VSCSE Summer School

Proven Algorithmic Techniques for Many-core Processors

Lecture 1: Introduction and Computational Thinking

Course Objective

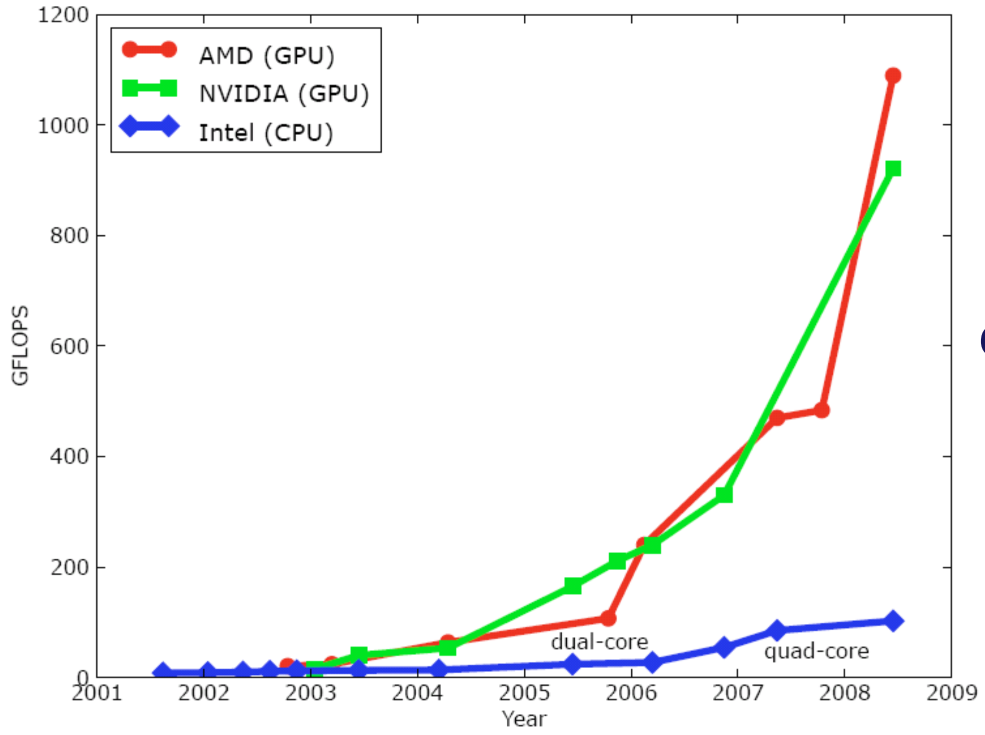
- To master the most commonly used algorithm techniques and computational thinking skills needed for many-core programming
  - Especially the simple ones!
- Specifically, to understand
  - Many-core hardware limitations and constraints
  - Desirable and undesirable computation patterns
  - Commonly used algorithm techniques to convert undesirable computation patterns into desirable ones

Course Staff

- Instructors
  - Wen-mei Hwu (UIUC), David Kirk (NVIDIA), John Stratton (UIUC), John Stone (UIUC)
- Keynote Speakers
  - Michael Garland (NVIDIA), David Kirk (NVIDIA), Lorena Barba (BU)
- Guest lectures
  - Jonathan Cohen (NVIDIA), Jeremy Meredith (ORNL)
- UIUC Teaching Assistants
  - Xiao-Long Wu, Deepthi Nandakumar, Liwen Chang, Hee-seok Kim, Nasser Anssari,
- Local Coordinators, Teaching Assistants, Technical Staff

