15-418 Project Final Report

Aimee Feng and Jackson Romero

Summary

We have implemented a 2D approximation simulation of the collision of particles in hard-body physics, where each particle in the simulation has mass and a gravitational pull, with the Barnes Hut algorithm. Using the NVIDIA GPUs in the lab, we have optimized a CUDA implementation, drawing inspiration from a paper published by Burtscher and Pingali[1] and a paper published by NVIDIA[2]. We measured the speedup of the optimized CUDA implementation against a sequential CPU implementation, and analyzed the tradeoffs between speedup and approximation accuracy.

Background

Barnes-Hut Algorithm

The Barnes-Hut algorithm provides an approximation for N-body simulations. Particles that are classified ‘far enough’ from a particular particle will be grouped together with nearby particles that are also ‘far enough’ from the particle for which we are computing the force acting on it. We abstract the particle bodies away to be points located at the corresponding center of mass. Given a set of points and their mass, location, and velocity, one iteration of the Barnes-Hut Algorithm will produce the updated location and velocity for each point after one time step based on the gravitational forces acting upon that point.

For N particles, calculating the force that every other particle exerts on a particle is $O(N^2)$. However, Barnes-Hut allows us to perform an approximation as described above in $O(N \log N)$. The key data structure that enables this speedup is the quadtree.

We can divide the Barnes-Hut algorithm into three main steps: (i) constructing the quadtree, (ii) traversing the quadtree to calculate the force being exerted on each particle, and (iii) updating the point locations and velocities. Each step is dependent on full completion of the previous step, so we can only parallelize within each step.

Constructing the Quadtree

We start by finding a bounding box that encapsulates the location of all the points. This bounding box is the root of our quadtree. Then, partition the space into four quadrants, with each partition represented by a child of the root in the quadtree. We insert points into the quadtree structure one at a time by finding the existing quadrant that the point should belong in. If the quadrant is empty we can insert the point into the quadtree; otherwise, we will recursively partition until each point is in its own quadrant. Internal nodes in the quadtree represent space partitions and contain the center of mass and total mass of particles in that space, and external nodes represent a particle. An example bounded box containing particles and its corresponding quadtree construction is depicted below in Figure 1.

Parallelizing the quadtree construction is difficult because the threads must update a common quadtree representation and pointer chasing when traversing trees takes longer memory read/write times with less locality. In addition, parallelizing over SIMD adds another layer of difficulty as points can lie in various levels and locations of the quadtree, leading to thread divergence. Building tree data structures, including this quadtree, is a naturally recursive process; however, recursion doesn’t work well on GPUs as there is too much overhead with

each thread in a warp needing to perform the same number of recursive calls, so we needed to build and traverse the quadtree iteratively.

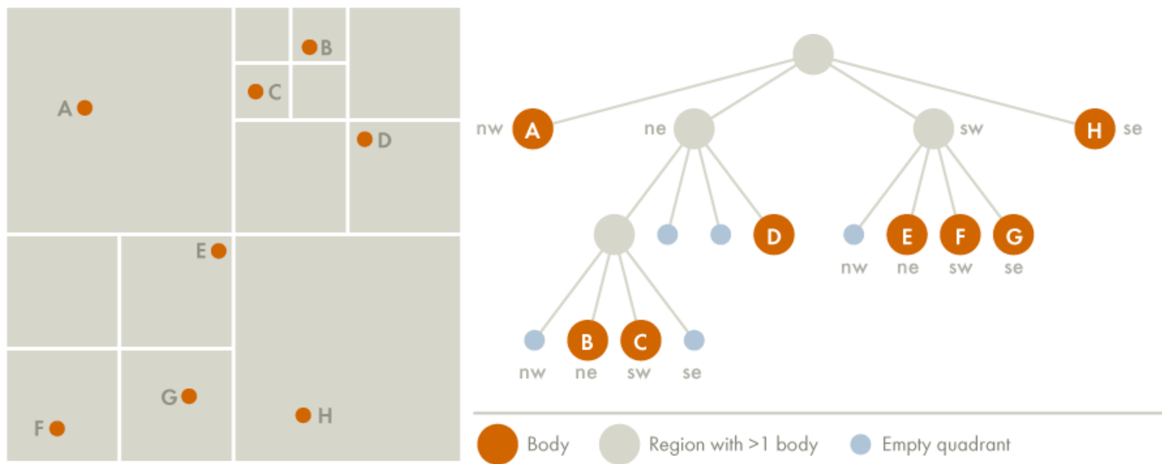

Figure 1 - Barnes-Hut quadtree example[3]

