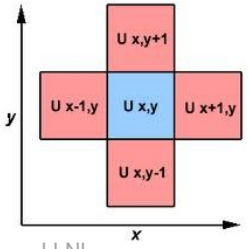
An Efficient CUDA Implementation of a Tree-Based N-Body Algorithm

Martin Burtscher
Department of Computer Science
Texas State University-San Marcos

Mapping Regular Code to GPUs

- Regular codes
  - Operate on array- and matrix-based data structures
  - Exhibit mostly strided memory access patterns
  - Have relatively predictable control flow (control flow behavior is largely determined by input *size*)

- Many regular codes are easy to port to GPUs
  - E.g., matrix codes executing many ops/word
    - Dense matrix operations (level 2 and 3 BLAS)
    - Stencil codes (PDE solvers)



LLNL

Mapping Irregular Code to GPUs

- Irregular codes
  - Build, traverse, and update dynamic data structures (e.g., trees, graphs, linked lists, and priority queues)
  - Exhibit data dependent memory access patterns
  - Have complex control flow (control flow behavior depends on input *values* and changes dynamically)

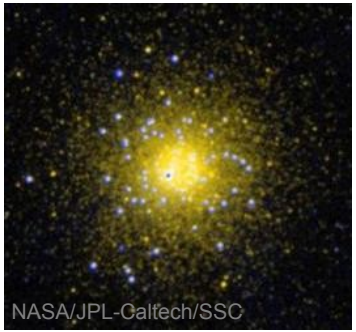
- Many important scientific programs are irregular
  - E.g., data clustering, SAT solving, social networks, meshing, …
- Need case studies to learn how to best map irregular codes

Example: N-Body Simulation

- Irregular Barnes Hut algorithm
  - Repeatedly builds unbalanced tree & performs complex traversals on it
- Our implementation
  - Designed for GPUs (not just port of CPU code)
  - First GPU implementation of entire BH algorithm
- Results
  - GPU is 21 times faster than CPU (6 cores) on this code
  - GPU has better architecture for this irregular algorithm
  - NVIDIA GPU Gems book chapter (2011)

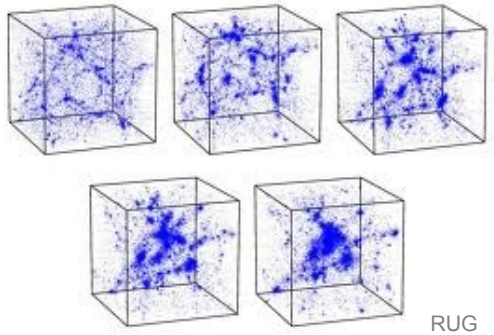
Outline

- Introduction
- Barnes Hut algorithm
- CUDA implementation
- Experimental results
- Conclusions



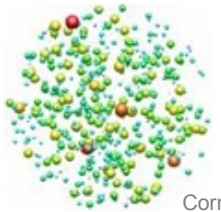
N-Body Simulation

- Time evolution of physical system
  - System consists of bodies
  - "n" is the number of bodies
  - Bodies interact via pair-wise forces



RUG

- Many systems can be modeled in this way
  - Star/galaxy clusters (gravitational force)
  - Particles (electric force, magnetic force)



Cornell

Simple $O(n^2)$ n-Body Algorithm

- Algorithm

  Initialize body masses, positions, and velocities

  Iterate over time steps {

  Accumulate forces acting on each body

  Update body positions and velocities based on force

  }

  Output result

- More sophisticated n-body algorithms exist
  - Barnes Hut algorithm
  - Fast Multipole Method (FMM)

Barnes Hut Idea

- Precise force calculation
  - Requires $O(n^2)$ operations ($O(n^2)$ body pairs)
  - Computationally intractable for large n

- Barnes and Hut (1986)
  - Algorithm to approximately compute forces
    - Bodies' initial position and velocity are also approximate
  - Requires only $O(n \log n)$ operations
  - Idea is to "combine" far away bodies
  - Error should be small because *force* is proportional to $1/distance^2$

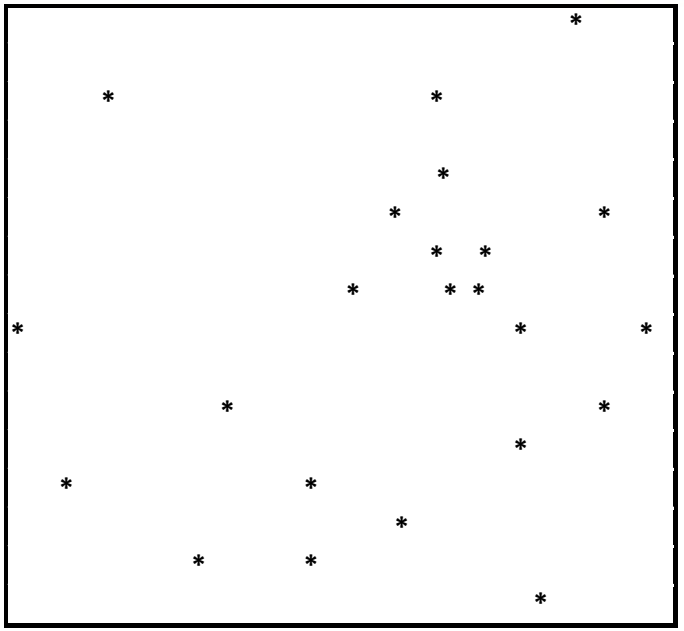
Barnes Hut Algorithm

- Set bodies' initial position and velocity
- Iterate over time steps
  1. Compute bounding box around bodies
  2. Subdivide space until at most one body per cell

     Record this spatial hierarchy in an octree
  3. Compute mass and center of mass of each cell
  4. Compute force on bodies by traversing octree

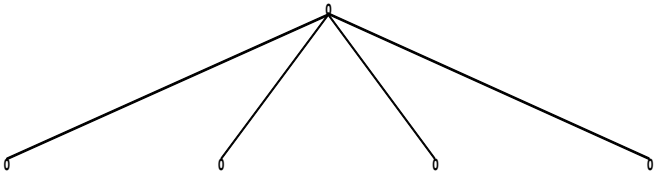
     Stop traversal path when encountering a leaf (body) or an internal node (cell) that is far enough away
  5. Update each body's position and velocity

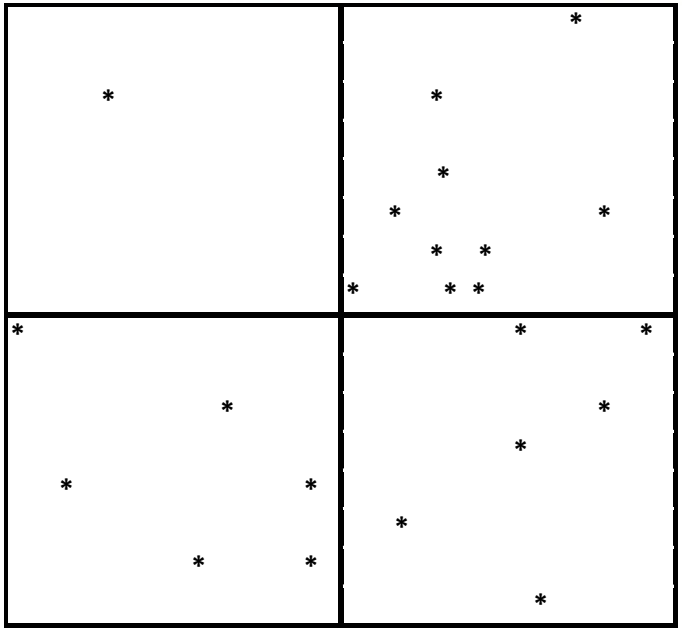
Build Tree (Level 1)



