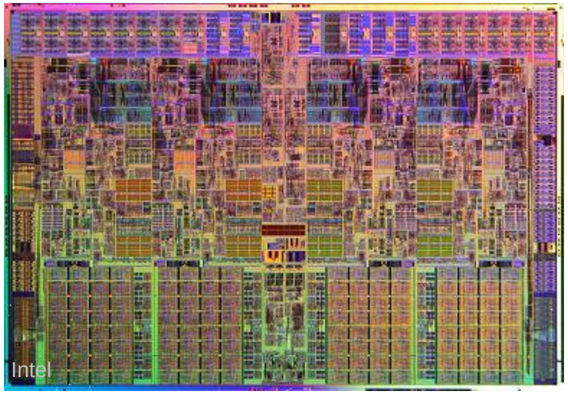
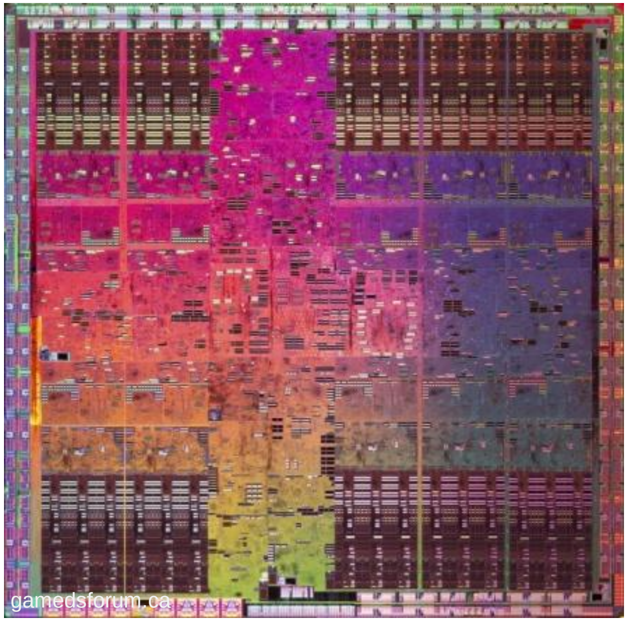
An Efficient GPU Implementation of the Irregular Barnes Hut N-Body Algorithm

Martin Burtscher
Department of Computer Science
TEXAS STATE UNIVERSITY
SAN MARCOS
The rising STAR of Texas

High-End CPU and GPU Dies





Core i7 (Nov. 2008)
4 superscalar cores

GT200 (Nov. 2008)
240 simple cores

CPU and GPU Comparison

Longhorn supercomputer at TACC

	Xeon E5540	Quadro FX 5800
Cores	4 (superscalar)	240 (simple)
Active threads	2 per core	32 per core
Frequency	2.53 GHz	1.3 GHz
Peak performance*	81 GFlop/s	933 GFlop/s
Peak bandwidth	25.6 GB/s	102 GB/s
Maximum power	80 W	189 W
Price (Dec. 2010)	$800	$2800
Main memory size	24 GB	4 GB

GPU Advantages over CPU

- Peak performance
  - 11.5x more single-precision operations per second
- Main memory bandwidth
  - 4x more bytes transferred per second
- Cost-, energy-, and size-efficiency
  - 3.3x more performance per dollar
  - 4.9x more performance per watt
  - 6.5x more performance per area

Longhorn system at TACC

(Based on peak values of Longhorn hardware)

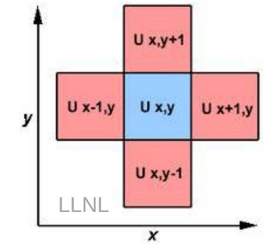
GPU Disadvantages over CPUs

- Programming and tuning are more difficult
  - More error prone and time intensive
  - Harder to get close to peak performance
  - Program needs to map well to hardware

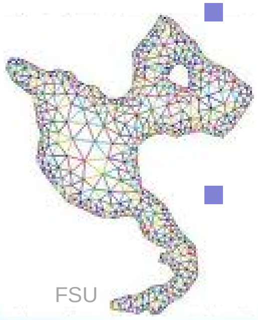
- Hardware requirements for high performance
  - Large amount of data parallelism
  - High degree of regularity (code and data accesses)
  - Little data transfer between CPU and GPU

Mapping Code to GPUs

- Only some regular codes are easy to port
  - Matrix based, regular access patterns, many ops/word
  - Dense matrix operations (level 2 and 3 BLAS)
  - Stencil codes (PDE solvers)



- Many important scientific programs are irregular
  - Build, traverse, and update dynamic data structures (trees, graphs, linked lists, priority queues, etc.)
  - E.g., *n*-body simulation, data mining, SAT solving, social networks, discrete-event simulation, meshing

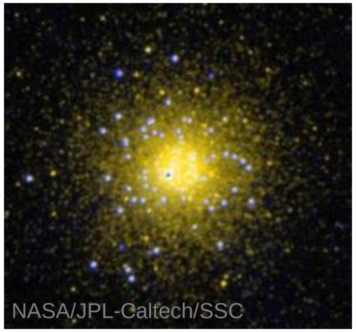


Project Goal

- Want to find *general* ways to efficiently run irregular codes on GPUs
  - Allows much broader range of applications to leverage the benefits of GPU execution

- Approach
  - Now: manually implement and optimize important irregular applications on GPUs to gain experience
  - Later: examine these and other case studies to extract common implementation and optimization strategies

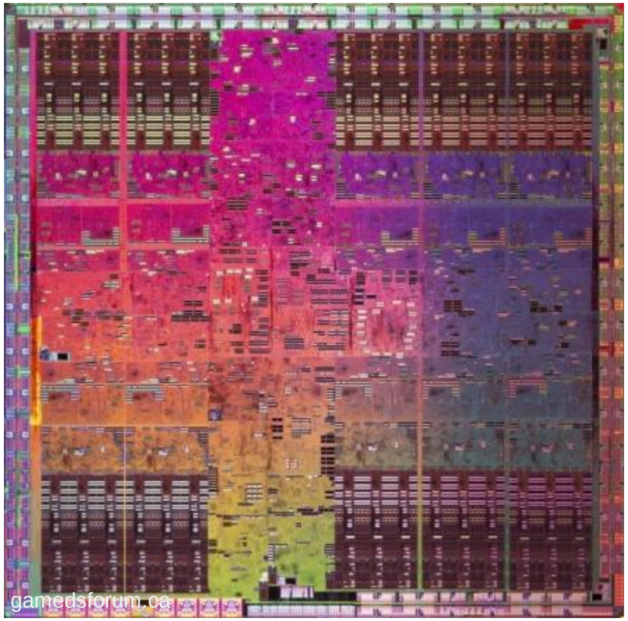
Example: N-Body Simulation



- Irregular Barnes Hut algorithm
  - Repeatedly builds unbalanced tree and performs complex traversals on it
- Our implementation
  - Designed for GPUs (not just port of CPU code)
  - First GPU implementation of entire BH algorithm
- Results
  - 1 GPU is faster than 16 CPUs (128 cores) on this code
  - GPU has better architecture for this irregular algorithm

Outline

- Introduction
- GT200 architecture
- Barnes Hut algorithm
- CUDA implementation
- Experimental results
- Conclusions



Calling GPU Kernels

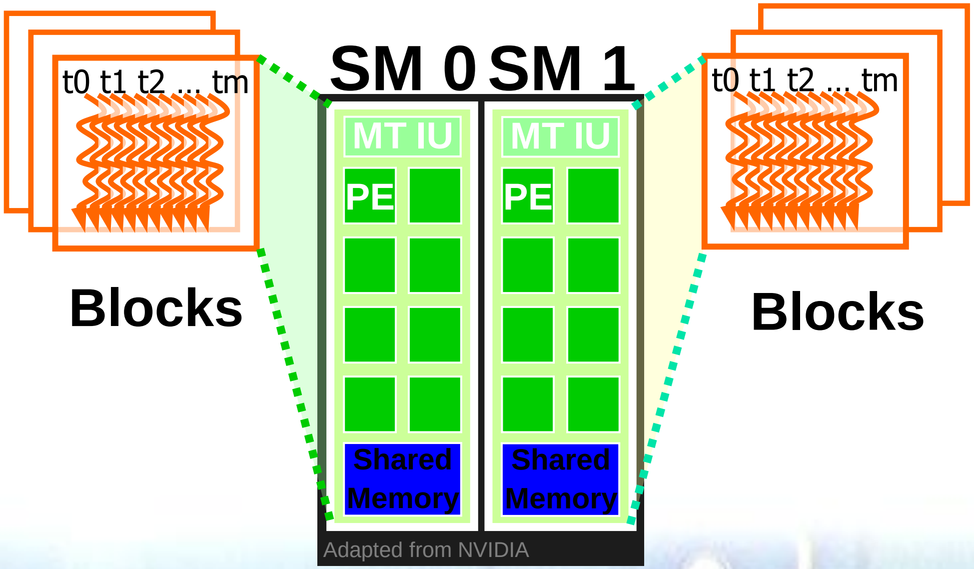
- Kernels are functions that run on the GPU
  - Callable by CPU code

```
KernelName<<<blocks, threads>>>(arg1, arg2, ...);
```

- Launch configuration (programmer selectable)
  - Special parameters: number of blocks and threads
    - Kernel call automatically spawns m blocks with n threads (i.e., m*n threads total) that run a copy of the kernel code