Performance Advantage of GPUs

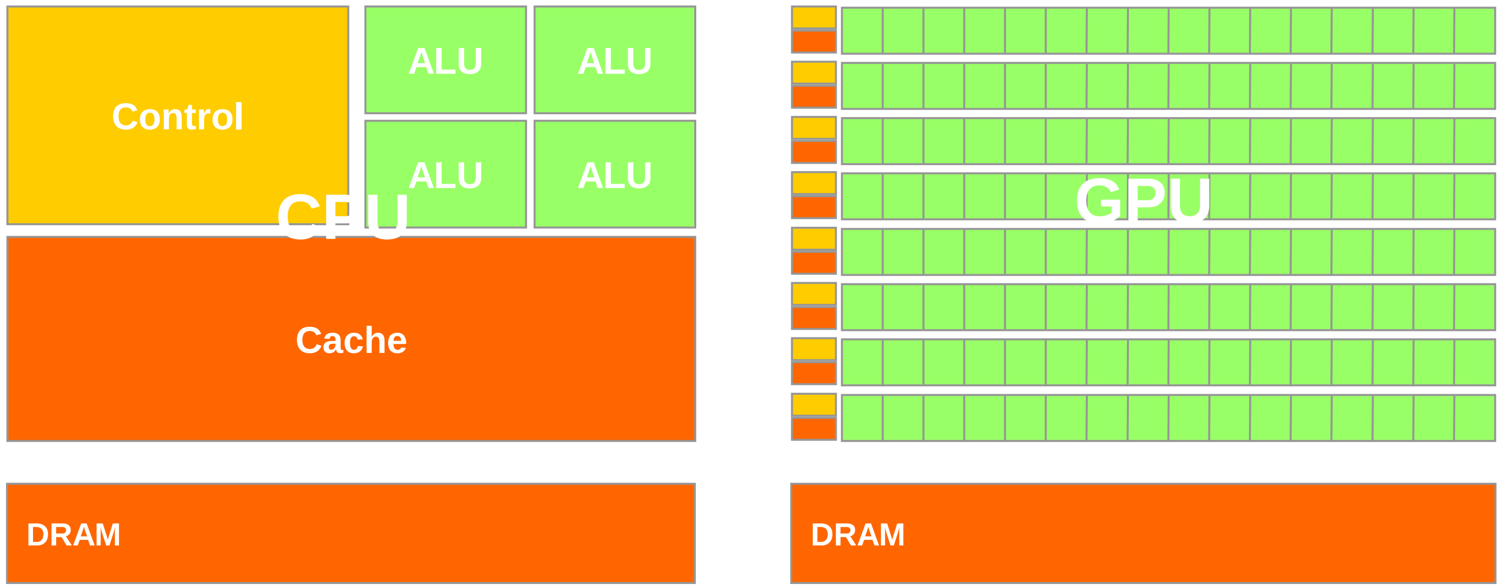
- An enlarging peak performance advantage:
  - Calculation: 1 TFLOPS vs. 100 GFLOPS
  - Memory Bandwidth: 100-150 GB/s vs. 32-64 GB/s



Courtesy: John Owens

  - GPU in every PC and workstation – massive volume and potential impact

CPUs and GPUs have fundamentally different design philosophies.



Harvesting Performance Benefit of Many-core GPU Requires

- Massive parallelism in application algorithms
  – Data parallelism
- Regular computation and data accesses
  – Similar work for parallel threads
- Avoidance of conflicts in critical resources
  – Off-chip DRAM (Global Memory) bandwidth
  – Conflicting parallel updates to memory locations

Massive Parallelism - Regularity

Main Hurdles to Overcome

- Serialization due to conflicting use of critical resources
- Over subscription of Global Memory bandwidth
- Load imbalance among parallel threads

“CUDA is an elegant solution to the problem of representing parallelism in algorithms, not all algorithms, but enough to matter.”

V. Natoli, Kudos for CUDA, HPCWire

Computational Thinking Skills

- The ability to translate/formulate domain problems into computational models that can be solved efficiently by available computing resources
  - Understanding the relationship between the domain problem and the computational models
  - **Understanding the strength and limitations of the computing devices**
  - **Designing the model implementations to steer away from the limitations**