Compute bounding box around all bodies → tree root

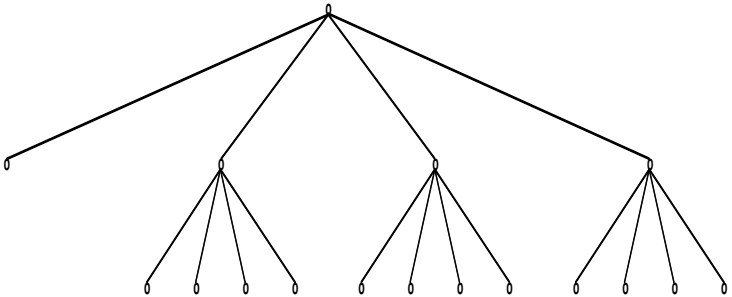
Build Tree (Level 2)

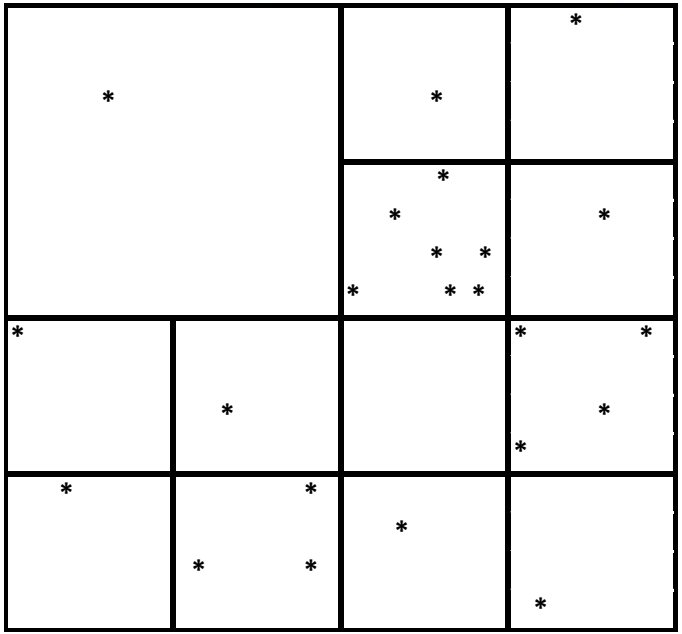




Subdivide space until at most one body per cell

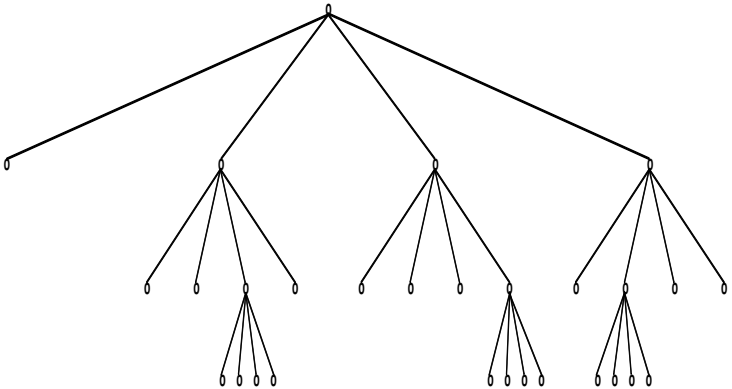
Build Tree (Level 3)





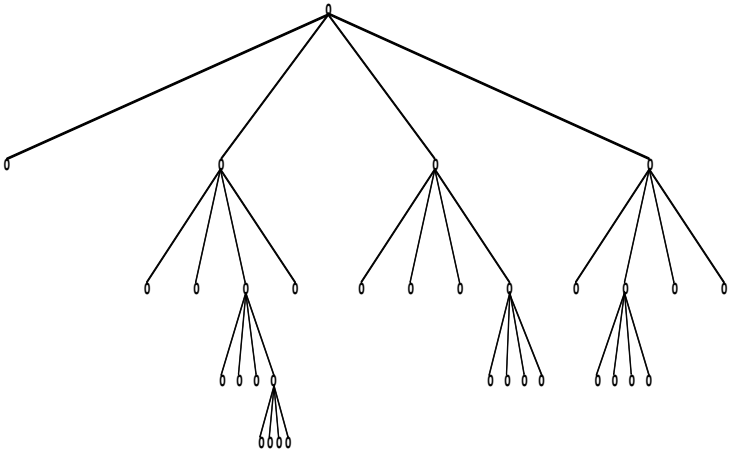
Subdivide space until at most one body per cell

Build Tree (Level 4)



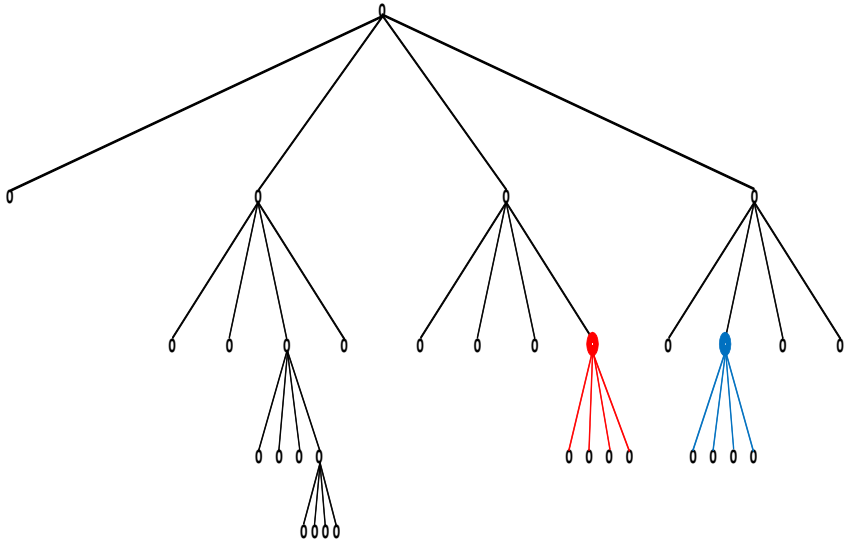
Subdivide space until at most one body per cell

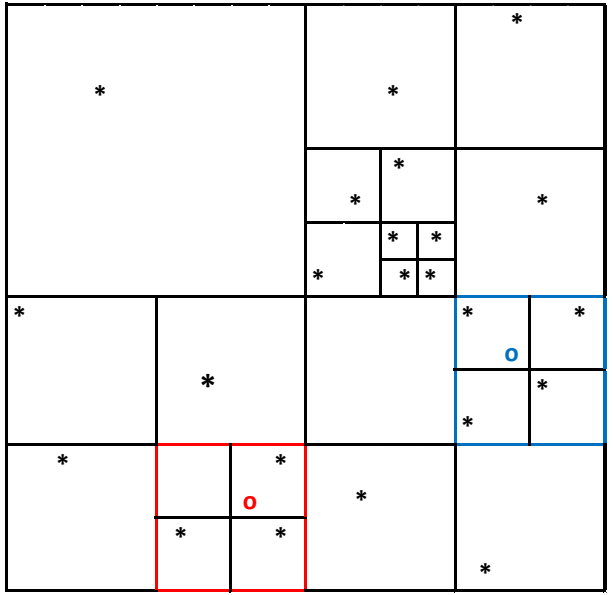
Build Tree (Level 5)