Despite these difficulties, there is still potential for SIMD parallelism. Points located in different quadrants and thus different parts of the quadtree can be inserted in parallel, as they do not interfere with each other's quadtree modifications. Furthermore, modifications due to point insertion only affect one existing quadrant, so most points will not interfere with each other in quadtree updates. In addition, we can implement the quadtree data structure with arrays to improve locality and minimize random memory accesses from pointer chasing. We describe this in more detail in the Approach section.

Force Approximations Traversing the Quadtree

To approximate forces, we use the parameter θ to determine what is considered 'far enough' to start grouping particles together. A particle/cluster of particles represented by an internal node *A* is 'far enough' from the particle *B* if the width of the space represented by *A* divided by the distance between *A*'s and *B*'s centers of mass is greater than θ. When $\theta = 0$, this is equivalent to using no approximation as each particle's individual gravitational effect is calculated on each particle. Figure 2 shows an example where a group of points are sufficiently 'far enough'

determined by parameter θ to be clustered together into one aggregate point, so the center of mass and mass of the marked inner node can be used for force calculation without needing further tree traversal.

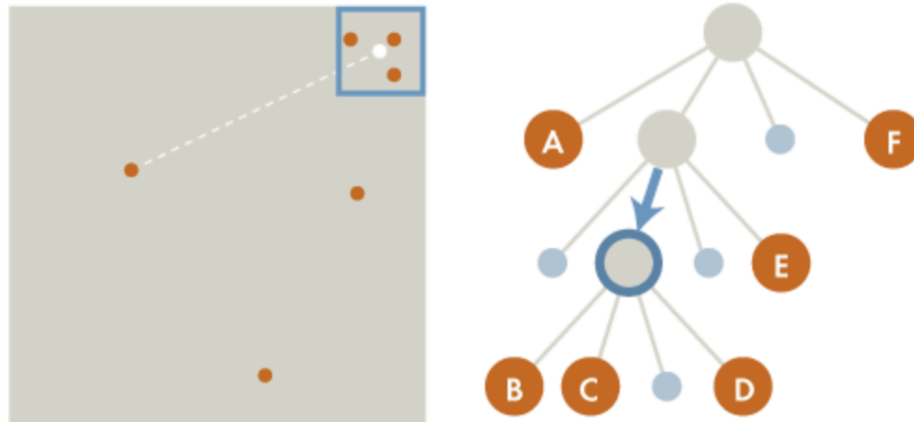

Figure 2 - Approximating Force from ‘Far Enough’ Points[3]

This step is data-parallel because we are only reading information stored in the quadtree, so there are no concurrency issues with respect to shared memory modifications. We can again parallelize across points, with each thread calculating the force exerted on a point; however, parallelizing over SIMD is difficult due to divergent thread execution since points located at different depths in the quadtree will require a different number of iterations in tree traversal, and different points can aggregate a different number of particles into clusters. As described above, a classic tree representation would also involve many random memory accesses from pointer chasing. Using the array representation will also improve parallelism here by increasing locality from removing pointer chasing.

Updating Particle Locations and Velocities

Once we have calculated the aggregate force acting on a particle, we can apply the uniform motion formula ($x_{new} = x_{old} + dx * timestep$) and Newton’s second law of motion ($F = ma$) to find

the updated location and velocity. This step is ideal for SIMD parallelization, as it is data-parallel and threads in the same warp can perform the arithmetic in lock-step.

Approach

We implemented Barnes-Hut sequentially on the CPU and parallelized with CUDA on the GPU. We targeted and tested our implementation on the NVIDIA GPUs in the lab.