Block and Thread Allocation

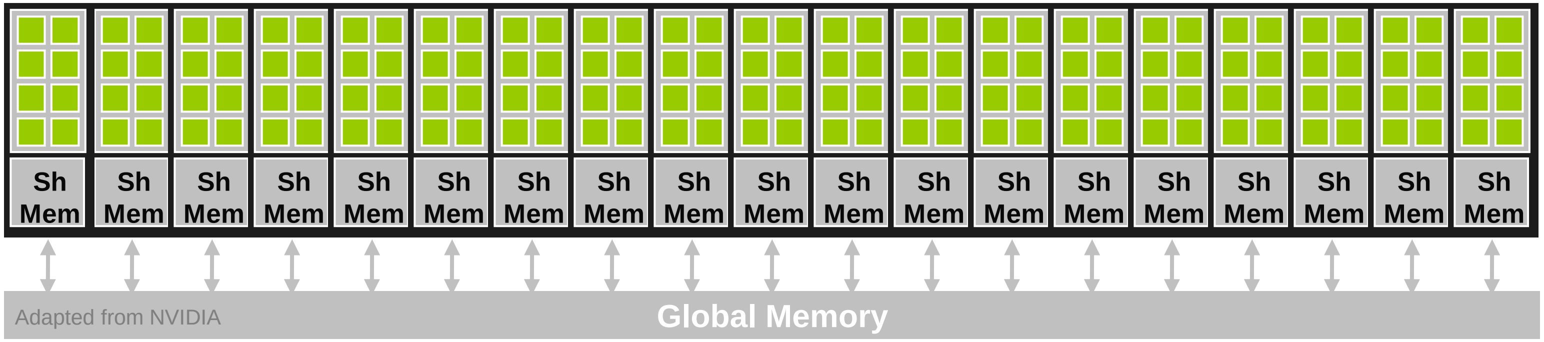
- Blocks assigned to SMs
  - Streaming multiprocessors
- Threads assigned to PEs
  - Processing elements



- Hardware limits
  - 8 resident blocks per SM
  - 1024 resident threads per SM
  - 512 threads per block
  - Above limits are lower if register or shared mem usage is too high
  - 65535 blocks per kernel

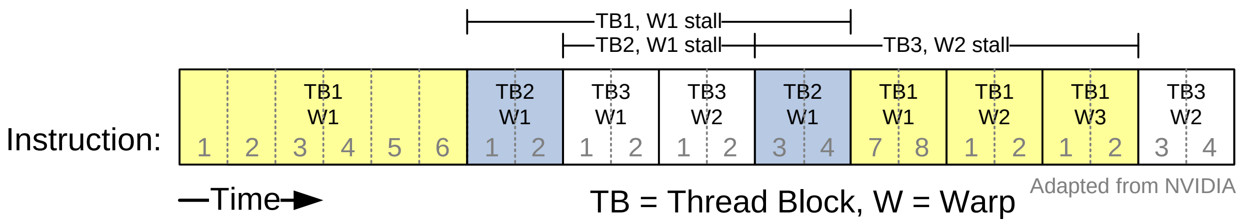
GPU Hardware

- 30 SMs with 8 PEs each
- SMs have fast barriers, thread voting, shared memory, and special instruction units
  - Very fast thread communication within block
  - Slow communication between blocks (DRAM atomics)



Warp-Based Execution

- 32 contiguous threads form a *warp*
  - Execute same instruction in same cycle (or disabled)
  - At any time, only one warp is executed per SM
  - Warps are scheduled out-of-order w.r.t. each other

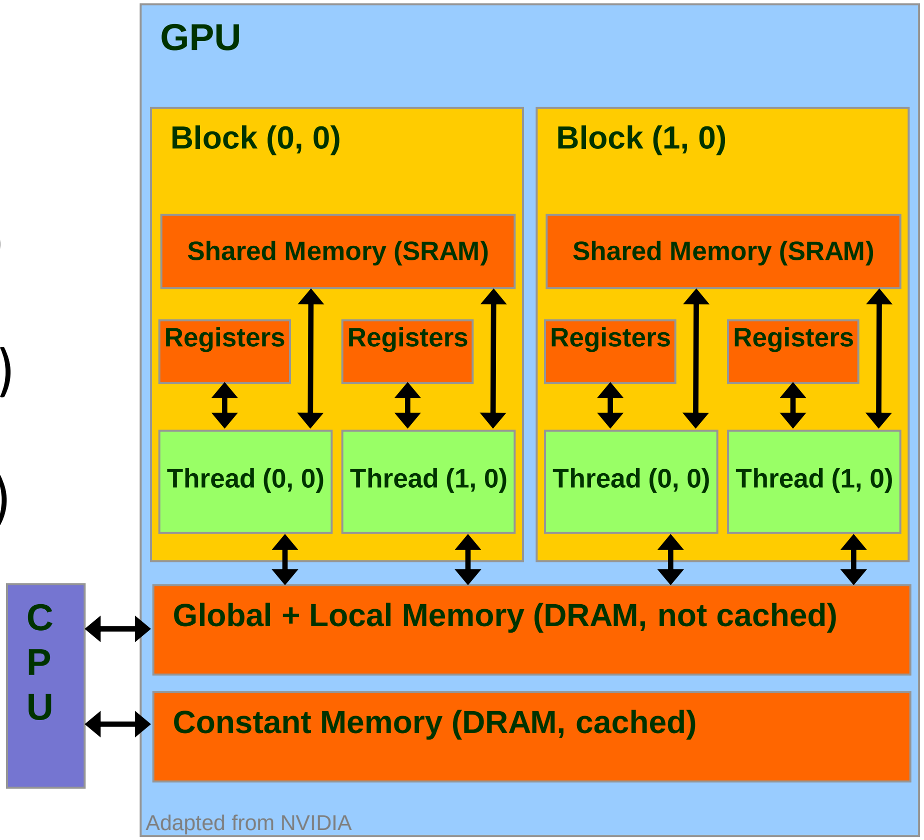


Adapted from NVIDIA

- Thread divergence (reduction of parallelism)
  - Some threads in warp jump to different PC than others
  - Hardware runs subsets of warp until they re-converge

GPU Memories

- Memory types
  - Registers (r/w per thread)
  - Local mem (r/w per thread)
  - Shared mem (r/w per block)
    - Software controlled cache
  - Global mem (r/w per kernel)
    - No hardware cache
  - Constant mem (r per kernel)
- Separate from CPU
  - CPU can access global and constant mem via PCIe bus
  - Requires explicit transfer

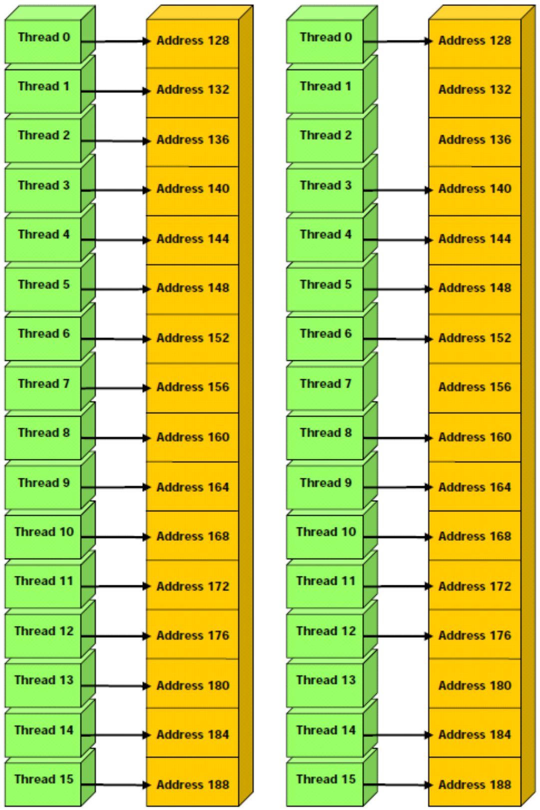


Fast Memory Accesses

- Coalesced main memory access (16x bandwidth)
  - Under some conditions, HW combines multiple half-warp memory accesses into a single coalesced access
  - 64-byte aligned 64-byte line (any word permutation)
- Bank-conflict-free shared memory access (16x)
  - No superword alignment requirement
  - 16 different banks per half warp or same word

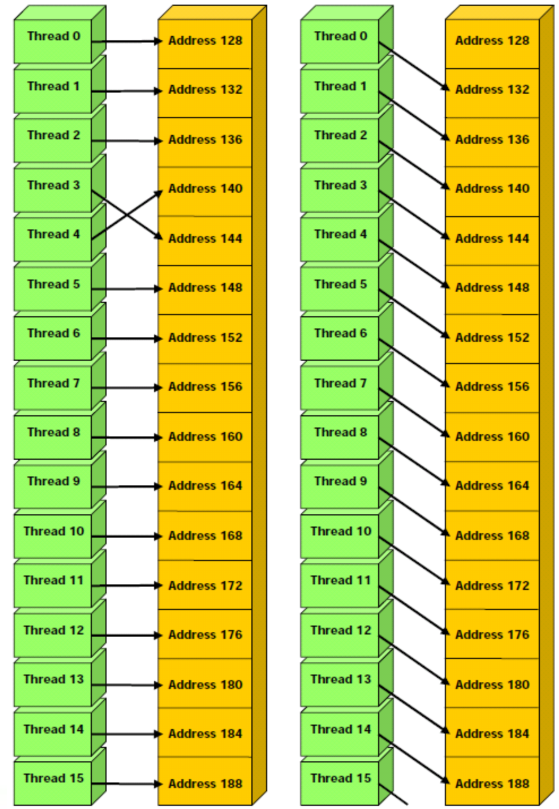
Coalesced Main Memory Accesses

single coalesced access



NVIDIA

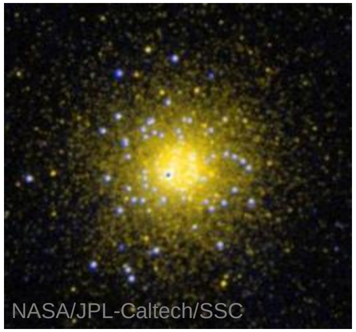
one and two coalesced accesses



NVIDIA

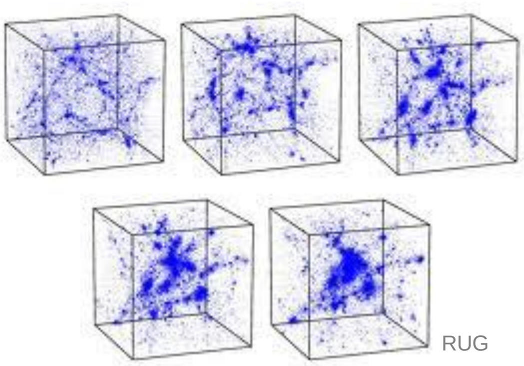
Outline

- Introduction
- GT200 architecture
- Barnes Hut algorithm
- CUDA implementation
- Experimental results
- Conclusions

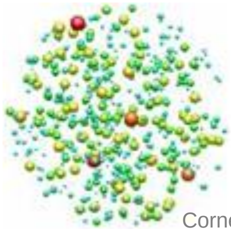


N-Body Simulation

- 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

Barnes Hut Idea

- Precise force calculation
  - Requires $O(n^2)$ operations ($O(n^2)$ body pairs)