Subdivide space until at most one body per cell

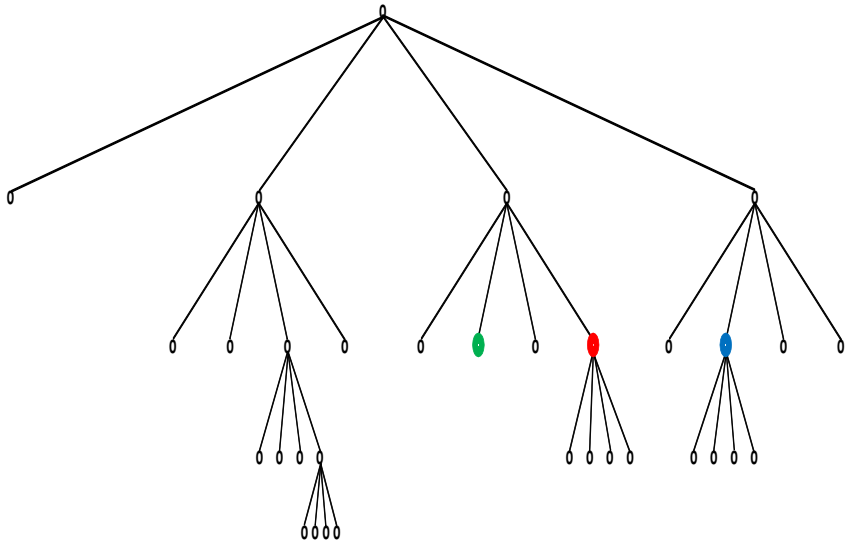
Compute Cells' Center of Mass

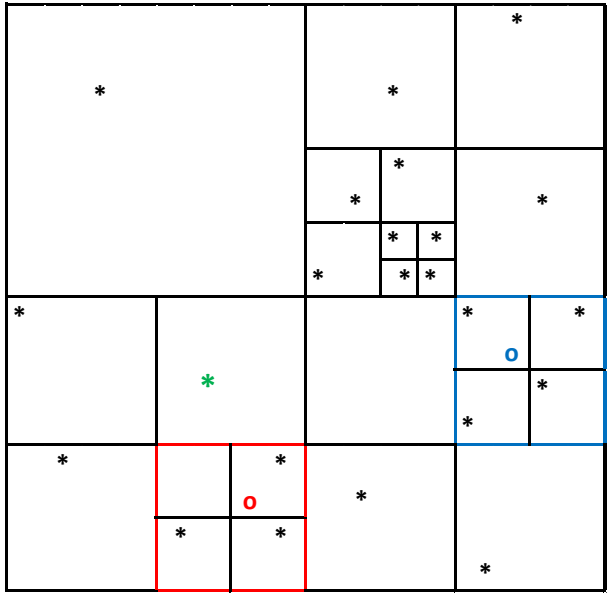




For each internal cell, compute sum of mass and weighted average of position of all bodies in subtree; example shows two cells only

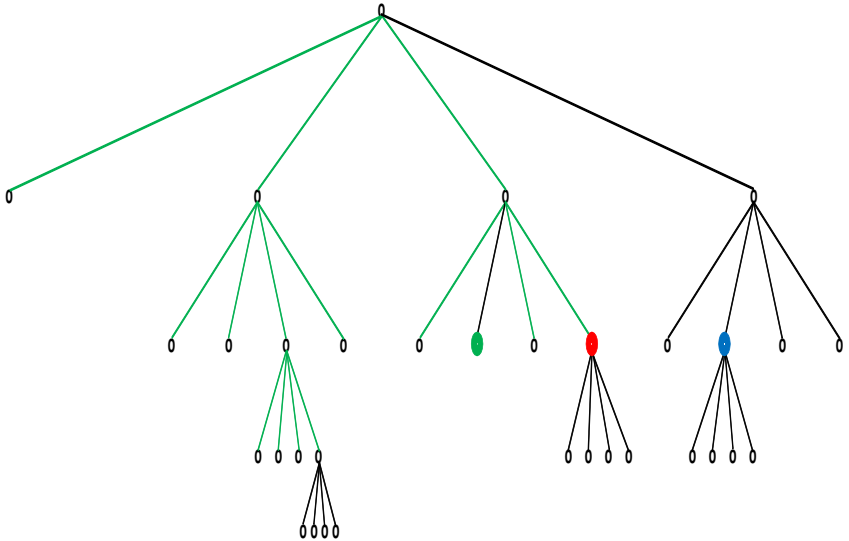
Compute Forces

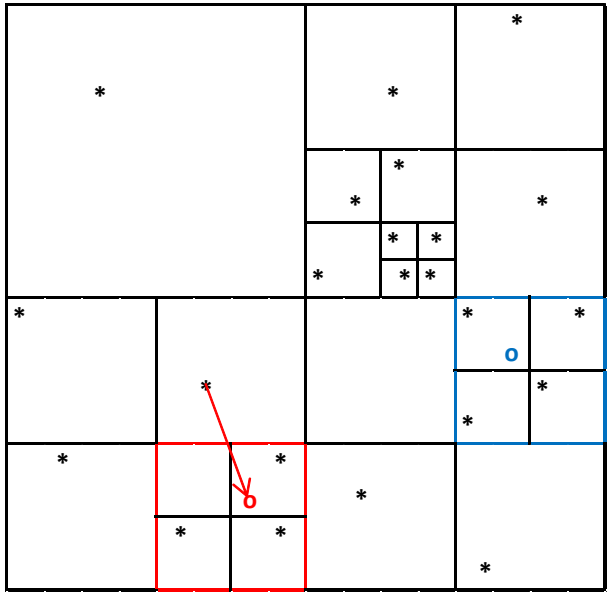




Compute force, for example, acting upon green body

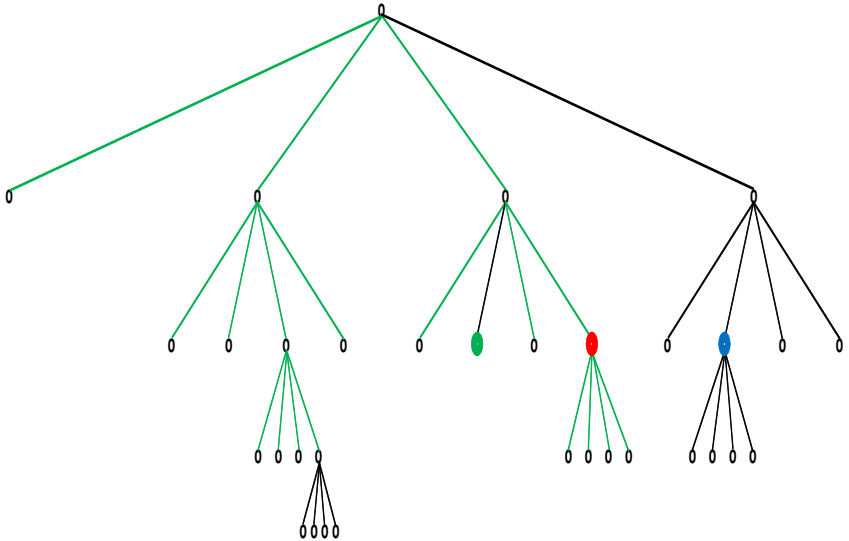
Compute Force (short distance)

