memory used to reduce Global Memory bandwidth consumption
  - Threads in a thread block collectively load one bin at a time into shared memory
  - Once loaded, threads scan atoms in shared memory
  - Reuse: Loaded bins used 128 times

**Execution cycle of a thread block**

- **Time**
  - Threads individually compute potentials using bin in shared mem
  - Collectively load next bin
  - Suspend
  - Data returned from global memory
  - Ready
  - Write bin to shared memory

Another thread block runs while this one waits
Handling Overfull Bins

• In typical use, 2.6% of atoms exceed bin capacity
• Spatial sorting puts these into a list of extra atoms
• Extra atoms processed by the CPU
  – Computed with CPU-optimized algorithm
  – Takes about 66% as long as GPU computation
  – Overlapping GPU and CPU computation yields additional speedup
  – CPU performs final integration of grid data
CPU Grid Data Integration

- Effect of overflow atoms are added to the CPU master energygrid array
- Slice of grid point values calculated by GPU are added into the master energygrid array while removing the padded elements
GPU Thread Coarsening

- Each thread computes potentials at four potential map points
  - Reuse x and z components of distance calculation
  - Check x and z components against cutoff distance (cylinder test)
- Exit inner loop early upon encountering the first empty slot in a bin
for (i = 0;  i < BIN_DEPTH;  i++) {
    aq = AtomBinCache[i].w;
    if (aq == 0) break;

    dx = AtomBinCache[i].x - x;
    dz = AtomBinCache[i].z - z;
    dxdz2 = dx*dx + dz*dz;
    if (dxdz2 < cutoff2) continue;

    dy = AtomBinCache[i].y - y;
    r2 = dy*dy + dxdz2;
    if (r2 < cutoff2)
        poten0 += aq * rsqrf(r2);
        // Simplified example

    dy = dy - 2 * grid_spacing;
    /* Repeat three more times */
}
Cutoff Summation Runtime

![Graph showing execution time vs. volume of potential map (Angstrom^3)]

- CPU-SSE3
- LargeBin
- SmallBin
- SmallBin-Overlap

GPU cutoff with CPU overlap: 12x-21x faster than CPU core

50k–1M atom structure size
Summary

• Large bins allow re-use of all-input kernels with little code change
  – But work efficiency can be very low
• Use of small-sized bins require more sophisticated kernel code to traverse list of small bins
  – Much higher work efficiency
  – Small bins also serve as tiles for locality
• CPU process overflow atoms from fixed capacity bins
Kirk & Hwu Chapter 13: Computational Thinking
Objective

• To provide you with a framework based on the techniques and best practices used by experienced parallel programmers for
  – Thinking about the problem of parallel programming
  – Addressing performance and functionality issues in your parallel program
  – Using or building useful tools and environments
  – Understanding case studies and projects
Fundamentals of Parallel Computing

• Parallel computing requires that
  – The problem can be decomposed into sub-problems that can be safely solved at the same time
  – The programmer structures the code and data to solve these sub-problems concurrently

• The goals of parallel computing are
  – To solve problems in less time, and/or
  – To solve bigger problems, and/or
  – To achieve better solutions

The problems must be large enough to justify parallel computing and to exhibit exploitable concurrency.
Key Parallel Programming Steps

1) To find the concurrency in the problem
2) To structure the algorithm so that concurrency can be exploited
3) To implement the algorithm in a suitable programming environment
4) To execute and tune the performance of the code on a parallel system

Unfortunately, these have not been separated into levels of abstractions that can be dealt with independently.
Amdahl’s Law

• “The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.”