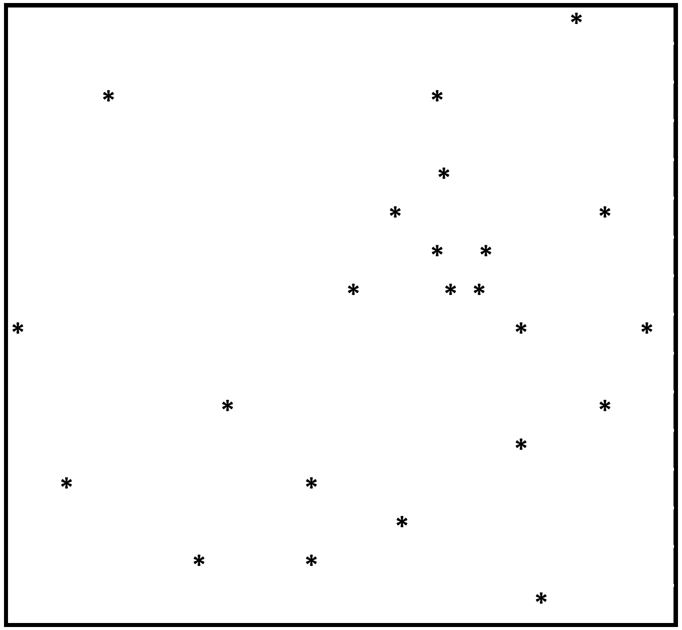
- Barnes and Hut (1986)
  - Algorithm to approximately compute forces
    - Bodies' initial position & velocity are also approximate
  - Requires only $O(n \log n)$ operations
  - Idea is to "combine" far away bodies
  - Error should be small because *force* $\sim 1/dist^2$

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

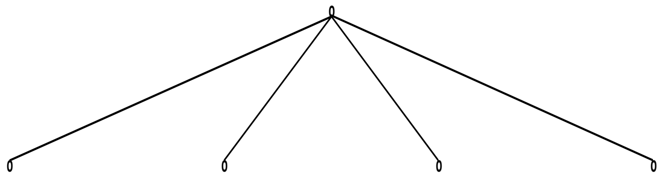
Build Tree (Level 1)

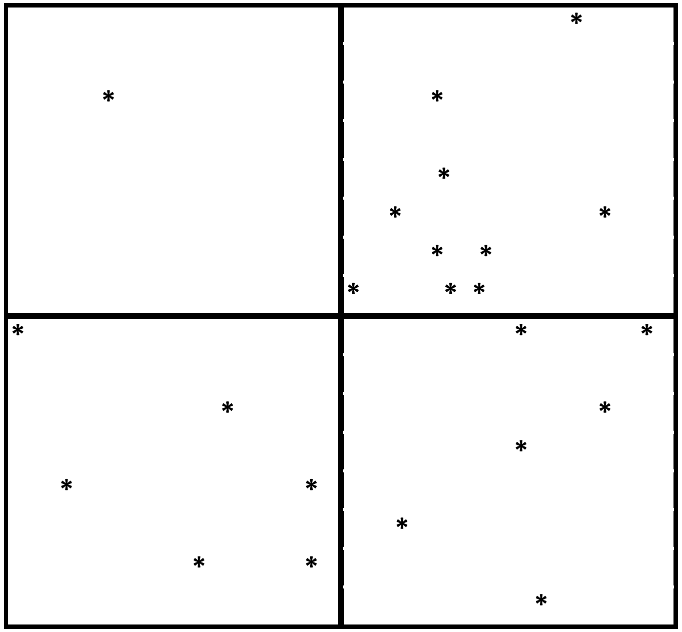
0



Compute bounding box around all bodies → tree root

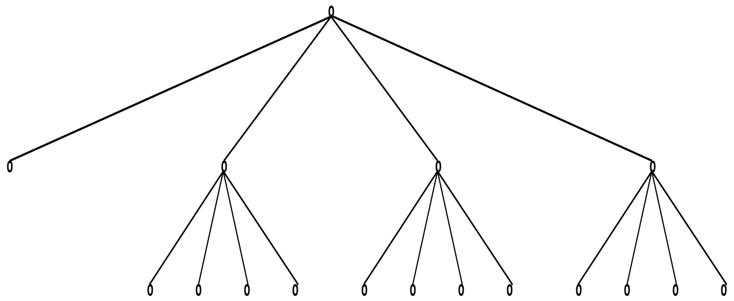
Build Tree (Level 2)

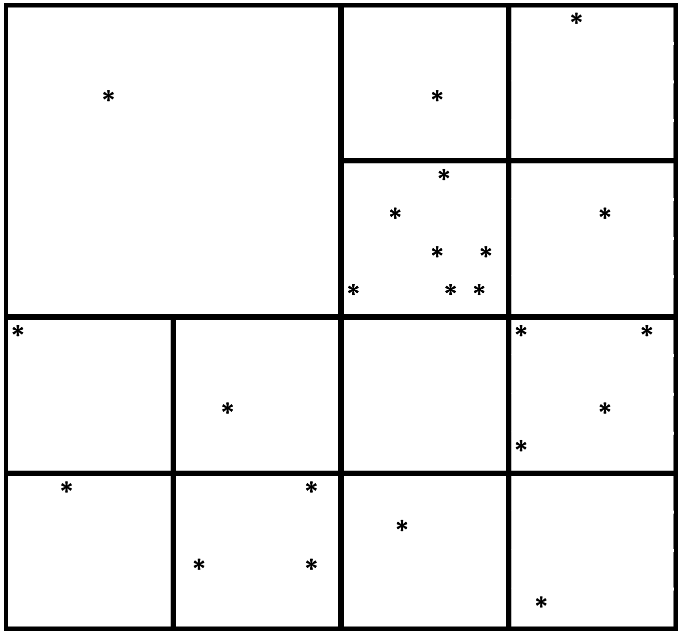




Subdivide space until at most one body per cell

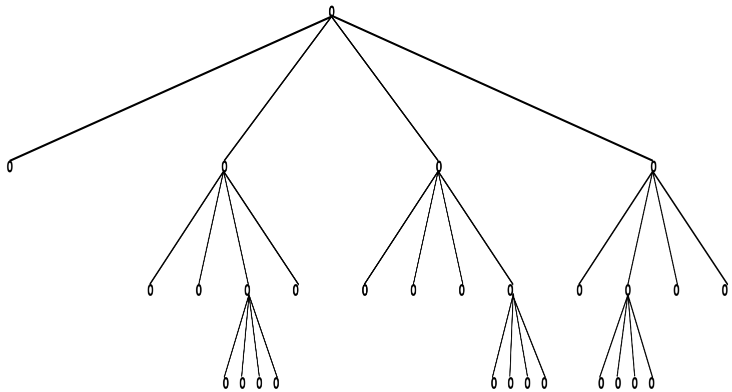
Build Tree (Level 3)





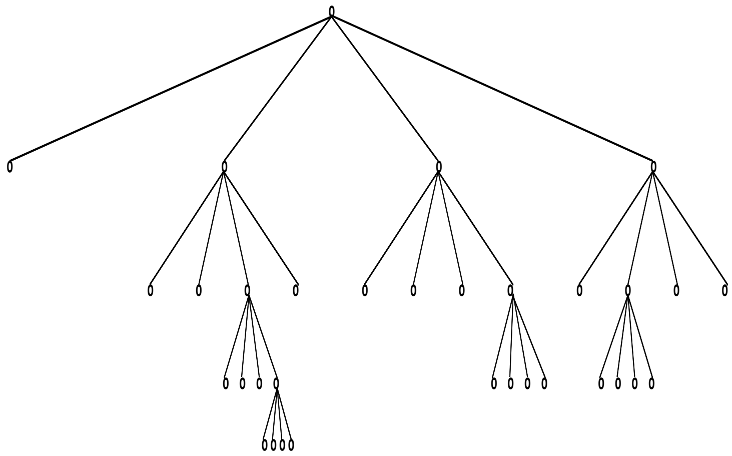
Subdivide space until at most one body per cell

Build Tree (Level 4)



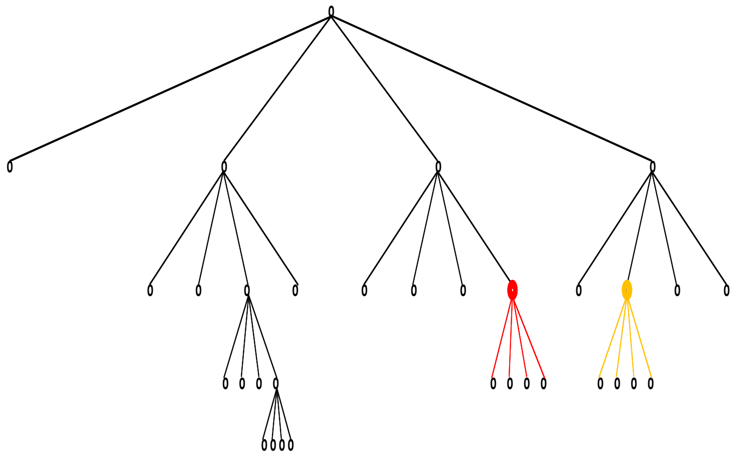
Subdivide space until at most one body per cell

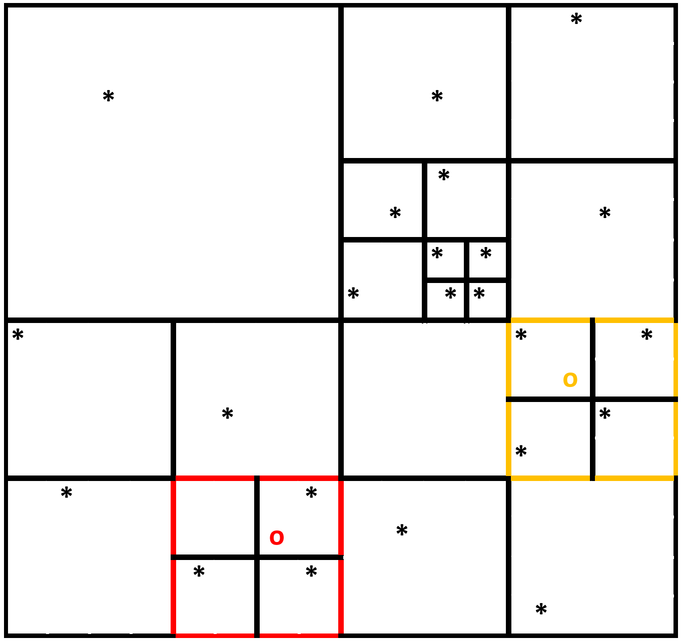
Build Tree (Level 5)