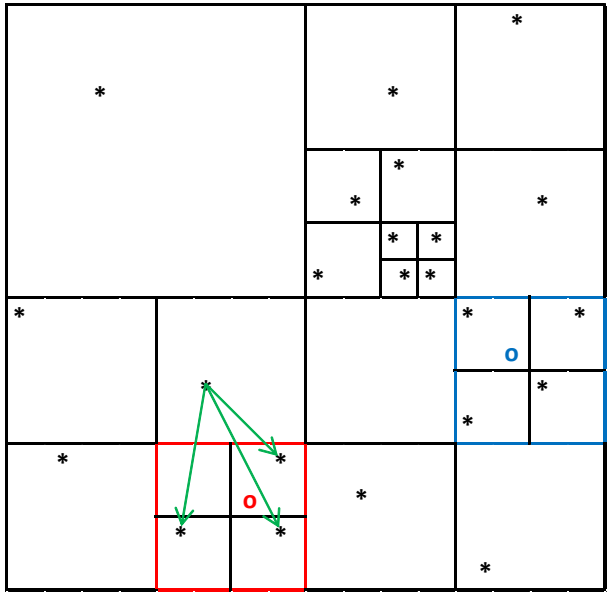




Scan tree depth first from left to right; green portion already completed

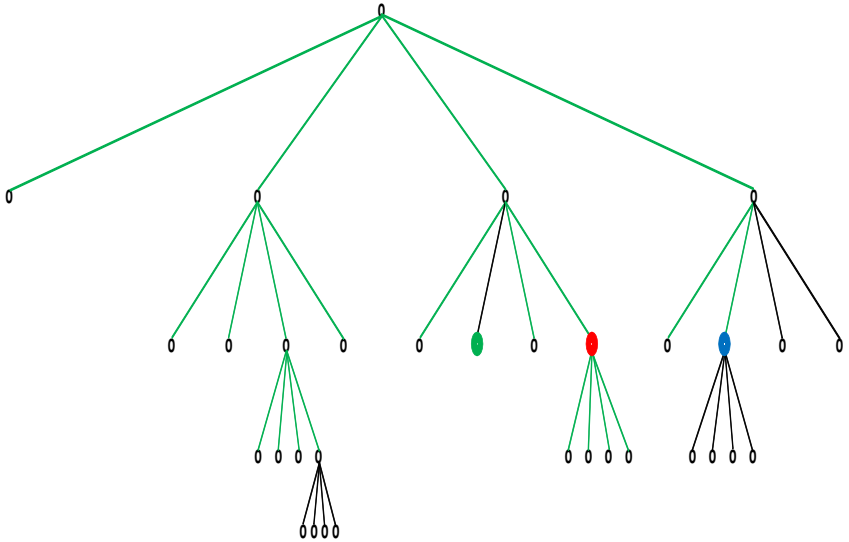
Compute Force (down one level)

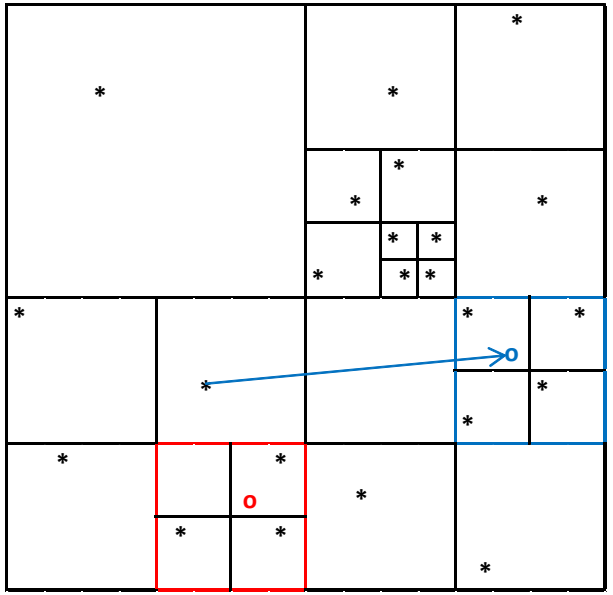




Red center of mass is too close, need to go down one level

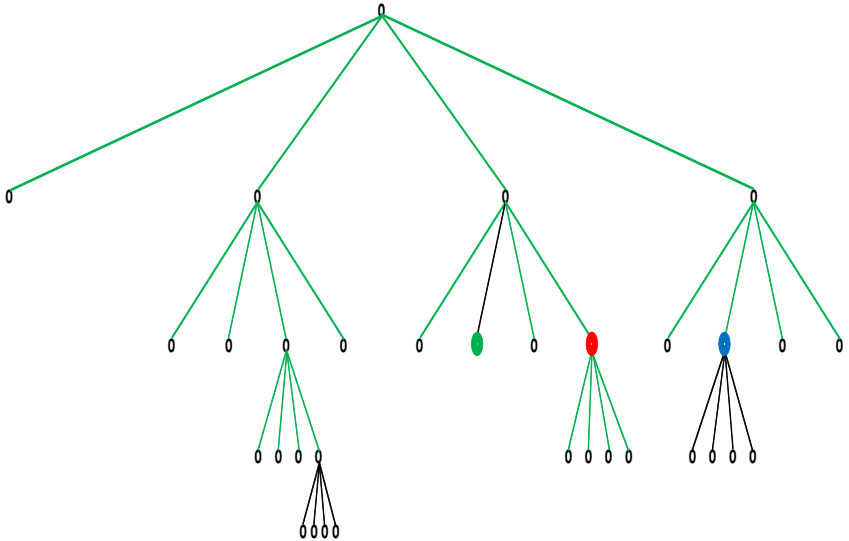
Compute Force (long distance)

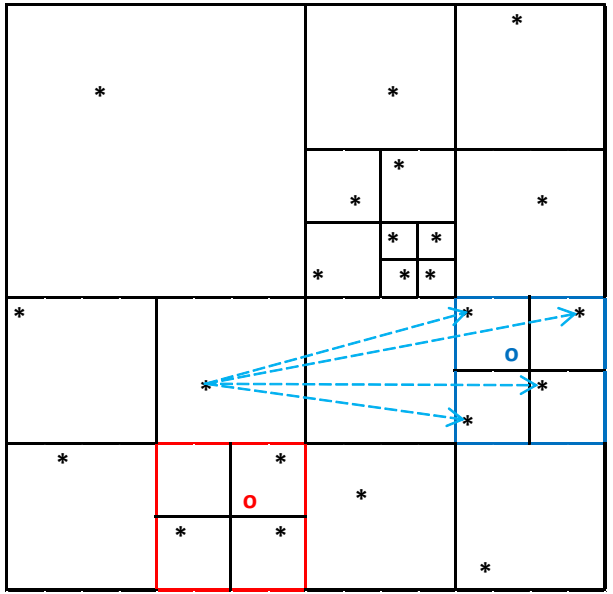




Blue center of mass is far enough away

Compute Force (skip subtree)