DATA ACCESS CONFLICTS

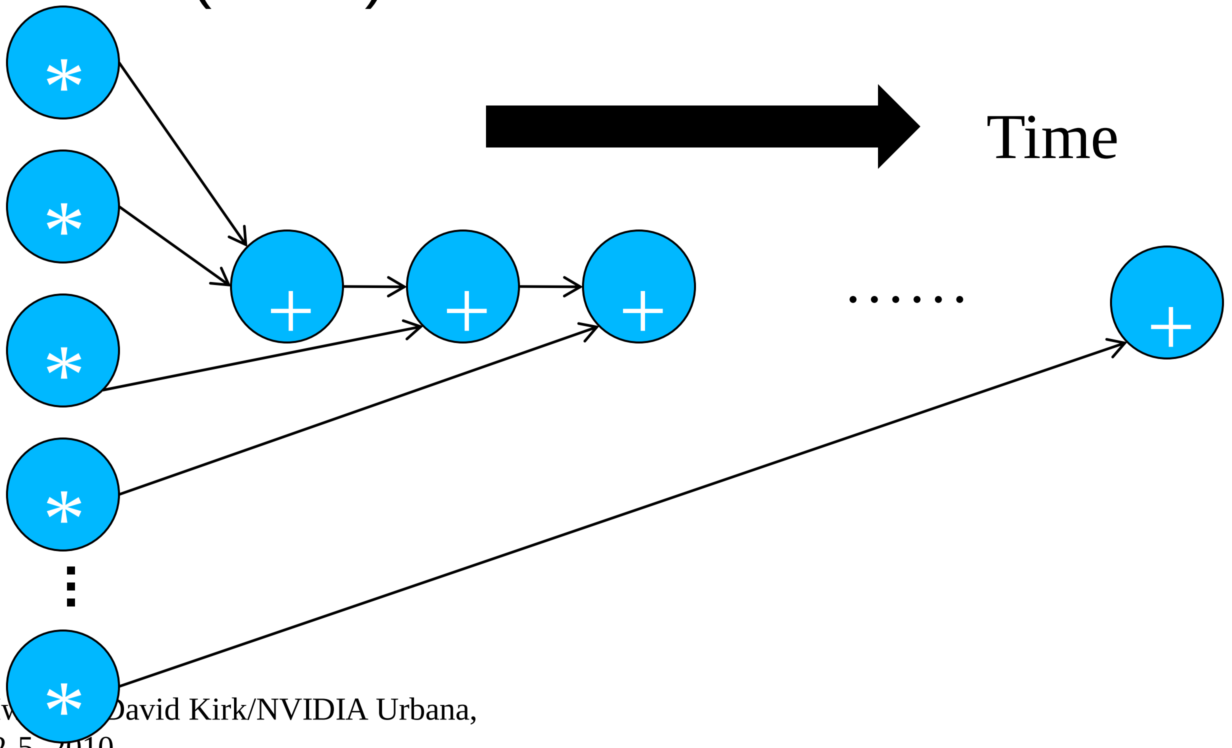
Conflicting Data Accesses Cause Serialization and Delays

- Massively parallel execution cannot afford serialization

- Contentions in accessing critical data causes serialization

A Simple Example

- A naïve inner product algorithm of two vectors of one million elements each
  - All multiplications can be done in time unit (parallel)
  - Additions to a single accumulator in one million time units (serial)



How much can conflicts hurt?

- Amdahl's Law
  - If fraction X of a computation is serialized, the speedup can not be more than 1/(1-X)
- In the previous example, X = 50%
  - Half the calculations are serialized
  - No more than 2X speedup, no matter how many computing cores are used

GLOBAL MEMORY BANDWIDTH

Global Memory Bandwidth

Ideal

Reality

Global Memory Bandwidth

- Many-core processors have limited off-chip memory access bandwidth compared to peak compute throughput
- GT200
  - 1 TFLOPS peak single precision peak throughput
  - 150 GB/s peak off-chip memory access bandwidth or 37.5 G single-precision operands per second
  - To achieve peak throughput, a program must perform 1,000/37.5 = 27 floating point arithmetic operations for each operand value fetched from off-chip memory,

Global Memory Bandwidth

- Fermi
  - 1 TFLOPS SPFP peak throughput
  - 0.5 TFLOPS DPFP peak throughput
  - 144 GB/s peak off-chip memory access bandwidth
    - 36 G SPFP operands per second
    - 18 G DPFP operands per second
  - To achieve peak throughput, a program must perform 1,000/36 = ~28 SPFP (14 DPFP) arithmetic operations for each operand value fetched from off-chip memory