Subdivide space until at most one body per cell

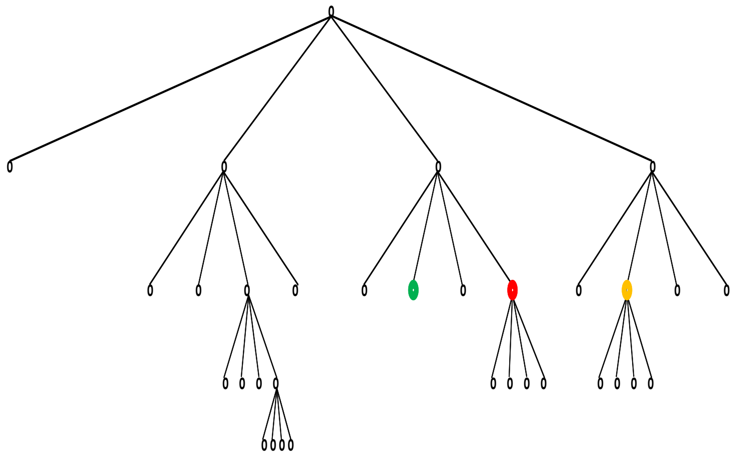
Compute Cells' Center of Mass

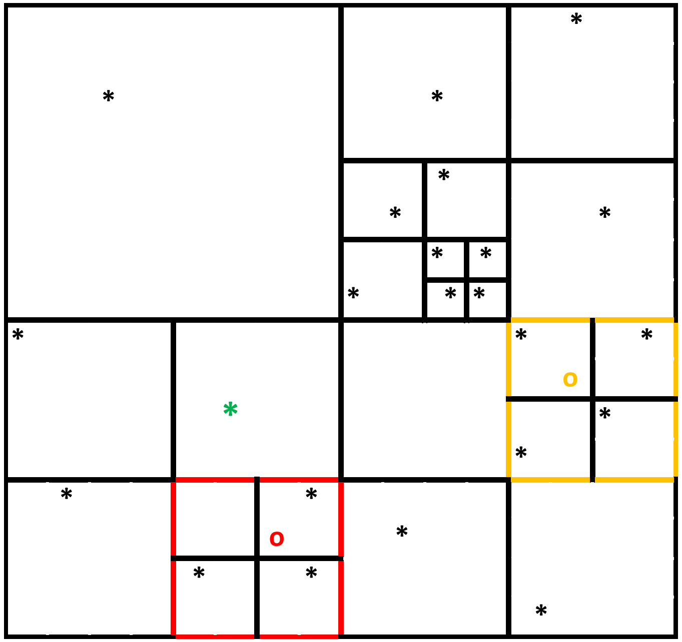




For each internal cell, compute sum of mass and weighted average of position of all bodies in subtree; example shows two cells only

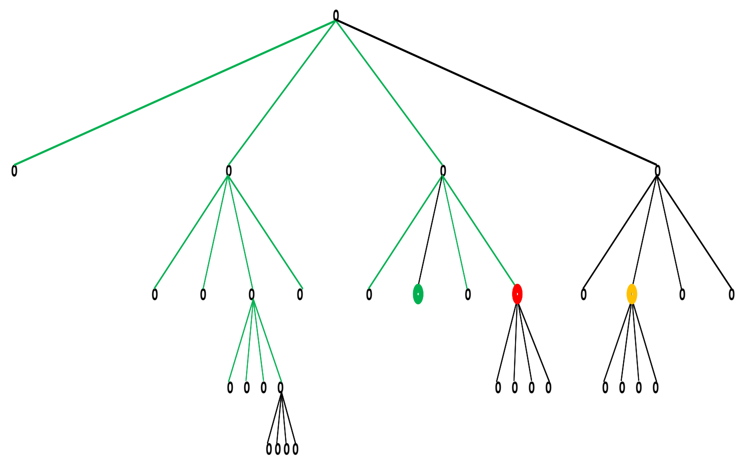
Compute Forces

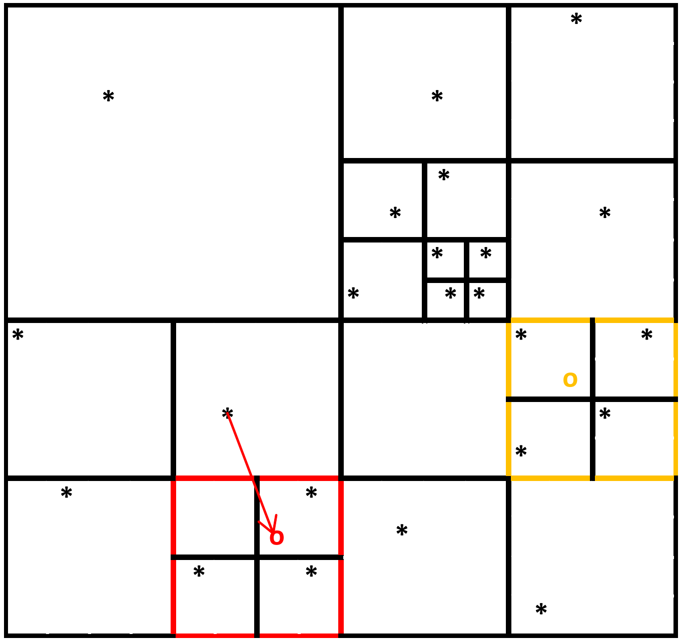




Compute force, for example, acting upon green body

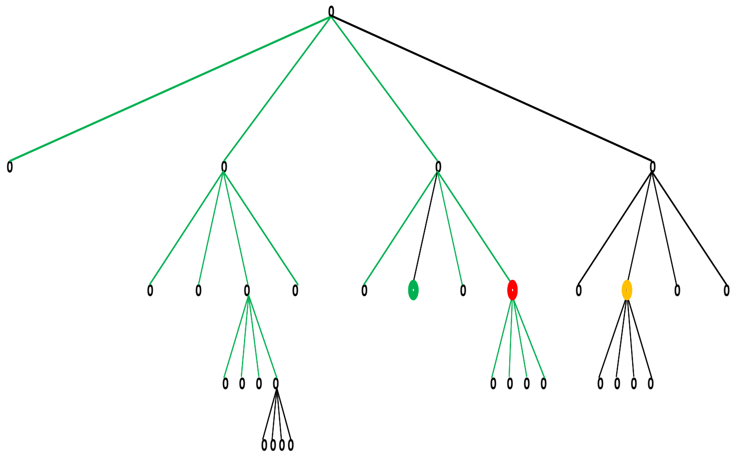
Compute Force (short distance)

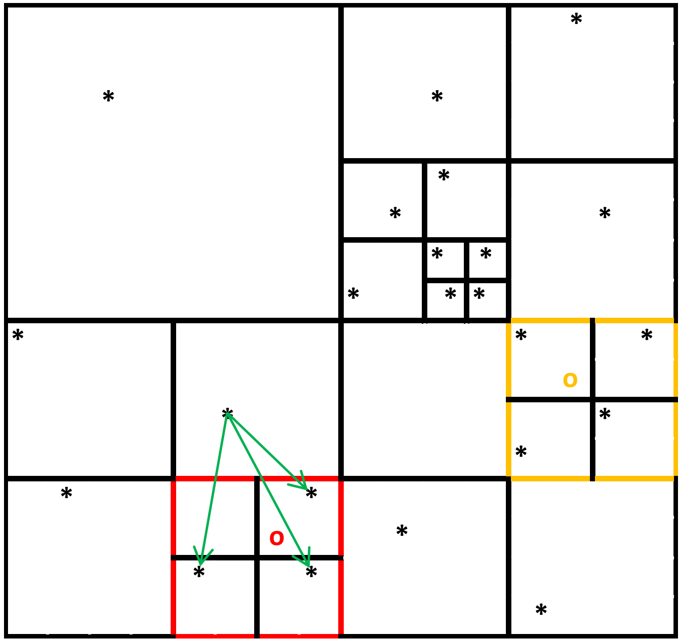




Scan tree depth first from left to right; green portion already completed

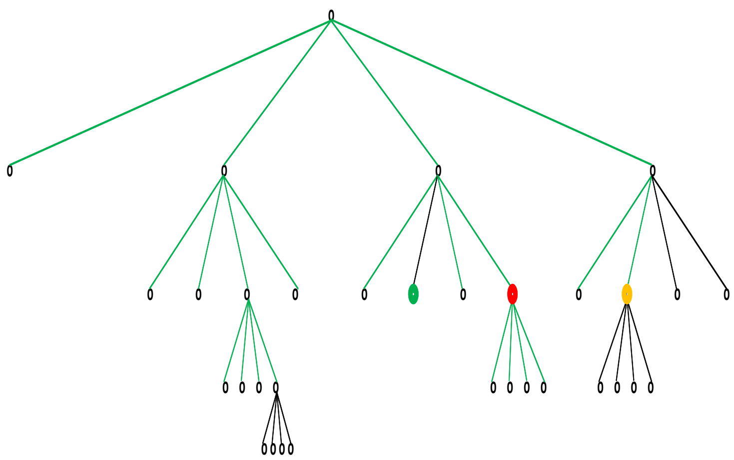
Compute Force (down one level)