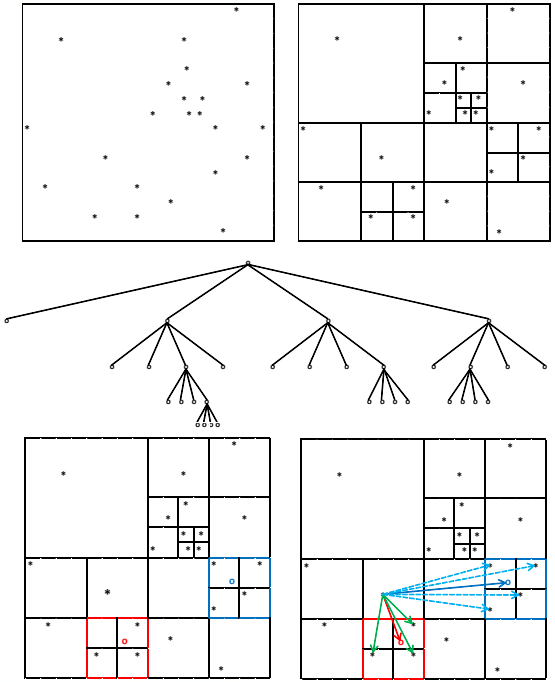




Therefore, entire subtree rooted in the blue cell can be skipped

Pseudocode

```
bodySet = ...
foreach timestep do {
  bounding_box = new Bounding_Box();
  foreach Body b in bodySet {
    bounding_box.include(b);
  }
  octree = new Octree(bounding_box);
  foreach Body b in bodySet {
    octree.Insert(b);
  }
  cellList = octree.CellsByLevel();
  foreach Cell c in cellList {
    c.Summarize();
  }
  foreach Body b in bodySet {
    b.ComputeForce(octree);
  }
  foreach Body b in bodySet {
    b.Advance();
  }
}
```



Complexity and Parallelism

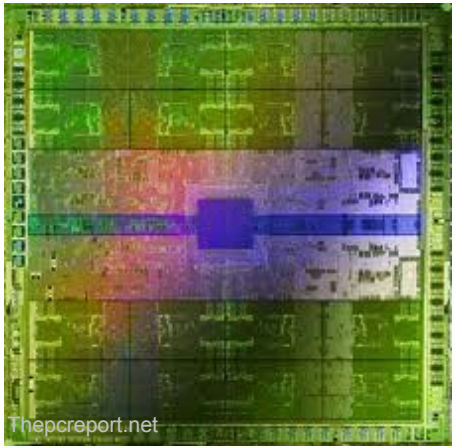
```
bodySet = ...
foreach timestep do {                  // O(n log n) + fully ordered sequential
  bounding_box = new Bounding_Box();
  foreach Body b in bodySet {          // O(n) parallel reduction
    bounding_box.include(b);
  }
  octree = new Octree(bounding_box);
  foreach Body b in bodySet {          // O(n log n) top-down tree building
    octree.Insert(b);
  }
  cellList = octree.CellsByLevel();
  foreach Cell c in cellList {         // O(n) + partially ordered bottom-up traversal
    c.Summarize();
  }
  foreach Body b in bodySet {          // O(n log n) fully parallel
    b.ComputeForce(octree);
  }
  foreach Body b in bodySet {          // O(n) fully parallel
    b.Advance();
  }
}
```

Outline

- Introduction
- Barnes Hut algorithm
- CUDA implementation
- Experimental results
- Conclusions

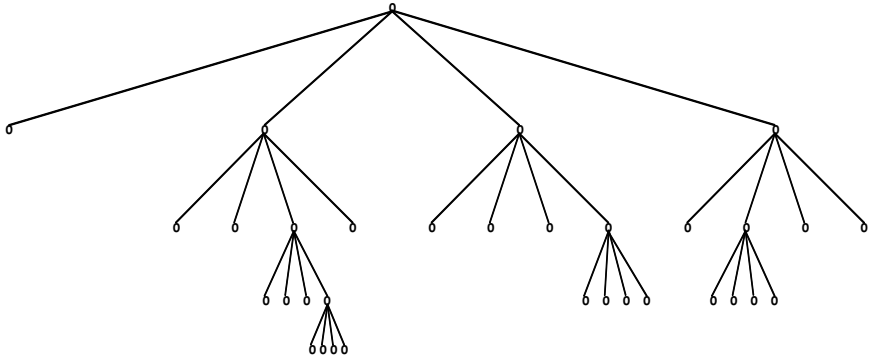
Efficient GPU Code

- Coalesced main memory accesses
- Little thread divergence
- Enough threads per block
  - Not too many registers per thread
  - Not too much shared memory usage
- Enough (independent) blocks
  - Little synchronization between blocks
- Little CPU/GPU data transfer
- Efficient use of shared memory



Main BH Implementation Challenges

- Uses irregular tree-based data structure
  - Load imbalance
  - Little coalescing
- Complex recursive traversals
  - Recursion not allowed*
  - Lots of thread divergence
- Memory-bound pointer-chasing operations
  - Not enough computation to hide latency



Six GPU Kernels

Read initial data and transfer to GPU

for each timestep do {

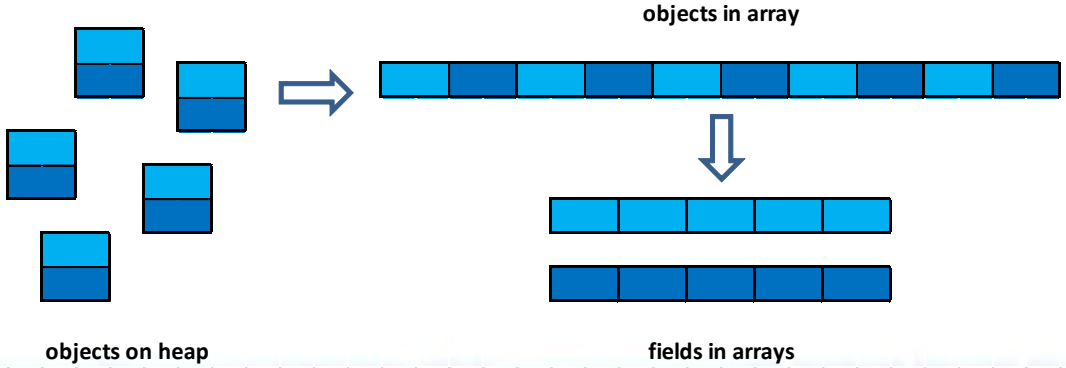
1. Compute bounding box around bodies
2. Build hierarchical decomposition, i.e., octree
3. Summarize body information in internal octree nodes
4. Approximately sort bodies by spatial location (optional)
5. Compute forces acting on each body with help of octree
6. Update body positions and velocities

}

Transfer result from GPU and output

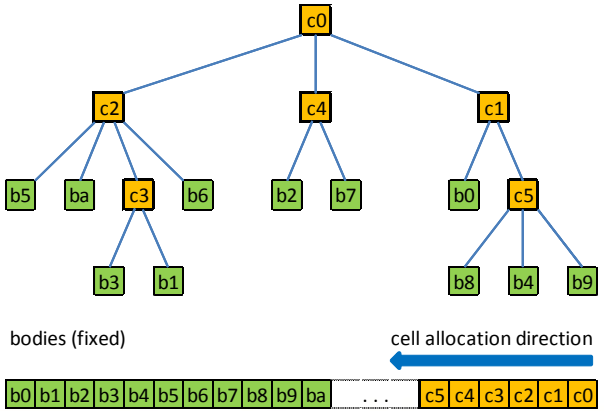
Global Optimizations

- Make code iterative (recursion not supported*)
- Keep data on GPU between kernel calls
- Use array elements instead of heap nodes
  - One aligned array per field for coalesced accesses