A Simple CUDA Kernel for Matrix Multiplication

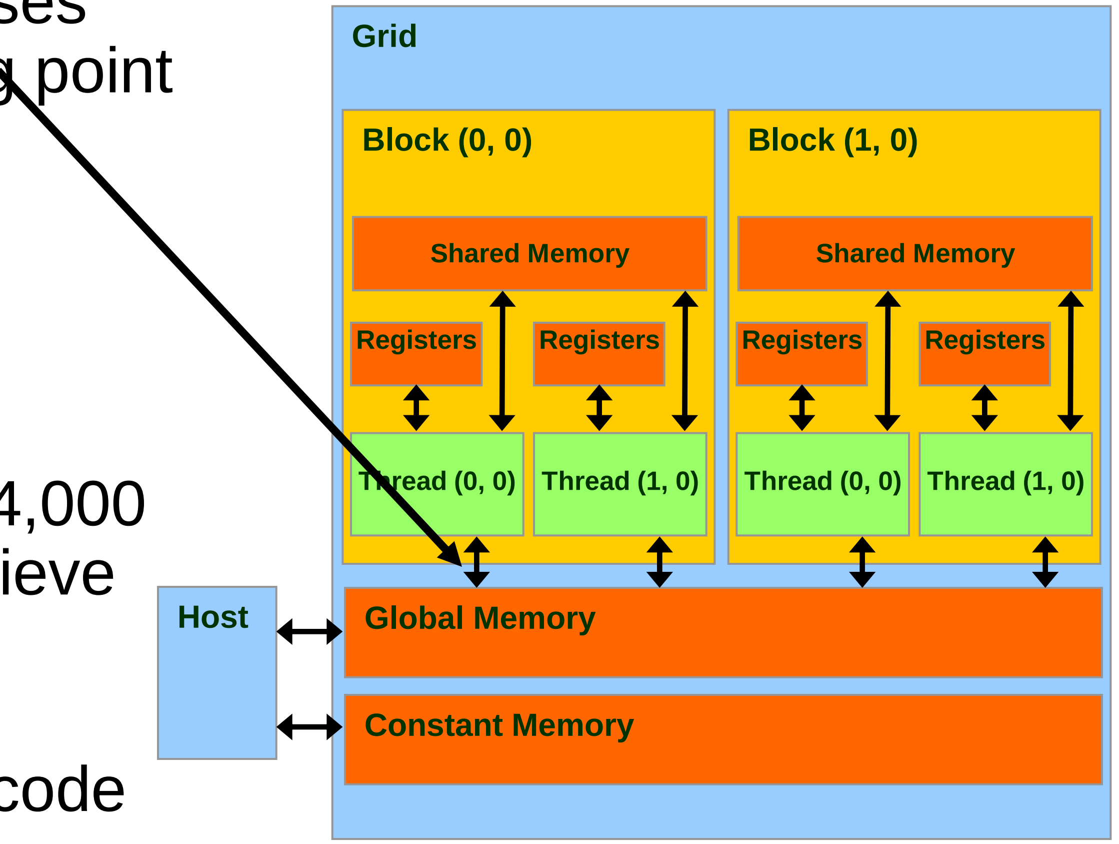
```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column idenx of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

float Pvalue = 0;
// each thread computes one element of the block sub-matrix
for (int k = 0; k < Width; ++k)
    Pvalue += Md[Row][k] * Nd[k][Col];

Pd[Row][Col] = Pvalue;
}
```

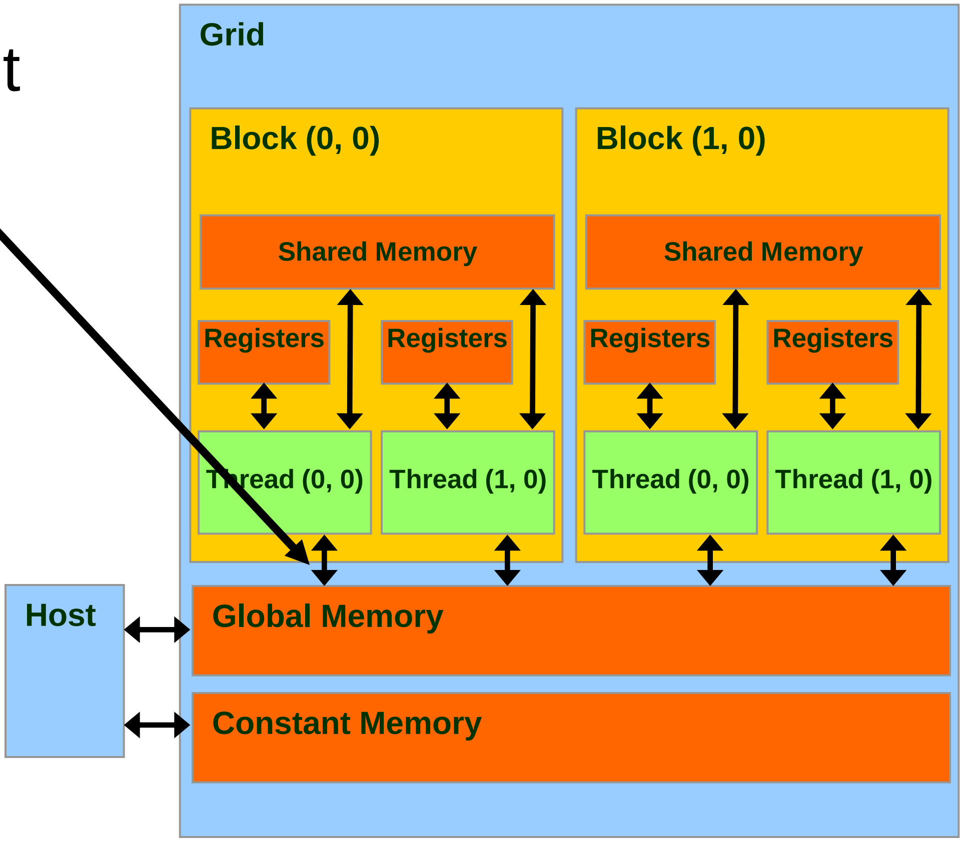
How about performance on GT200?