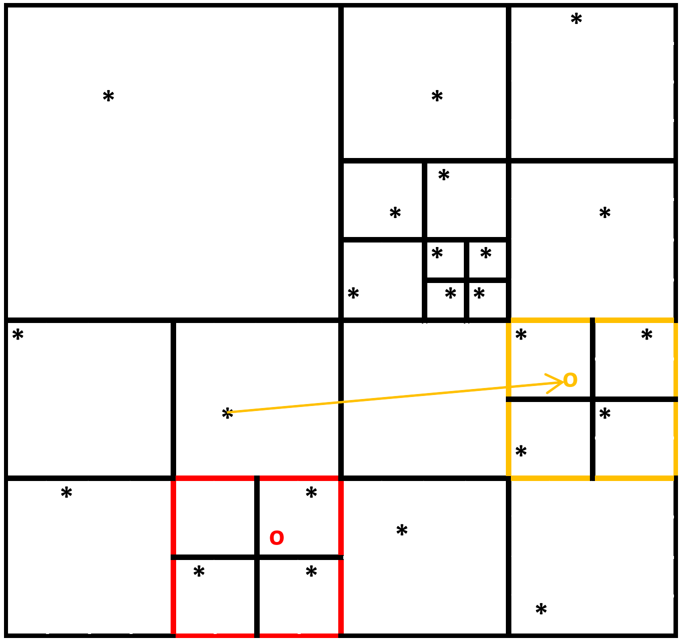




Red center of mass is too close, need to go down one level

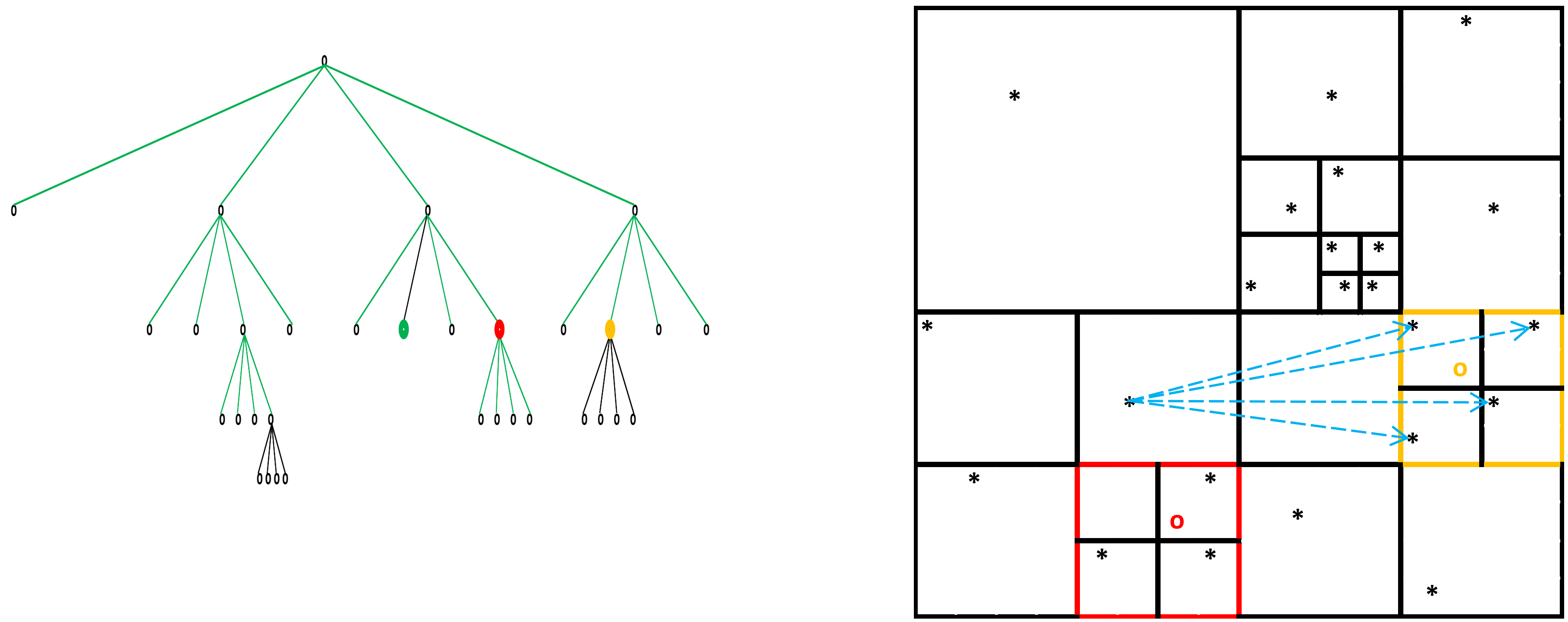
Compute Force (long distance)





Yellow center of mass is far enough away

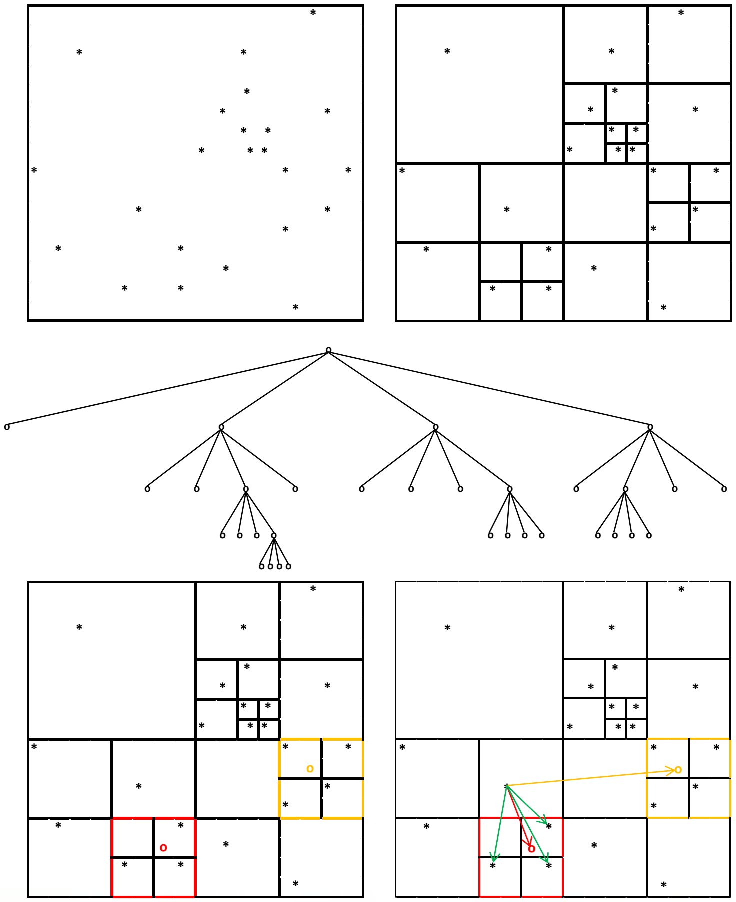
Compute Force (skip subtree)



Therefore, entire subtree rooted in the yellow 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