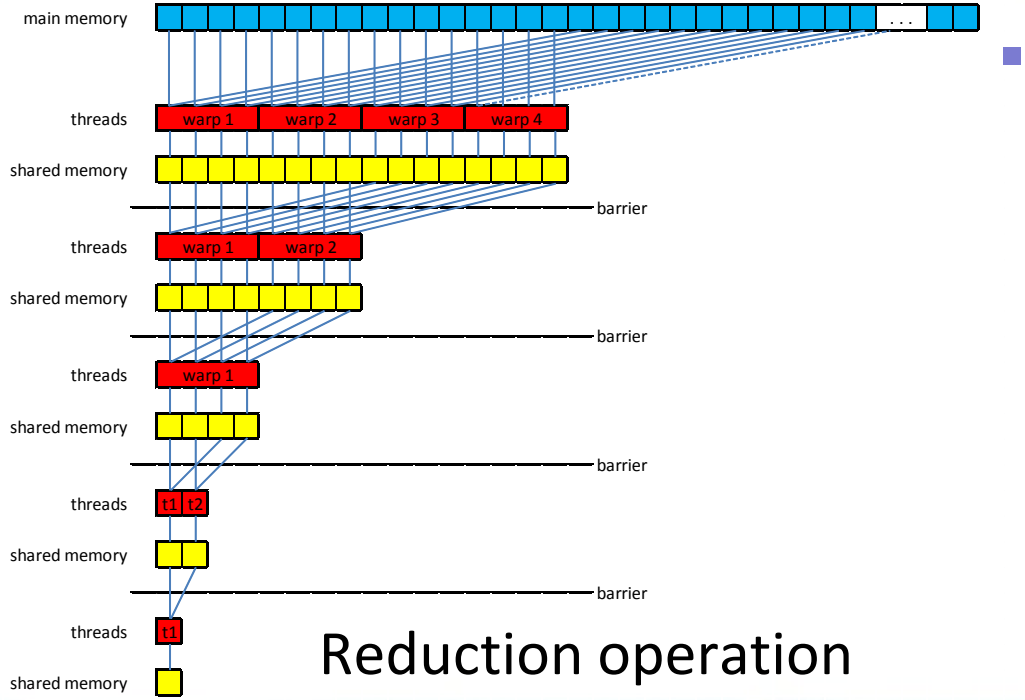


Global Optimizations (cont.)

- Maximize thread count (round down to warp size)
- Maximize resident block count (all SMs filled)
- Pass kernel parameters through constant memory
- Use special allocation order
- Alias arrays (56 B/node)
- Use index arithmetic
- Persistent blocks & threads
- Unroll loops over children



Kernel 1: Bounding Box (Regular)



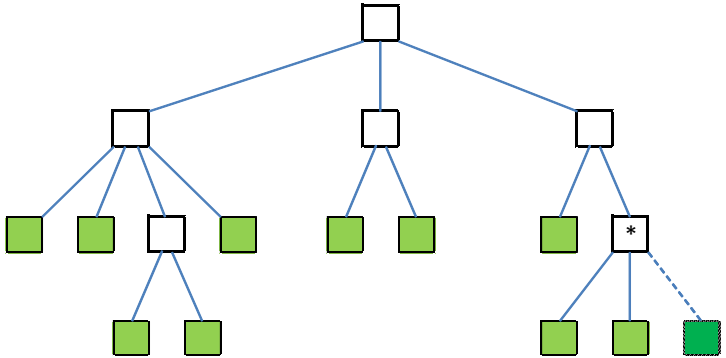
Reduction operation

- Optimizations
  - Fully coalesced
  - Fully cached
  - No bank conflicts
  - Minimal divergence
  - Built-in min and max
  - 2 red/mem, 6 red/bar
  - Bodies load balanced
  - 512*3 threads per SM

Kernel 2: Build Octree (Irregular)

- Optimizations
  - Only lock leaf “pointers”
  - Lock-free fast path
  - Light-weight lock release on slow path
  - No re-traverse after lock acquire failure
  - Combined memory fence per block
  - Re-compute position during traversal
  - Separate init kernels for coalescing
  - 512*3 threads per SM

Top-down tree building

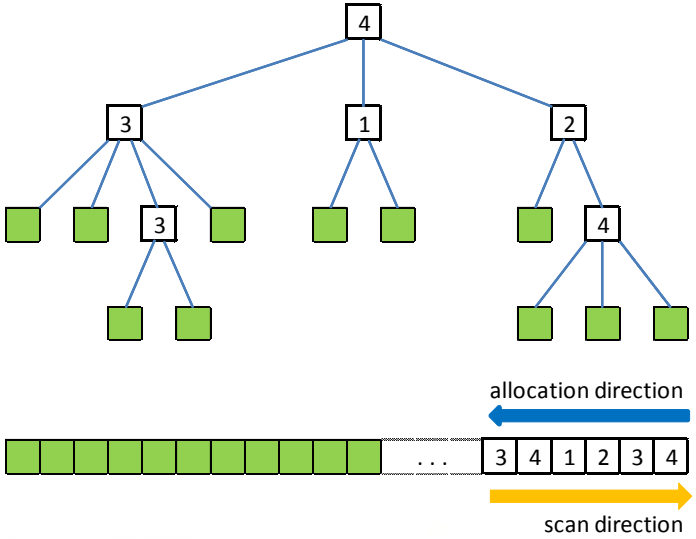


Kernel 2: Build Octree (cont.)

```
// initialize
cell = find_insertion_point(body);  // no locks, cache cell
child = get_insertion_index(cell, body);
if (child != locked) {  // skip atomic if already locked
  if (child == null) {  // fast path (frequent)
    if (null == atomicCAS(&cell[child], null, body)) {  // lock-free insertion
      // move on to next body
    }
  } else {  // slow path (first part)
    if (child == atomicCAS(&cell[child], child, lock)) {  // acquire lock
      // build subtree with new and existing body
      flag = true;
    }
  }
}
__syncthreads();  // barrier
if (threadIdx == 0) __threadfence();  // push data out if L1 cache
__syncthreads();  // barrier
if (flag) {  // slow path (second part)
  cell[child] = new_subtree;  // insert subtree and release lock
  // move on to next body
}
```

Kernel 3: Summarize Subtrees (Irregular)

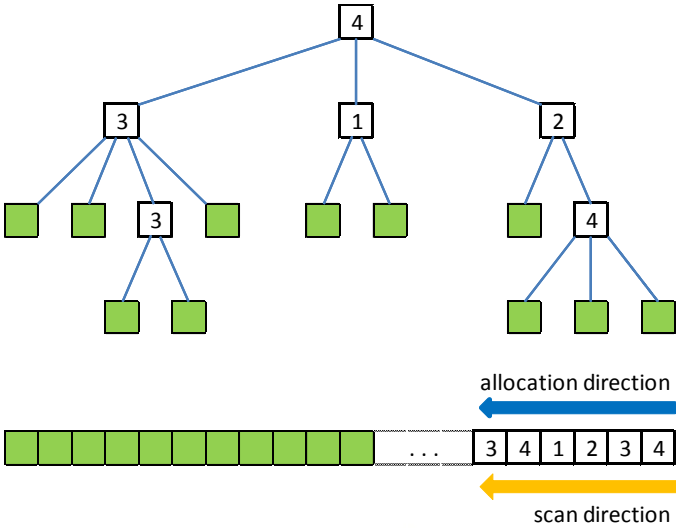
Bottom-up tree traversal



- Optimizations
  - Scan avoids deadlock
  - Use mass as flag + fence
    - No locks, no atomics
  - Use wait-free first pass
  - Cache the ready information
  - Piggyback on traversal
    - Count bodies in subtrees
  - No parent “pointers”
  - 128*6 threads per SM

Kernel 4: Sort Bodies (Irregular)