- Two memory accesses (8 bytes) per floating point multiply-add
- 4B/s of memory bandwidth/FLOPS
- 4*1,000GFLOPS = 4,000 GB/s needed to achieve peak FLOP rating
- 150 GB/s limits the code at 37.5 GFLOPS



How about performance on Fermi?

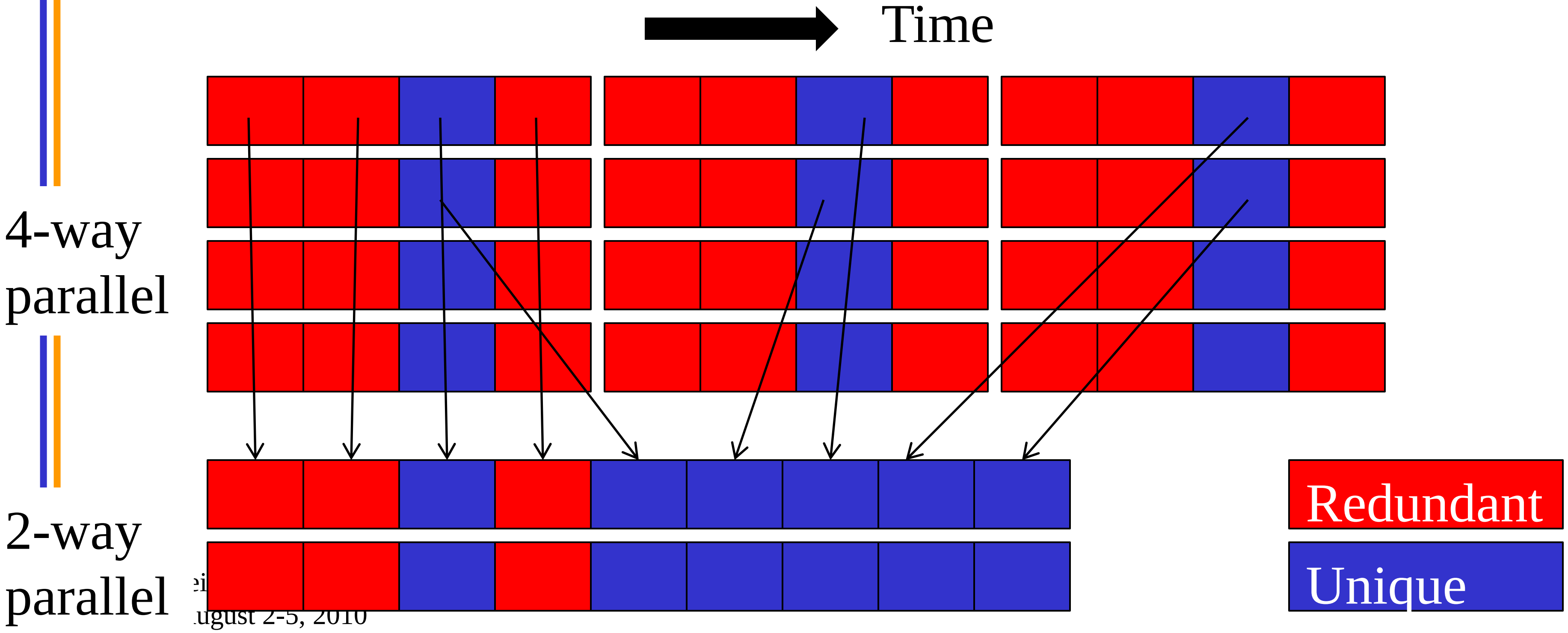
- Two memory accesses (8 bytes) per floating point multiply-add
- 4B/s of memory bandwidth/FLOPS
- 4*1,000GFLOPS = 4,000 GB/s needed to achieve peak SP FLOP rating
- 8*500GFLOPS = 4,000 GB/s needed to achieve peak DP FLOP rating
- 144 GB/s limits the code at 36 SP / 18 DP GFLOPS