CPU Implementation

Our sequential implementation is a naive Barnes-Hut implementation. The purpose of this implementation was to provide a benchmark for the CUDA implementation, so we did not implement any optimizations. This version uses a classic tree data structure with 'struct TreeNode' representing a node in the quadtree containing pointers to its children.

GPU Implementation

We modeled our implementation after the implementation presented by Burtscher and Pingali:

0. Read input data and transfer to GPU
for each timestep do {
    1. Compute bounding box around all bodies
    2. Build hierarchical decomposition by inserting each body into octree
    3. Summarize body information in each internal octree node
    4. Approximately sort the bodies by spatial distance
    5. Compute forces acting on each body with help of octree
    6. Update body positions and velocities
}
7. Transfer result to CPU and output

Figure 3 - Pseudocode Presented by Burtscher and Pingali[1]

Each bold step in Figure 3 above represents a separate kernel launch. Through reading Burtscher and Pingali's[1] findings, we discovered that much of the optimizations rely on the quadtree being implemented over arrays. Thus, our first attempt at a CUDA implementation

used arrays for the quadtree rather than the classic tree node representation for the quadtree. We implemented step 1 using the thrust library to find the minimum and maximum x- and y-coordinates for the bounding box. Steps 2 through 6 were more involved, so we will discuss them in more detail. In our code, we separated these steps into separate kernel launches so that the number of blocks and threads per block could be controlled on a per-kernel basis, which is especially important for Kernel 5 (`ComputeForces`), as it is designed to work with a one-warp-per-block design.

Targeting GPU Optimizations with Array Implementation of Quad Trees

Trees are commonly implemented with a struct representing a node in the tree, and pointers to children stored as fields in the struct. Thus, tree nodes can be stored anywhere in memory, with tree traversal causing pointer-chasing to random locations in memory. To increase locality, we remove the need for pointer-chasing by storing the quadtree data in contiguous sections of memory with arrays. Each field from the struct implementation is pulled out to be an array, with the ith index across the arrays corresponding to the same cell/particle in the quadtree as depicted in Figure 4 below.

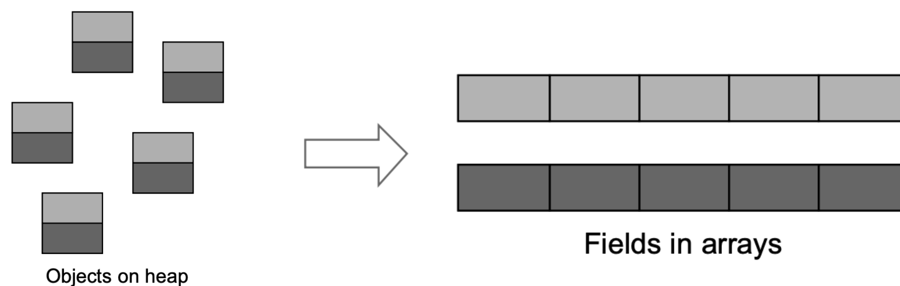

Figure 4 - Node Struct to Array Conversion[1]

We use a separate array `childNodes` to store indices of the children of an inner cell. An inner cell whose information is stored at index `i` in the field arrays will have the indices of where the information for the children nodes can be found at `i*4`, `i*4+1`, `i*4+2`, and `i*4+3`. A `-1` represents a null pointer, where there is no child node for a specified quadrant yet.

In addition, the information for particles is stored in the same array as the array for quadtree inner cells in the layout shown in Figure 5. We store particle information from the start of the array and inner cell information from the end of the array to allow us to easily differentiate between whether we are looking at a particle or inner cell based on only the index. This also reduces storage overhead, as we don't need to add particle information for external cell nodes that contain a particle, and can instead succinctly represent them with letting children be the particle itself.