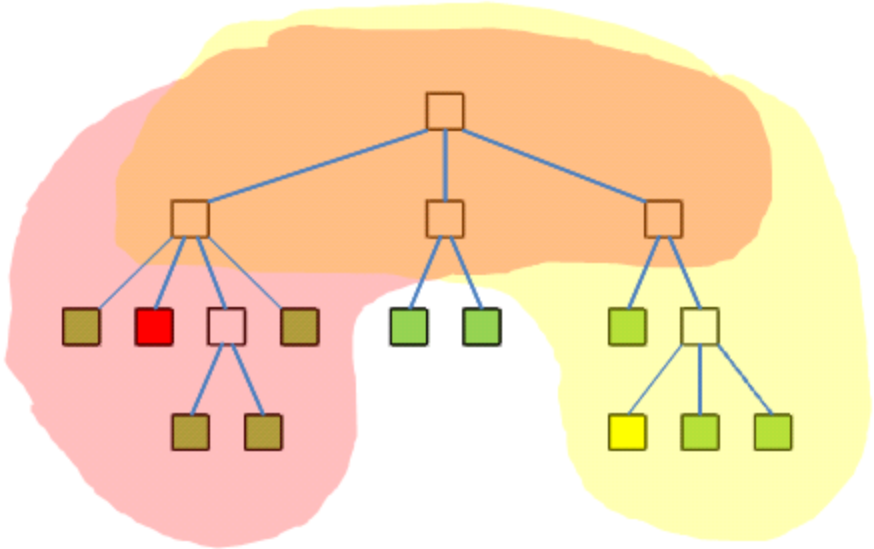
Top-down tree traversal



- Optimizations
  - (Similar to Kernel 3)
  - Scan avoids deadlock
  - Use data field as flag
    - No locks, no atomics
  - Use counts from Kernel 3
  - Piggyback on traversal
    - Move nulls to back
  - Throttle flag polling requests with *optional* barrier
  - 64*6 threads per SM

Kernel 5: Force Calculation (Irregular)

Multiple prefix traversals



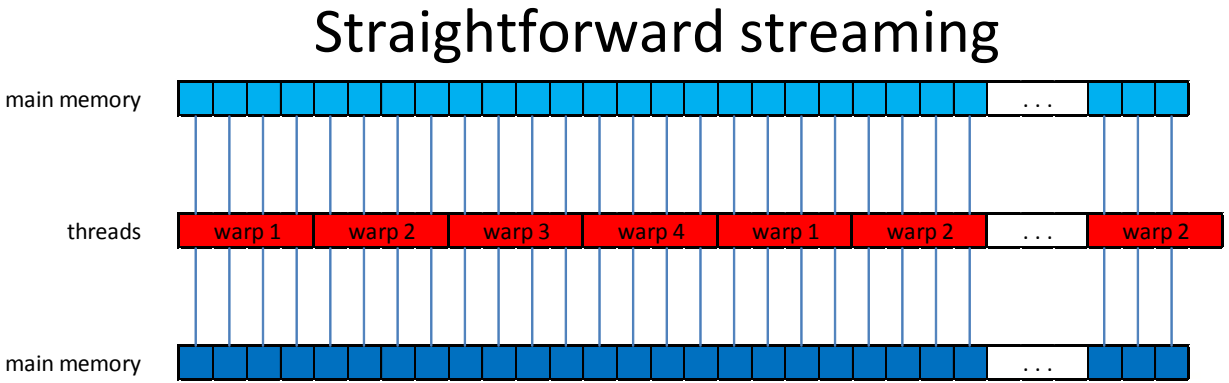
- Optimizations
  - Group similar work together
    - Uses sorting to minimize size of prefix union in each warp
    - Early out (nulls in back)
  - Traverse whole union to avoid divergence (warp voting)
  - Lane 0 controls iteration stack for entire warp (fits in shared mem)
  - Avoid unneeded volatile accesses
  - Cache tree-level-based data
  - 256*5 threads per SM

Architectural Support

- Coalesced memory accesses & lockstep execution
  - All threads in warp read same tree node at same time
  - Only one mem access per warp instead of 32 accesses
- Warp-based execution
  - Enables data sharing in warps without synchronization
- RSQRTF instruction
  - Quickly computes good approximation of 1/sqrtf(x)
- Warp voting instructions
  - Quickly perform reduction operations within a warp

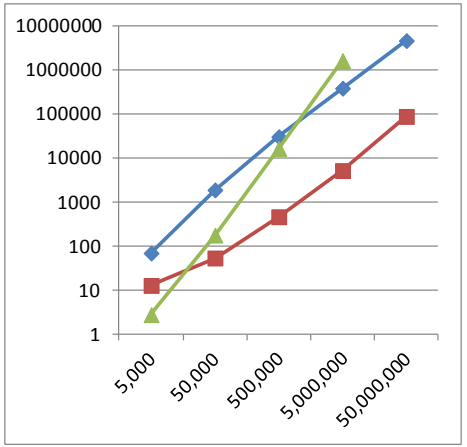
Kernel 6: Advance Bodies (Regular)

- Optimizations
  - Fully coalesced, no divergence
  - Load balanced, 1024*1 threads per SM



Outline

- Introduction
- Barnes Hut algorithm
- CUDA implementation
- Experimental results
- Conclusions



Evaluation Methodology

- Implementations
  - CUDA: irregular Barnes Hut & regular O(n^2) algorithm
  - OpenMP: Barnes Hut algorithm (derived from CUDA code)
  - Pthreads: Barnes Hut algorithm (from SPLASH-2 suite)
- Systems and compilers
  - nvcc 4.0 (-O3 -arch=sm_20 -ftz=true*)
    GeForce GTX 480, 1.4 GHz, 15 SMs, 32 cores per SM
  - gcc 4.1.2 (-O3 -fopenmp* -ffast-math*)
    Xeon X5690, 3.46 GHz, 6 cores, 2 threads per core
- Inputs and metric
  - 5k, 50k, 500k, and 5M star clusters (Plummer model)
  - Best runtime of three experiments, excluding I/O

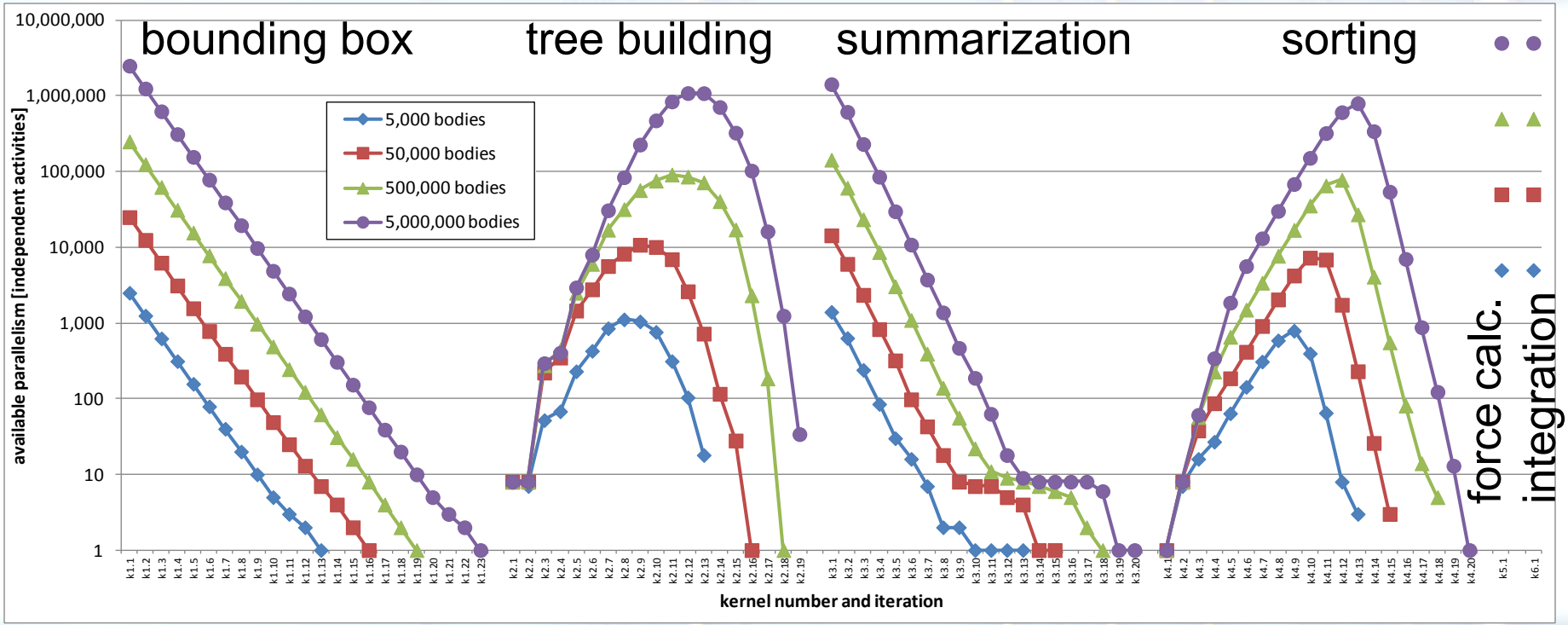
Nodes Touched per Activity (5M Input)