REDUCING REDUNDANCY

Computational Efficiency

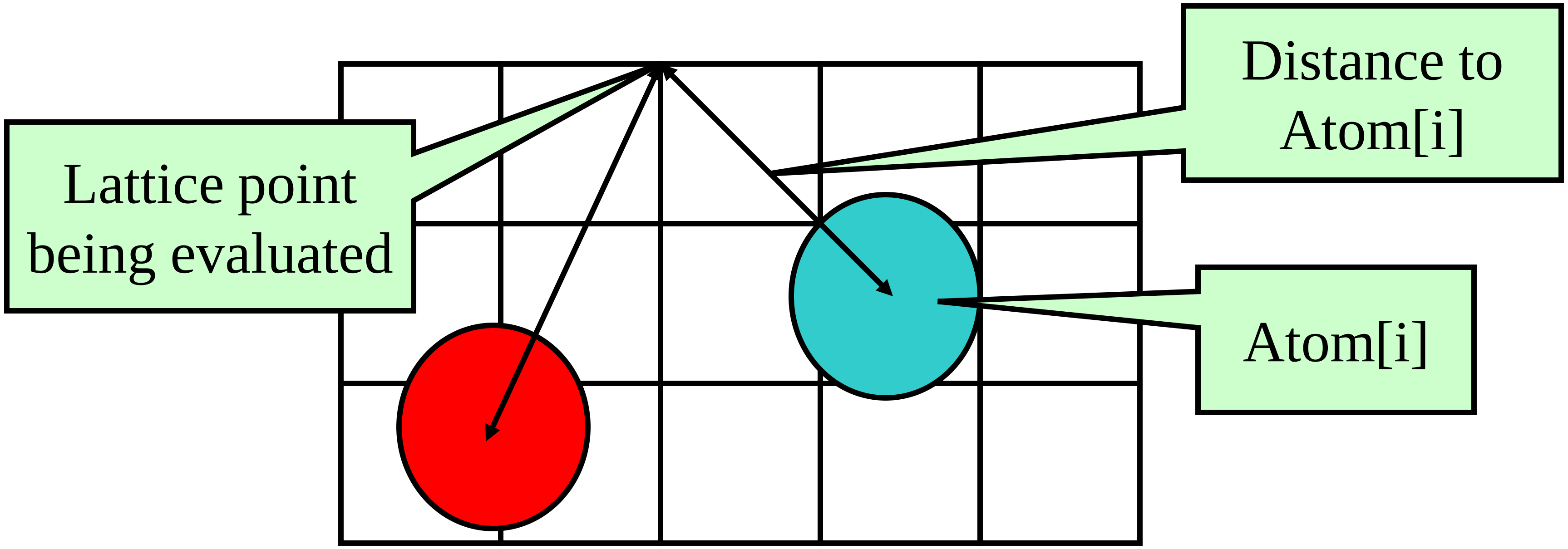
- Parallel execution sometime requires doing redundant work
- Total parallel execution may result in too much redundant work and longer execution