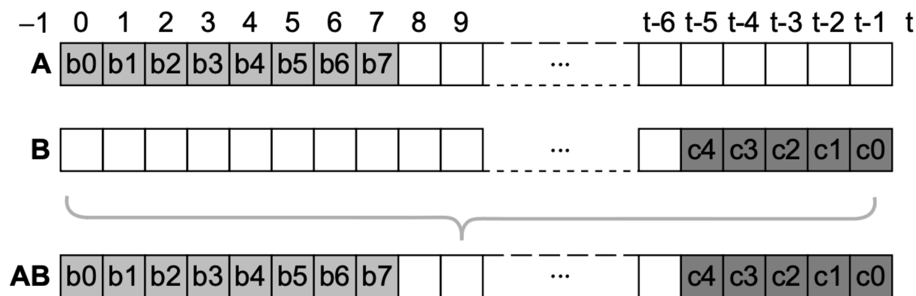

Figure 5 - Array Layout where Array A containing b0, b1, … are particles and Array B containing c0, c1, … are inner cells. A and B are concatenated, with the quadtree root stored at c0.

Algorithm Modifications for Parallelism

The sequential version combined step 3 with step 2 from Figure 3, keeping track of a running center of mass and total mass for each inner cell in the quadtree as points were inserted; however, the CUDA implementation splits these two steps into different kernel launches because of differing quadtree implementations. Namely, in the sequential version, a TreeNode

struct contains all the information at a node in the quadtree, so spatial locality for fields within the same struct meant it was ideal to update the node during point insertion once the node had been fetched from memory. With all the information stored in different arrays, we wanted to increase locality by minimizing excess memory loads during quadtree construction so that more relevant lines for building the quadtree could be kept in the cache. Thus, we split this into two different steps for the CUDA implementation.

Modifying Shared Quadtree over CUDA

When building the quadtree, each thread handles inserting one point into the quadtree that is shared by all threads across all the thread blocks. We used locks to handle race conditions from modifying shared memory. Since threads in the same warp run with SIMD execution in lock-step, we had to put careful consideration into when locks are acquired and released to avoid deadlock and lost updates. We locked on the `childNodes` array by using `-2` as a placeholder representing that a thread is currently modifying the subtree rooted at the quadtree node. Only the node which is being modified by adding new partitions or inserting a point is locked. Since two threads in the same warp may be trying to acquire the lock on the same cell at the same time, we avoided race conditions by using an `atomicCAS` to ensure that at most one thread held the lock for a node at any time.

Optimizing CUDA Implementation: Thread Divergence

Thread divergence is a major issue when building and traversing the quadtrees. Threads in the same warp may need to traverse a different number of nodes before reaching the desired cell or particle, and due to lock-step, all threads in the same warp need to wait for the thread that must perform the most node traversals to finish before proceeding to the next new instruction. In addition, introducing locks increases thread divergence as threads in the same warp trying to acquire the same lock will need to wait to acquire the lock sequentially, forcing all threads in that

warp to also wait. When building the quadtree, we want threads of the same warp to handle points that don't interfere with each other's quadtree modifications, to avoid waiting sequentially for locks in the same warp. However, when traversing the quadtree for force calculations, thread divergence only comes from traversing the quadtree, so ideally we want threads in the same warp to traverse the quadtree in approximately the same way. Points spatially close to each other are also on similar paths in the quadtree. This is where we introduced kernel 4, which sorts points such that spatially close points are sorted near each other, to reduce thread divergence. Kernel 4 does not affect correctness, only performance, and we leave it active by default unless otherwise specified in our report during experimentation and analysis.

Optimizing CUDA Implementation: Memory Accesses

The quadtree is stored in device memory, which is shared by all the blocks. Updates must be pushed to device memory for all threads across all blocks to see with the use of slow thread fences, but when blocks are executing kernels that only involve reading from the quadtree, these fences aren't necessary. In addition, the layout of the array leads to nodes being stored near their children, so loading in contiguous chunks of memory at a time helps decrease overhead from memory accesses.

Locks for nodes are also stored in device memory as they must be accessible to all threads across all blocks. Each attempt to acquire the lock and each lock release requires fetching from device memory, which are slow memory accesses. To reduce excess memory accesses which can lead to overall slowdown of memory accesses across all threads, we throttle the threads for attempting to acquire the desired lock. We do this by adding a `__syncthreads()` call before each new lock acquisition attempt so that all threads in a block wait for all other threads in the block to finish quadtree modification if they have a lock. It's likely that a thread in a different

warp in the same block has the lock, so this throttling decreases the excess memory accesses for lock acquisition while giving all threads in the block a chance to make progress.