• Example
  – 95% of original execution time can be sped up by 100x on GPU
  – Speed up for entire application:

\[
\frac{1}{\left(5\% + \frac{95\%}{100}\right)} = \frac{1}{5\% + 0.95\%} = \frac{1}{5.95\%} = 17x
\]
Challenges of Parallel Programming

• Finding and exploiting concurrency often requires looking at the problem from a non-obvious angle
  – Computational thinking

• Dependences need to be identified and managed
  – The order of task execution may change the answers
    • Obvious: One step feeds result to the next steps
    • Subtle: numeric accuracy may be affected by ordering steps that are logically parallel with each other

• Performance can be drastically reduced by many factors
  – Overhead of parallel processing
  – Load imbalance among processor elements
  – Inefficient data sharing patterns
  – Saturation of critical resources such as memory bandwidth
Shared Memory vs. Message Passing

• We will focus on shared memory parallel programming
  – This is what CUDA is based on
  – Future massively parallel microprocessors are expected to support shared memory at the chip level

• The programming considerations of message passing model is quite different!
  – Look at MPI (Message Passing Interface) and its relatives such as Charm++
Finding Concurrency in Problems

- Identify a decomposition of the problem into sub-problems that can be solved simultaneously
  - A task decomposition that identifies tasks for potential concurrent execution
  - A data decomposition that identifies data local to each task
  - A way of grouping tasks and ordering the groups to satisfy temporal constraints
  - An analysis on the data sharing patterns among the concurrent tasks
  - A design evaluation that assesses of the quality the choices made in all the steps
Finding Concurrency - The Process

This is typically an iterative process. Opportunities exist for dependence analysis to play earlier role in decomposition.
Task Decomposition

• Many large problems can be naturally decomposed into tasks - CUDA kernels are largely tasks
  – The number of tasks used should be adjustable to the execution resources available.
  – Each task must include sufficient work in order to compensate for the overhead of managing their parallel execution.
  – Tasks should maximize reuse of sequential program code to minimize effort.

“In an ideal world, the compiler would find tasks for the programmer. Unfortunately, this almost never happens.”
- Mattson, Sanders, Massingill
Task Decomposition Example - Square Matrix Multiplication

- $P = M \times N$ of WIDTH $\times$ WIDTH
  - One natural task (sub-problem) produces one element of $P$
  - All tasks can execute in parallel in this example.
Task Decomposition Example - Molecular Dynamics

- Simulation of motions of a large molecular system
- For each atom, there are natural tasks to calculate
  - Vibrational forces
  - Rotational forces
  - Neighbors that must be considered in non-bonded forces
  - Non-bonded forces
  - Update position and velocity
  - Misc physical properties based on motions
- Some of these can go in parallel for an atom

It is common that there are multiple ways to decompose any given problem.
Data Decomposition

- The most compute intensive parts of many large problem manipulate a large data structure
  - Similar operations are being applied to different parts of the data structure, in a mostly independent manner
  - This is what CUDA is optimized for
- The data decomposition should lead to
  - Efficient data usage by tasks within the partition
  - Few dependencies across the tasks that work on different partitions
  - Adjustable partitions that can be varied according to the hardware characteristics
Task Grouping

- Sometimes natural tasks of a problem can be grouped together to improve efficiency
  - Reduced synchronization overhead - all tasks in the group can use a barrier to wait for a common dependence
  - All tasks in the group efficiently share data loaded into a common on-chip, shared storage (Shared Memory)
  - Grouping and merging dependent tasks into one task reduces need for synchronization
  - CUDA thread blocks are task grouping examples
Task Grouping Example - Square Matrix Multiplication

• Tasks calculating a P sub-block
  – Extensive input data sharing, reduced memory bandwidth using Shared Memory
  – All synched in execution
Task Ordering

• Identify the data and resource required by a group of tasks before they can be executed
  – Find the task group that creates them
  – Determine a temporal order that satisfies all data constraints
Data Sharing

• Data sharing can be a double-edged sword
  – Excessive data sharing can drastically reduce advantage of parallel execution
  – Localized sharing can improve memory bandwidth efficiency
• Efficient memory bandwidth usage can be achieved by synchronizing the execution of task groups and coordinating their usage of memory data
  – Efficient use of on-chip, shared storage
• Read-only sharing can usually be done at much higher efficiency than read-write sharing, which often requires synchronization
Data Sharing Example - Matrix Multiplication