Direct Coulomb Summation (DCS) Where Redundancy Originates

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

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



DCS Initial 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;
```

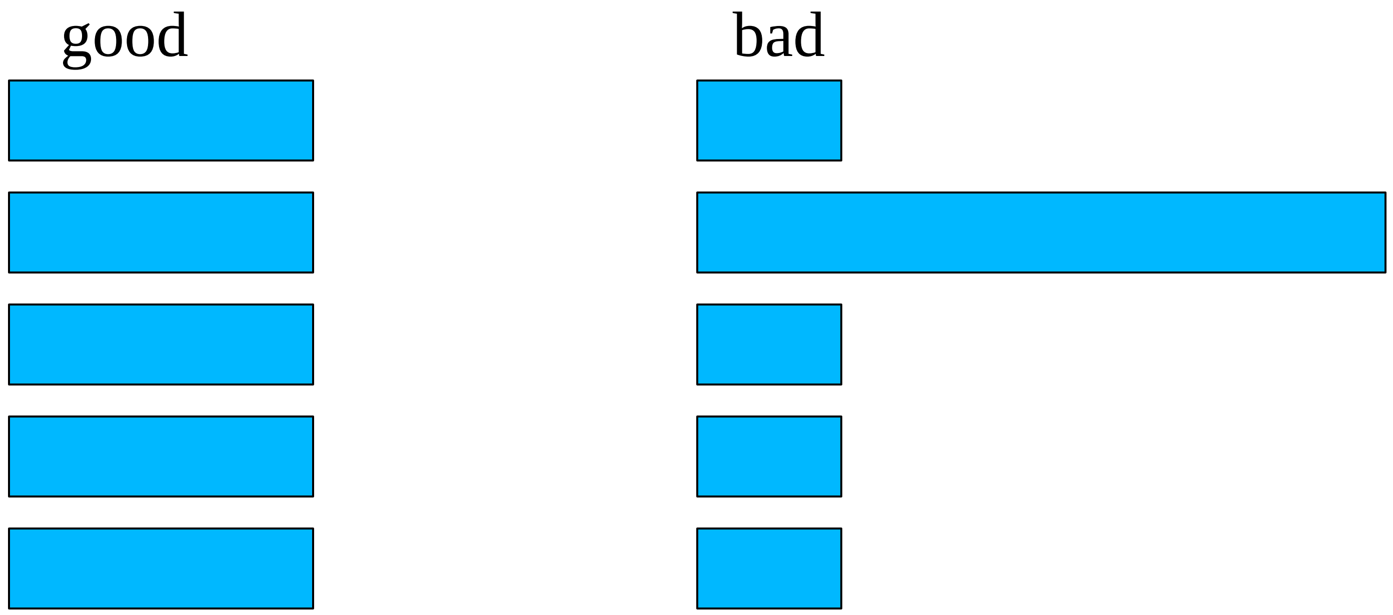
Implementation Analysis

- Parallelism: each potential lattice point is processed in a different thread
- Redundancy: distance calculation (dx^2+dy^2+dz^2)
  - In this case, one kernel processes all elements of a fixed z coordinate, a 2D “slice” of the full 3D lattice
  - dz^2 is precomputed for every atom
- The primary bottleneck of the first kernel is instruction issue
  - Too many instructions for indexes and counting loops
  - Too much redundant computation among threads in x and y

LOAD BALANCE

Load Balance

- The total amount of time to complete a parallel job is limited by the thread that takes the longest to finish

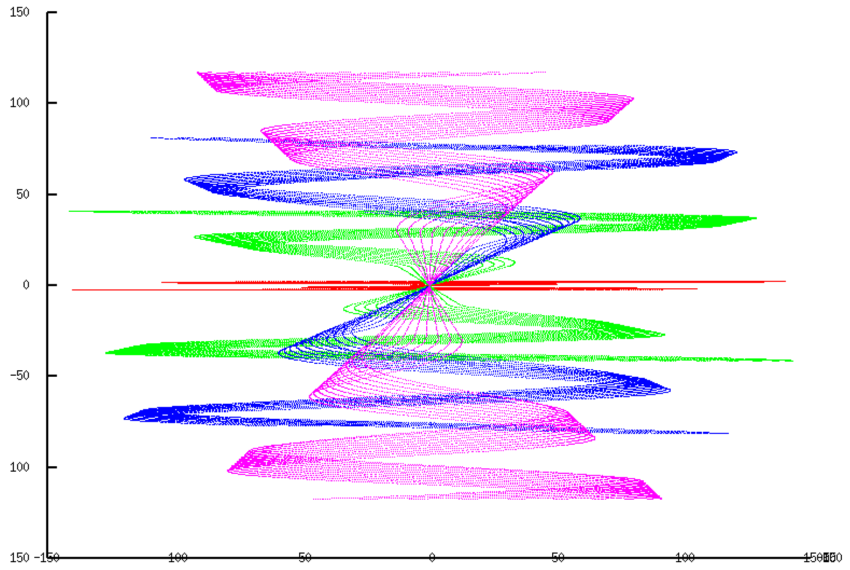


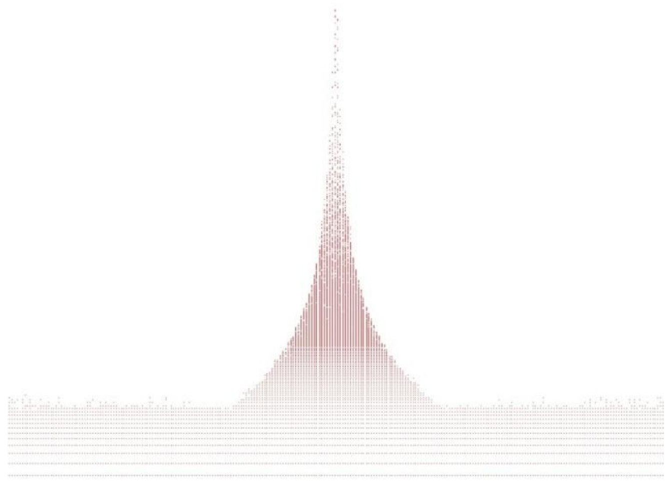
How bad can it be?

- Assume that a job takes 100 units of time for one person to finish
  - If we break up the job into 10 parts of 10 units each and have fo10 people to do it in parallel, we can get a 10X speedup
  - If we break up the job into 50, 10, 5, 5, 5, 5, 5, 5, 5, 5 units, the same 10 people will take 50 units to finish, with 9 of them idling for most of the time. We will get no more than 2X speedup.

How does imbalancing come about?

- Non-uniform data distributions
  - Highly concentrated spatial data areas
  - Astronomy, medical imaging, computer vision, rendering, …
- If each thread processes the input data of a given spatial volume unit, some will do a lot more work than others





Known Algorithm Techniques

- Increasing locality in dense arrays
  – tiling of data access and layout
- Improving efficiency and vectorization in dense arrays
  – granularity coarsening
- Reducing output interference
  – conversion from scatter to gather
  – parallelizing reductions and histograms

Known Algorithm Techniques (cont.)

- Dealing with non-uniform data
  - data sorting and binning