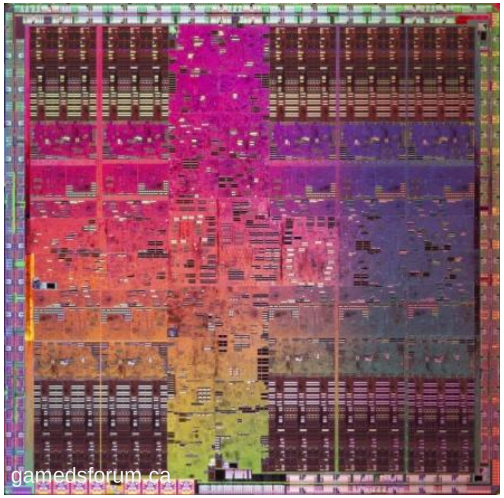
```
bodySet = ...
foreach timestep do {                          // O(n log n) + ordered sequential
  bounding_box = new Bounding_Box();
  foreach Body b in bodySet {                  // O(n) parallel reduction
    bounding_box.include(b);
  }
  octree = new Octree(bounding_box);
  foreach Body b in bodySet {                  // O(n log n) top-down tree building
    octree.Insert(b);
  }
  cellList = octree.CellsByLevel();
  foreach Cell c in cellList {                 // O(n) + 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
- GT200 architecture
- Barnes Hut algorithm
- CUDA implementation
- Experimental results
- Conclusions

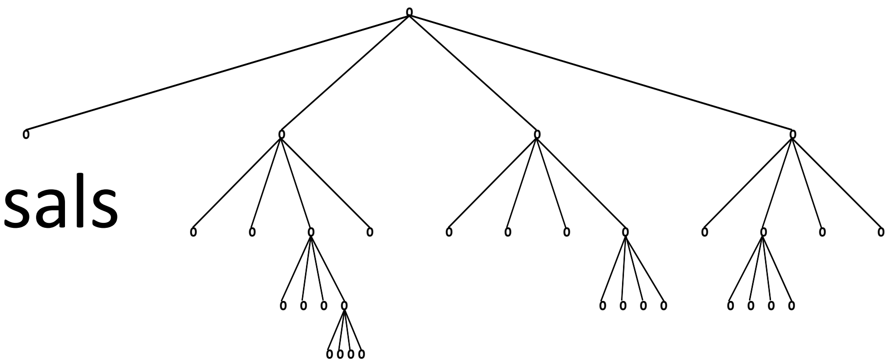
Efficient GPU Code

- 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

- Based on 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 {

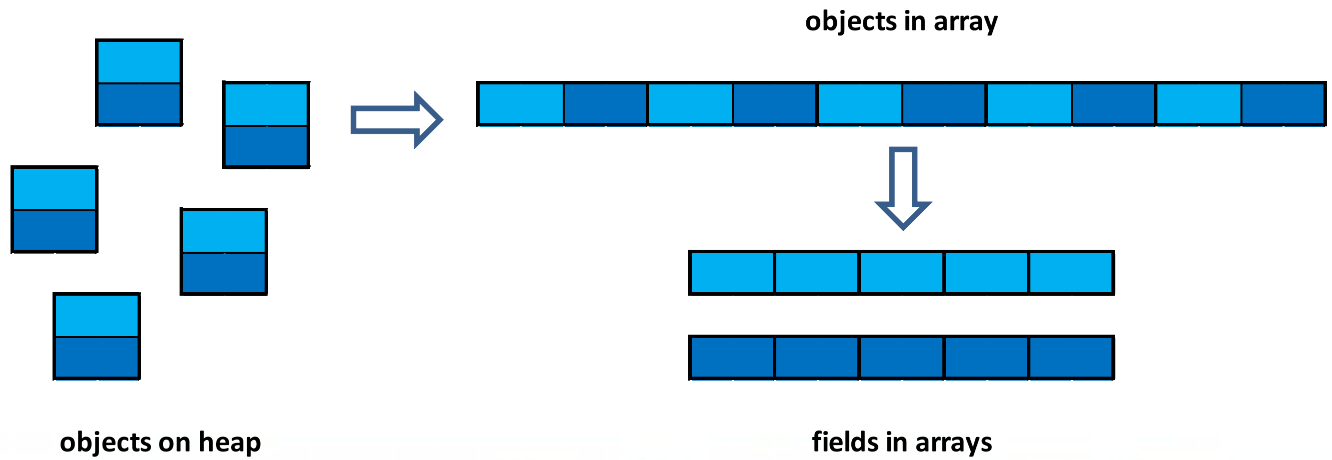
1. Compute bounding box around bodies
2. Build hierarchical decomposition, i.e., octree
3. Summarize body information in internal octree nodes
4. Approximately sort bodies by spatial location (optional)
5. Compute forces acting on each body with help of octree
6. Update body positions and velocities

}

Transfer result from GPU and output

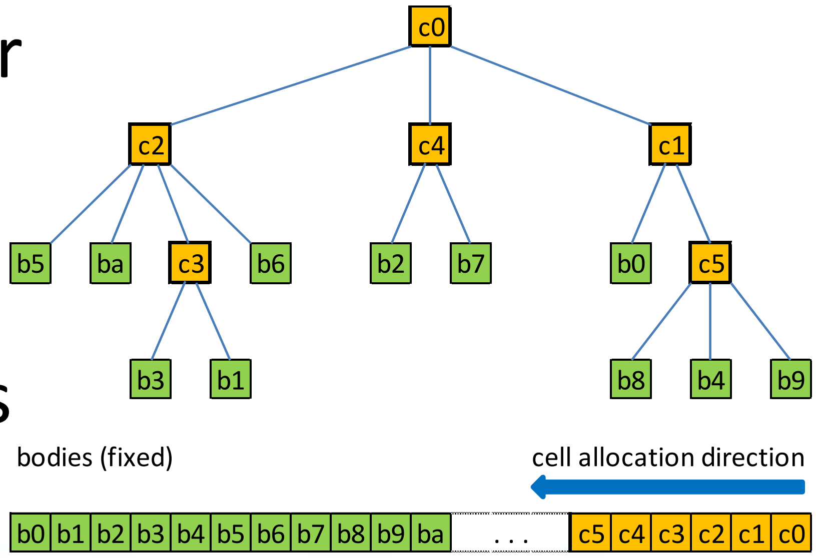
Global Optimizations

- 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

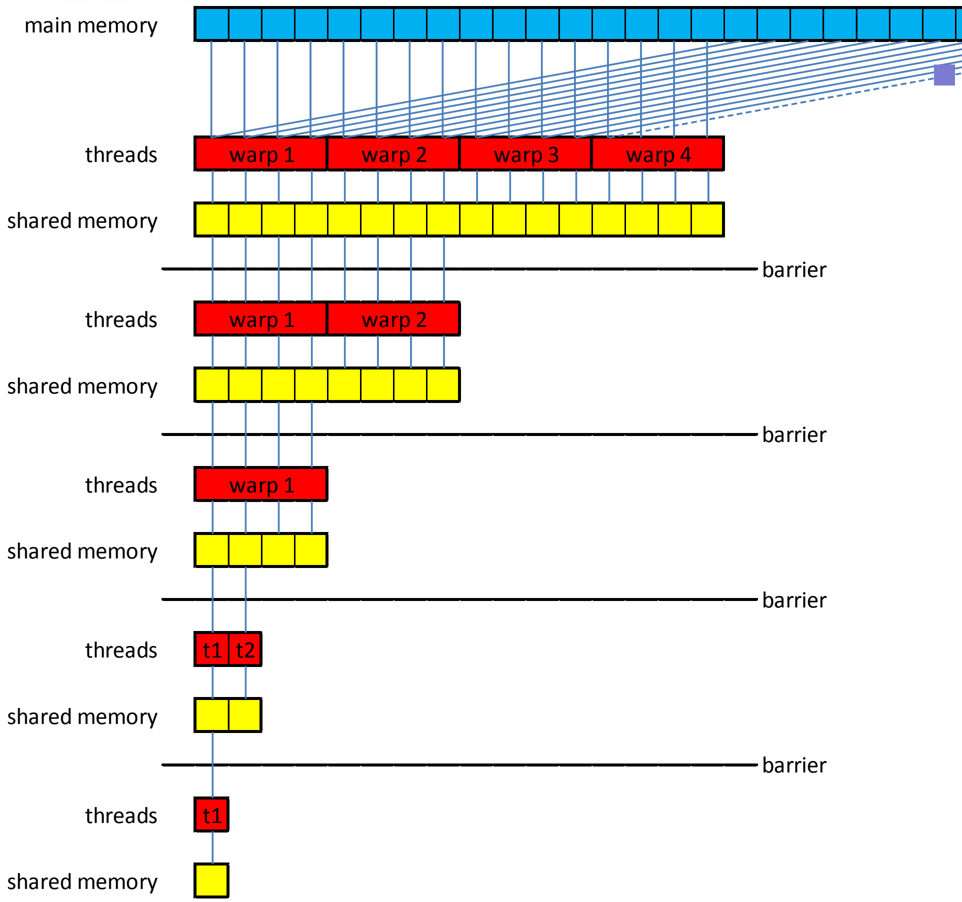


Global Optimizations (cont.)

- Maximized thread count (rounded to warp size)
- Maximized resident block count (all SMs used)
- 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

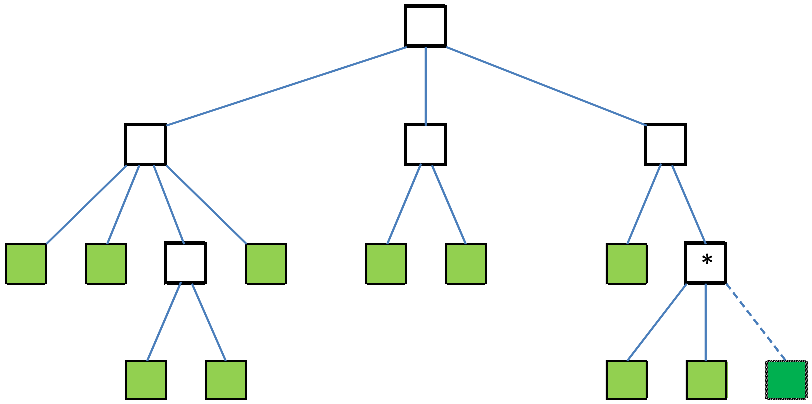


- Optimizations
  - Equal sized chunks
  - Fully coalesced
  - Fully cached
  - No bank conflicts
  - Minimal divergence
  - Built-in min and max
  - 2 red/mem, 6 red/bar
  - 1 atomic inc per block
  - 512 threads per SM

Kernel 2: Build Octree

- Optimizations
  - Load-balance bodies
  - Cache root in registers
  - Only lock leaf “pointers”
  - Light-weight lock release
  - No re-traverse after lock acquire failure
  - Throttle lock polling
  - 288*2 threads per SM

Top-down tree building



Kernel 2: Build Octree (cont.)

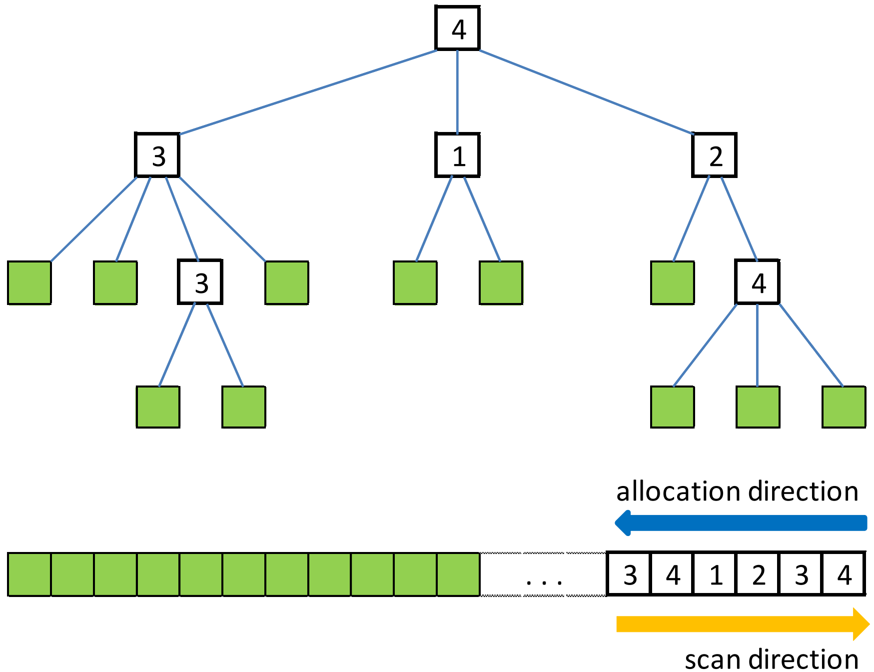
```
// initialize
cell = find_insertion_point(body);  // nothing locked, cell cached
child = get_insertion_index(cell, body);
if (child != locked) {  // skip atomic if already locked
  if (child == atomicCAS(&cell[child], child, lock)) {
    if (child == null) {  // fast path (frequent)
      cell[child] = body;  // insert body (releases lock)
    } else {  // slow path (infrequent)
      new_cell = ...;  // atomically get next unused cell
      // insert the existing and new body into new_cell
      __threadfence();  // make new_cell subtree visible
      cell[child] = new_cell;  // insert subtree (releases lock)
    }
    success = true;  // flag showing insertion succeeded
  }
}
__syncthreads();  // wait for other warps
```

Architectural Advantage

- Thread throttling
  - Avoids likely useless work, in particular expensive memory polling operations to acquire a lock
  - Speeds up threads that successfully acquired a lock because more bandwidth is available to them
- Hardware support
  - Thread divergence enforces throttling within warp
  - Fast HW barriers make warp throttling possible in SW (CPU barriers are implemented in SW via memory)

Kernel 3: Summarize Subtrees

Bottom-up tree traversal



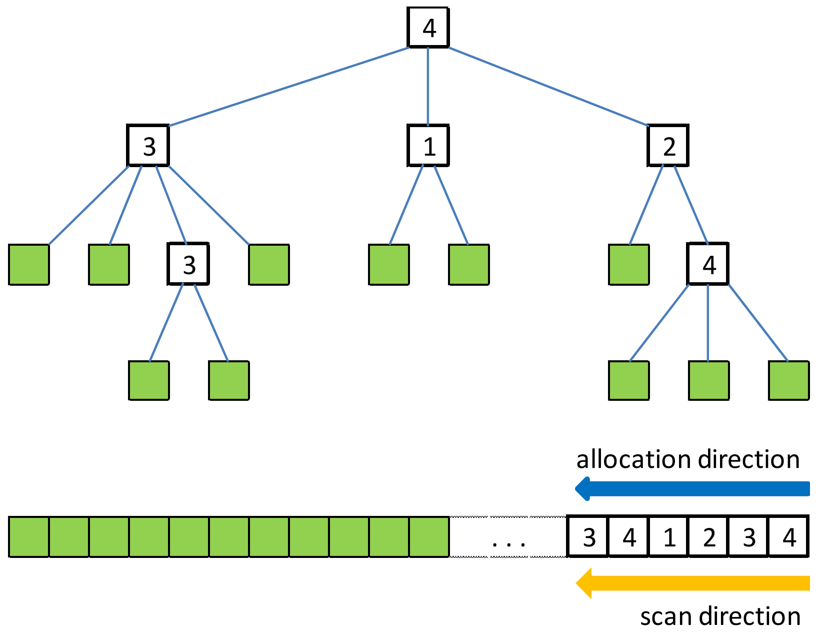
- Optimizations
  - Load-balance cells
  - No parent "pointers"
  - Scan avoids deadlock
  - Partially coalesced
  - Use mass as flag + fence
    - No locks, no atomics
  - Cache unready "children"
  - Automatic throttling
  - Piggyback on traversal
    - Count bodies in subtrees
    - Move nulls to back
  - 256 threads per SM

Kernel 3: Summarize Subtrees (cont.)