- Kernel “activities”
  - K1: pair reduction
  - K2: tree insertion
  - K3: bottom-up step
  - K4: top-down step
  - K5: prefix traversal
  - K6: integration step
- Max tree depth ≤ 22
- Cells have 3.1 children

	neighborhood size		
	min	avg	max
kernel 1	1	2.0	2
kernel 2	2	13.2	22
kernel 3	2	4.1	9
kernel 4	2	4.1	9
kernel 5	818	4,117.0	6,315
kernel 6	1	1.0	1

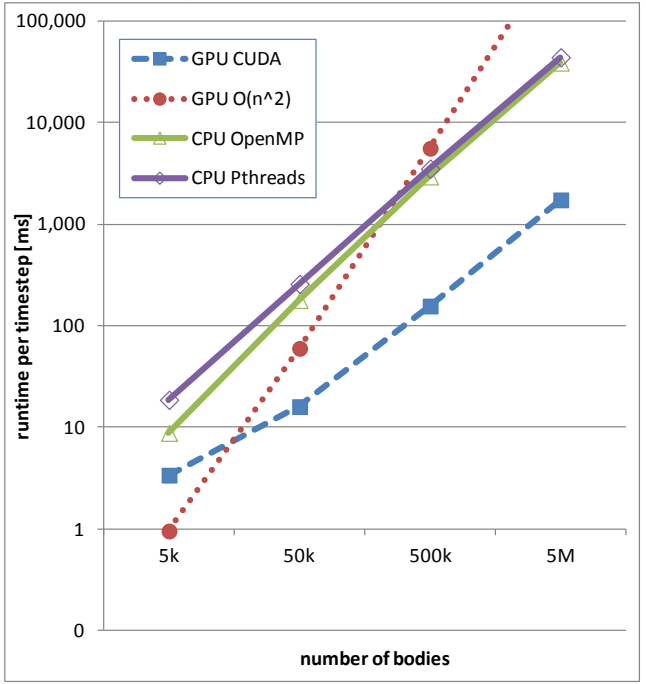
- Prefix ≤ 6,315 nodes (≤ 0.1% of 7.4M)
- BH algorithm & sorting to minimize size of prefix union work well

Available Amorphous Data Parallelism



Runtime Comparison

- GPU BH inefficiency
  - 5k input too small for 5,760 to 23,040 threads
- BH vs. O(n^2) algorithm
  - Regular O(n^2) code is faster with fewer than about 15,000 bodies
- GPU vs. CPU (5M input)
  - 21.1x faster than OpenMP
  - 23.2x faster than Pthreads



Kernel Performance for 5M Input

- $400 GPU delivers 228 GFlops/s on irregular code

	kernel 1	kernel 2	kernel 3	kernel 4	kernel 5	kernel 6	BarnesHut	O(n^2)
Gflops/s	71.6	5.8	2.5	n/a	240.6	33.5	228.4	897.0
GB/s	142.9	26.8	10.6	12.8	8.0	133.9	8.8	2.8
runtime [ms]	0.4	44.6	28.0	14.2	1641.2	2.2	1730.6	557421.5

- GPU chip is 2.7 to 23.5 times faster than CPU chip

	non-compliant fast single-precision version						IEEE 754-compliant double-precision version					
	kernel 1	kernel 2	kernel 3	kernel 4	kernel 5	kernel 6	kernel 1	kernel 2	kernel 3	kernel 4	kernel 5	kernel 6
X5690 CPU	5.5	185.7	75.8	52.1	38,540.3	16.4	10.3	193.1	101.0	51.6	47,706.4	33.1
GTX 480 GPU	0.4	44.6	28.0	14.2	1,641.2	2.2	0.8	46.7	31.0	14.2	7,714.6	4.2
CPU/GPU	13.1	4.2	2.7	3.7	23.5	7.3	12.7	4.1	3.3	3.6	6.2	7.9

- GPU hardware is better suited for running BH than CPU hw is
  - But more difficult and time consuming to program

Kernel Speedups

- Optimizations that are generally applicable

	avoid volatile	rsqrtf instr.	recalc. data	thread voting	full multi-threading
50,000	1.14x	1.43x	0.99x	2.04x	20.80x
500,000	1.19x	1.47x	1.32x	2.49x	27.99x
5,000,000	1.18x	1.46x	1.69x	2.47x	28.85x

- Optimizations for irregular kernels

	throttling barrier	waitfree pre-pass	combined mem fence	sorting of bodies	sync'ed execution
50,000	0.97x	1.02x	1.54x	3.60x	6.23x
500,000	1.03x	1.21x	1.57x	6.28x	8.04x
5,000,000	1.04x	1.31x	1.50x	8.21x	8.60x

Outline

- Introduction
- Barnes Hut algorithm
- CUDA implementation
- Experimental results
- Conclusions

Optimization Summary

- Minimize thread divergence
  - Group similar work together, force synchronicity
- Reduce main memory accesses
  - Share data within warp, combine memory fences & traversals, re-compute data, avoid volatile accesses
- Implement entire algorithm on and for GPU
  - Avoid data transfers & data structure inefficiencies, wait-free pre-pass, scan entire prefix union

Optimization Summary (cont.)

- Exploit hardware features
  - Fast synchronization and thread startup, special instructions, coalesced memory accesses, even lockstep execution
- Use light-weight locking and synchronization
  - Minimize locks, reuse fields, use fence + store ops
- Maximize parallelism
  - Parallelize every step within and across SMs

Conclusions

- Irregularity does not necessarily prevent high-performance on GPUs
  - Entire Barnes Hut algorithm implemented on GPU
    - Builds and traverses unbalanced octree
  - GPU is 22.5 times (float) and 6.2 times (double) faster than high-end hyperthreaded 6-core Xeon
- Code directly for GPU, do not merely adjust CPU code
  - Requires different data and code structures
  - Benefits from different algorithmic modifications

Acknowledgments

- Collaborators
  - Ricardo Alves (Universidade do Minho, Portugal)
  - Molly O'Neil (Texas State University-San Marcos)
  - Keshav Pingali (University of Texas at Austin)

- Hardware and funding
  - NVIDIA Corporation