- Dealing with sparse data
  - sorting and packing
- Dealing with dynamic data
  - parallel queue-based algorithms
- Improving data efficiency in large data traversal
  - stencil and other grid-based computation

You can do it.

- Computational thinking is not as hard as you may think it is.
  - Most techniques have been explained, if at all, at the level of computer experts.
  - The purpose of the course is to make them accessible to domain scientists and engineers.

Agenda (Monday)

- 10:00 – 11:30 Lecture 1 – Introduction to computational thinking for many-core computing
- 12:30 – 2:00 Lecture 2 – Scatter-to-Gather transformation for scalability
- 3:00 – 4:30 Lecture 3 - Loop blocking and register tiling for locality

- Lab 1 – 2D Blocking and Register tiling

Agenda (Tuesday)

- Lecture 4 – Cut-off and Binning for regular data sets
- Lecture 5 – Data Layout for Grid Applications
- Keynote 1 – Michael Garland (NVIDIA)
  - Algorithm Design for GPU Computing

- Lab 1 – 2D Blocking and Register tiling
- Lab 2 - Data Layout Transformation for LBM Methods and Binning for Regular Data Sets

Agenda (Wednesday)

- Lecture 6 – Dealing with non-uniform data distribution
- Lecture 7 – Dealing with dynamic data sets
  – With guest lecture by Jonathan Cohen (NVIDIA) on PDE Solvers and OpenCurrent
- Keynote 2 – David Kirk (NVIDIA)
  – Fermi and Future of GPU Computing Technology

- Lab 2 - Data Layout Transformation for LBM Methods and Binning for Regular Data Sets
- Lab 3 – Binning for non-uniform data distribution

Agenda (Thursday)

- Keynote 3 – Lorena Barba (Boston University)
  - Multiplying speedups: fast algorithms on GPUs. A case study on biomolecular electrostatics.
- Lecture 8 – Transformations of key computation patterns
  - With guest lecture from Jeremy Meredith (Oak Ridge National Lab) on Accelerating HPC Applications with GPUs – Two Case Studies"
- Hands-on Lab 3 – Data binning for non-uniform data distribution
- Hands-on Lab wrap-up discussions

Agenda (Friday)

- Lecture 9: Directions for Further Studies
- Closing remarks

- Certificate and student feedback online survey

ANY MORE QUESTIONS?