```
// initialize
if (missing == 0) {  // new cell, get child info
  // initialize center of gravity
  for (/*iterate over existing children*/) {
    if (/*child is ready*/) {
      // add its contribution to center of gravity
    } else {
      // cache child index
      missing++;
} } }
if (missing != 0) {  // try to get missing child info
  do {
    if (/*last cached child is now ready*/) {
      // remove from cache and add its contribution to center of gravity
      missing--;
    }
  } while (/*missing changed*/ && (missing != 0));  // exit to avoid deadlock
}
if (missing == 0) {  // got all info, update cell info
  // store center of gravity
  __threadfence();  // make sure center of gravity is visible
  // store cumulative mass (indicates cell is ready)
  success = true;  // local flag indicating that computation for cell is done
}
```

Kernel 4: Sort Bodies (optional)

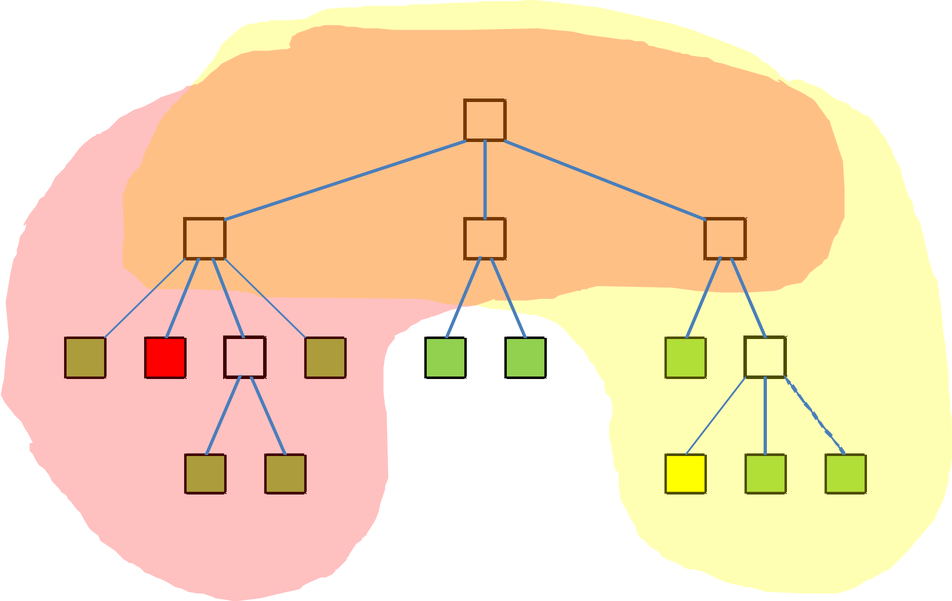
Top-down tree traversal



- Optimizations
  - (Similar to Kernel 3)
  - Load-balance cells
  - Scan avoids deadlock
  - Use data field as flag
    - No locks, no atomics
  - Use counts from Kernel 3
  - Automatic throttling
  - 512 threads per SM

Kernel 4: Force Calculation

Top-down prefix traversal



- Optimizations
  - Load balanced
  - Use built-in rsqrt

- Optimizations (cont.)
  - Group similar work together
    - Uses sorting to minimize union of prefixes in warp
    - Early out (nulls in back)
  - Traverse whole union to avoid divergence (thread voting)
  - Lane 0 reads data for entire warp, no sync needed
  - Lane 0 controls iteration stack for entire warp (fits in cache)
  - Cache tree-level-based data
  - 384*2 threads per SM

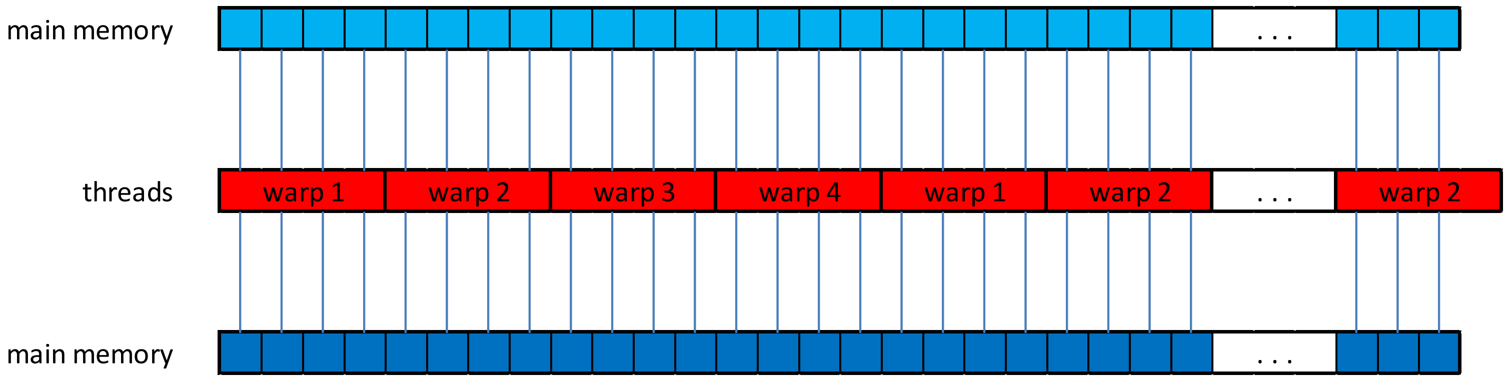
Architectural Advantages

- 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
  - CPUs can only do this partially in highest shared cache level (no sync guarantee, still incurs p*L3 latency)
- Warp-based execution
  - Enables data sharing in warps w/o synchronization
- RSQRT instruction
  - Quickly computes approximation of 1/sqrt(x)

Kernel 5: Advance Bodies

- Optimizations
  - Fully coalesced, no divergence
  - Load balanced, 512 threads per SM

Straightforward streaming

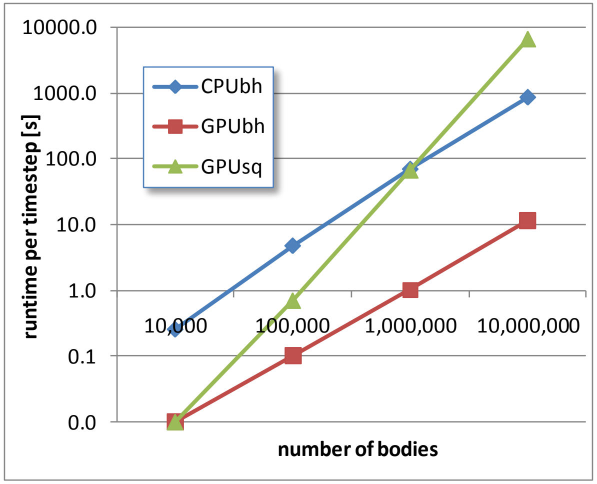


Related Work

- GPU-based *n*-body simulation
  - GPU only: O(n^2) algorithm
    - Close to peak performance with blocking
  - CPU + GPU: tree construction and traversal on CPU, force calculation (based on interaction lists) on GPU
    - Problem size not restricted to GPU memory size
- Irregular GPU codes
  - Mostly sparse matrix computations
  - Parallel traversals of graphs built on CPU

Outline

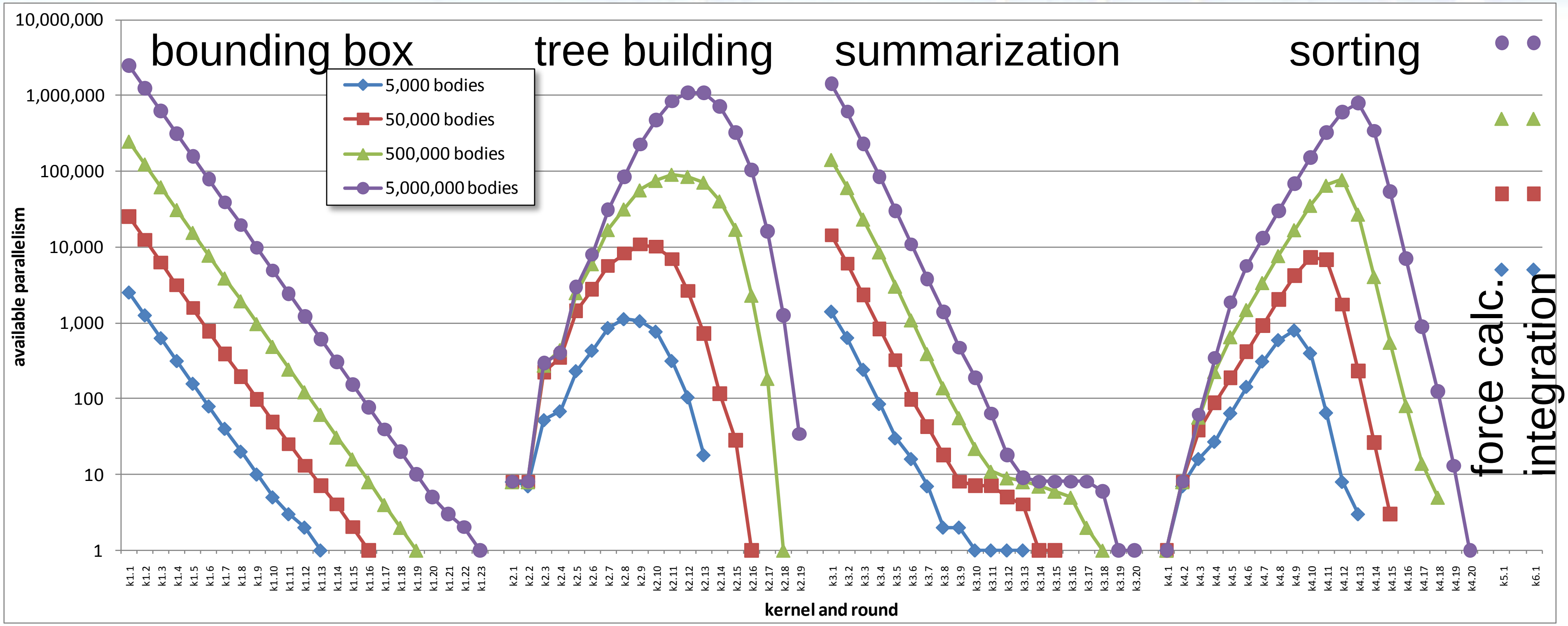
- Introduction
- GT200 architecture
- Barnes Hut algorithm
- CUDA implementation
- Experimental results
- Conclusions



Evaluation Methodology

- Implementations
  - Parallel CUDA C versions of Barnes Hut & O(n^2) algorithm
  - Parallel pthreads C version of BH algorithm (SPLASH-2)
- Systems and compilers
  - Longhorn (TACC): Quadro FX 5800 GPU, 1.3 GHz, 30 SMs
  - Nautilus (NICS): Xeon X7550 CPU, 2 GHz, 8 cores per CPU
  - nvcc v3.0 (-O3 -arch=sm_13); icc v11.1 (-O3 -xW -pthread)
- Inputs and metric
  - 5k, 50k, 500k, and 5M star clusters (Plummer model)
  - Median runtime of three experiments, excluding I/O

Available Amorphous Data Parallelism



- Lots of bodies (K 1, 2, 5, 6) and cells (K 3, 4) can be processed in parallel (with only data dependencies)

Nodes Touched per Activity (5M Input)

- K1: pair reduction
- K2: tree insertion
- K3: bottom-up step
- K4: top-down step
- K5: prefix traversal
- K6: integration step

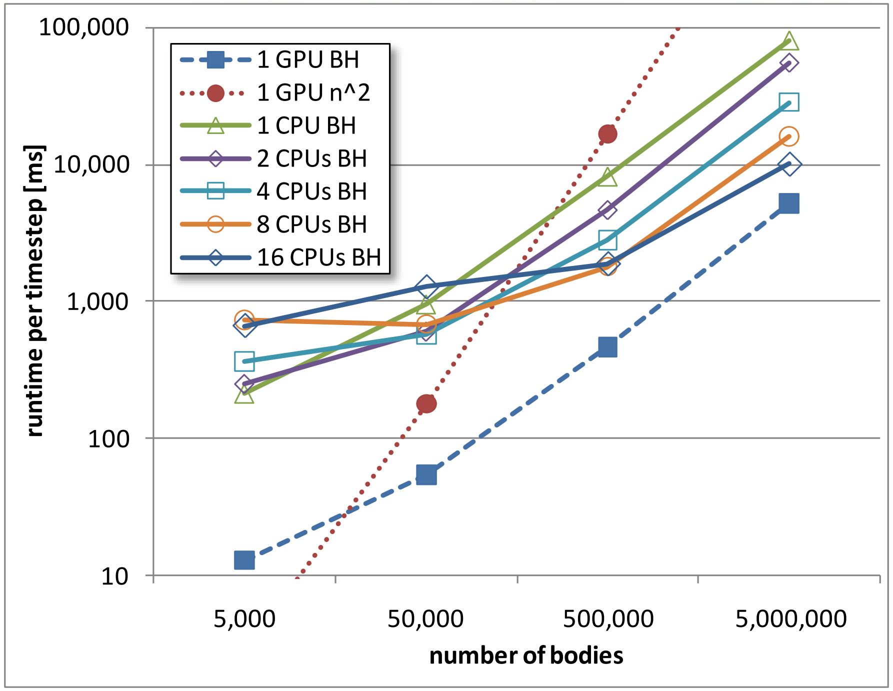
	neighborhood size		
	min	avg	max
kernel 1	1	2.0	2
kernel 2	2	13.2	22
kernel 3	2	4.1	9
kernel 4	2	4.1	9
kernel 5	818	4,117.0	6,315
kernel 6	1	1.0	1

- Max tree depth ≤ 22
- Cells have 3.1 children

- Prefix ≤ 6,315 nodes (≤ 0.1% of 7.4 million)
- BH algorithm & sorting to min. union work well

Runtime Comparison

- GPU vs. CPU (all inputs)
  - GPU over 15x faster than CPU on irregular BH code