• Each task group will finish usage of each sub-block of N and M before moving on
  – N and M sub-blocks loaded into Shared Memory for use by all threads of a P sub-block
  – Amount of on-chip Shared Memory strictly limits the number of threads working on a P sub-block

• Read-only shared data can be more efficiently accessed as Constant or Texture data
Task Ordering Example: Molecular Dynamics

- Vibrational and Rotational Forces
- Neighbor List: Complex computation involving many atoms
- Non-bonded Force
- Update atomic positions and velocities
- Next Time Step
Data Sharing Example - Molecular Dynamics

• The atomic coordinates
  – Read-only access by the neighbor list, bonded force, and non-bonded force task groups
  – Read-write access for the position update task group

• The force array
  – Read-only access by position update group
  – Accumulate access by bonded and non-bonded task groups

• The neighbor list
  – Read-only access by non-bonded force task groups
  – Generated by the neighbor list task group
Key Parallel Programming Steps

1) To find the concurrency in the problem
2) To structure the algorithm to translate concurrency into performance
3) To implement the algorithm in a suitable programming environment
4) To execute and tune the performance of the code on a parallel system
Algorithm

- A step by step procedure that is guaranteed to terminate, such that each step is precisely stated and can be carried out by a computer
  - Definiteness - the notion that each step is precisely stated
  - Effective computability - each step can be carried out by a computer
  - Finiteness - the procedure terminates

- Multiple algorithms can be used to solve the same problem
  - Some require fewer steps
  - Some exhibit more parallelism
  - Some have larger memory footprint than others
Choosing Algorithm Structure

Start

Organize by Task
- Linear
  - Task Parallelism
- Recursive
  - Divide and Conquer

Organize by Data
- Linear
  - Geometric Decomposition
- Recursive
  - Recursive Data

Organize by Data Flow
- Regular
  - Pipeline
- Irregular
  - Event Driven
Mapping a Divide and Conquer Algorithm

Thread 0
- Iterations: 0
- Array elements: 0

Thread 2
- Iterations: 1
- Array elements: 1

Thread 4
- Iterations: 2
- Array elements: 2

Thread 6
- Iterations: 3
- Array elements: 4

Thread 8
- Iterations: 4
- Array elements: 6

Thread 10
- Iterations: 5
- Array elements: 8

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2010
ECE408, University of Illinois, Urbana-Champaign
Tiled (Stenciled) Algorithms are Important for Geometric Decomposition

• A framework for memory data sharing and reuse by increasing data access locality.
  – Tiled access patterns allow small cache/scartchpad memories to hold on to data for re-use.
  – For matrix multiplication, a 16X16 thread block perform 2*256 = 512 float loads from device memory for 256 * (2*16) = 8,192 mul/add operations.

• A convenient framework for organizing threads (tasks)
Increased Work per Thread for even more locality

- Each thread computes two elements of $P_{d_{sub}}$
- Reduced loads from global memory ($M_d$) to shared memory
- Reduced instruction overhead
  - More work done in each iteration
Double Buffering
- a frequently used algorithm pattern

- One could double buffer the computation, getting better instruction mix within each thread
  - This is classic software pipelining in ILP compilers

```c
Loop {
    Load current tile to shared memory
    syncthreads()
    Compute current tile
    syncthreads()
}
```

```c
Loop {
    Load next tile from global memory
    Deposit current tile to shared memory
    syncthreads()
    Load next tile from global memory
    Compute current tile
    syncthreads()
}
```
Double Buffering

- Deposit blue tile from register into shared memory
- Syncthreads
- Load orange tile into register
- Compute Blue tile
- Deposit orange tile into shared memory
- ....
(a) Direct summation
At each grid point, sum the electrostatic potential from all charges

(b) Cutoff summation
Electrostatic potential from nearby charges summed; spatially sort charges first

(c) Cutoff summation using direct summation kernel
Spatially sort charges into bins; adapt direct summation to process a bin
Cut-Off Summation Restores Data Scalability

Execution time (seconds)

Volume of potential map (Angstrom$^3$)

Same scalability among all cutoff implementations

Scalability and Performance of different algorithms for calculating electrostatic potential map.