  - GPU faster than 16 CPUs with 128 x86 cores
- BH vs. $O(n^2)$ algorithm
  - $O(n^2)$ better for ≤ 10k
- GPU BH inefficiency
  - 5k input too small for 7,680 to 23,040 threads



- Architectural advantage
  - Low thread startup cost

Kernel Performance for 5M Input

runtime [ms]	kernel 1	kernel 2	kernel 3	kernel 4	kernel 5	kernel 6
CPU serial	50.0	2,160.0	430.0	310.0	382,840.0	990.0
GPU parallel	0.8	868.0	100.3	38.6	4,202.8	4.1
GPU percent	0.0%	16.6%	1.9%	0.7%	80.6%	0.1%
CPU/GPU	62.5	2.5	4.3	8.0	91.1	241.5

- Heterogeneous solution not useful
  - PCIe transfer @ 3.13 GB/s requires over 130ms
  - K2 is weak but also scales poorly on CPU (DS mismatch)
  - K3 is a little slow but too short to move to CPU

	kernel 1	kernel 2	kernel 3	kernel 4	kernel 5	kernel 6	total	O(n^2) alg
Gflop/s	37.62	0.30	0.70	0.00	93.94	18.29	75.79	304.90
Gbytes/s	75.00	1.38	2.95	4.69	3.13	73.17	2.91	0.95
runtime [s]	0.0	0.9	0.1	0.0	4.2	0.0	5.2	1,639.9

- 76 Gflop/s on irregular code (memory bound)

Kernel Scaling on 5M Input

			kernel 1	kernel 2	kernel 3	kernel 4	kernel 5	kernel 6
warp	scaling	warps	16	9	8	16	12	16
		speedup	9.8	4.8	7.2	1.0	18.6	14.0
		efficiency	61.0%	53.4%	90.3%	6.3%	154.8%	87.5%
block	scaling	blocks	30	60	30	30	60	30
		speedup	14.8	1.2	2.9	1.7	15.4	6.0
		efficiency	49.2%	2.0%	9.5%	5.6%	25.7%	19.9%

- Warps & blocks capped by register & cache use
- Warp scaling is good
  - K4 almost saturates memory bandwidth with 1 warp
  - K5 exhibits superlinear speedup due to OOO execution
- Block scaling is poor (memory bandwidth limited)
  - Lot of computations help (K5), coalescing helps (K1,K6)

Optimization Benefit by Kernel

	throttling of kernel 2	warp-based mem access in kernel 5	thread voting in kernel 5	sorting of bodies for kernel 5	sync'ed execution in kernel 5
5,000	1.062	0.914	3.276	1.845	3.91
50,000	1.073	0.829	1.900	4.214	52.83
500,000	1.016	1.088	1.817	6.254	568.68
5,000,000	1.004	1.123	1.688	9.056	5088.67

- Warp throttling: helps while tree is small
- 1 access per warp: can help (5.7x on older GPUs)
- Voting: much faster than cache-based reduction
- Sorting: helps a lot, helps more for larger inputs
- Divergence avoidance: absolutely paramount
  - CPU-style coding causes divergence and de-coalescing

Outline

- Introduction
- GT200 architecture
- Barnes Hut algorithm
- CUDA implementation
- Experimental results
- Conclusions

Optimization Summary

- Exploit hardware features
  - Fast synchronization & thread startup, special instructions, coalescing, even lockstep execution and thread divergence
- Minimize thread divergence
  - Group similar work together & force synchronicity
- Minimize main memory accesses
  - Share data within warp and throttle polling accesses
- Implement entire algorithm on GPU
  - Avoids data transfers & data structure inefficiencies

Optimization Summary (cont.)

- Use light-weight locking and synchronization
  - Minimize locks, reuse fields, and use fence + store ops
- Combine traversals
  - Perform multiple operations during single traversal
- Maximize parallelism and load balance
  - Parallelize every step within and across SMs
- Maximize coalescing
  - Partial coalescing due to array-based implementation

Conclusions

- Irregularity does not necessarily prevent high-performance on GPUs
  - Entire Barnes Hut algorithm implemented on GPU
    - Builds and traverses unbalanced octree
  - One GPU outperforms 16 high-end 8-core Xeons
- Code directly for GPU, do not merely adjust CPU code
  - Requires different data and code structures
  - Benefits from different algorithmic modifications

Future Work

- Implement other important irregular codes on GPUs
  - Discover new implementation and optimization techniques
- Extract and generalize common strategies
  - Enable entire classes of irregular programs to leverage the performance and energy/cost-efficiency of GPU execution

- Acknowledgments
  - Keshav Pingali: overall project support
  - TACC, NCIS, NVIDIA: hardware resources
  - NSF, IBM, NEC, Intel, UT Austin, Texas State: funding