Rendering

We used the CUDA rendering setup from assignment 2 to visualize the Barnes-Hut simulation over CUDA. We also used the CycleTimer from assignment 2 to measure performance.

Results

Performance Measurement Criteria

We measured performance by taking the CycleTimer provided in assignment 2 of the course to time each kernel and each time step iteration for the CUDA version, and we timed each time step iteration for the CPU version. We compared the runtime from the average of 20 iterations of the CUDA version against the CPU version to calculate speedup across varying problem sizes (number of particles in the simulation).

To measure simulation accuracy, we took the squared distance of particles for a given parameter θ after 20 timestep iterations of the CUDA simulation from their corresponding location after running the CPU version with $\theta = 0$ for 20 timestep iterations. We also took the squared difference in the resulting velocities, calculated by taking the square root of the sum of the squared x- and y-components of the velocity.

Speedup Across Problem Sizes

Our baseline for measuring speedup is the CPU implementation. We use $\theta = 0.9$ for these simulations, and take the runtime averaged across 20 timestep iterations for both the CPU and GPU versions. Since we launch a thread to handle each point throughout the algorithm, we expect that for simulating X points, we see an X-times speedup. The speedup graph is shown below in Figure 6.

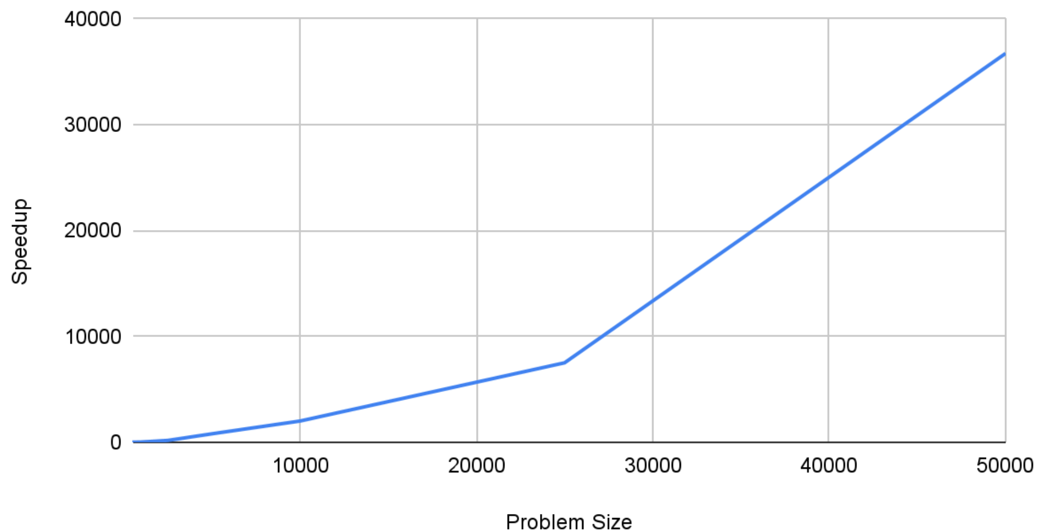

Figure 6 - Speedup of CUDA implementation over naive CPU implementation

We'd like to note that in the paper we reference by Burtscher and Pingali[1], their analysis uses an optimized serial CPU implementation, whereas we are benchmarking against a naive serial implementation to analyze the potential improvements from optimizations on top of parallelization over GPU.

Similar to Burtscher and Pingali[1], we see that as the problem size increases, the speedup also increases at a greater rate because more points provides greater opportunity for parallelism. However, across the board, we observe non-ideal speedup, which we believe is due to poor SIMD utilization from thread divergence and multiple memory accesses. We found that as the number of particles increases, an increasing percentage of runtime is spent in the kernels that build the quadtree and traverse the quadtree to compute the total force (shown in Figure 7). These two kernels have the thread divergence and memory access issues discussed above, whereas the other kernels are fairly data-parallel and can easily perform arithmetic in lock-step.

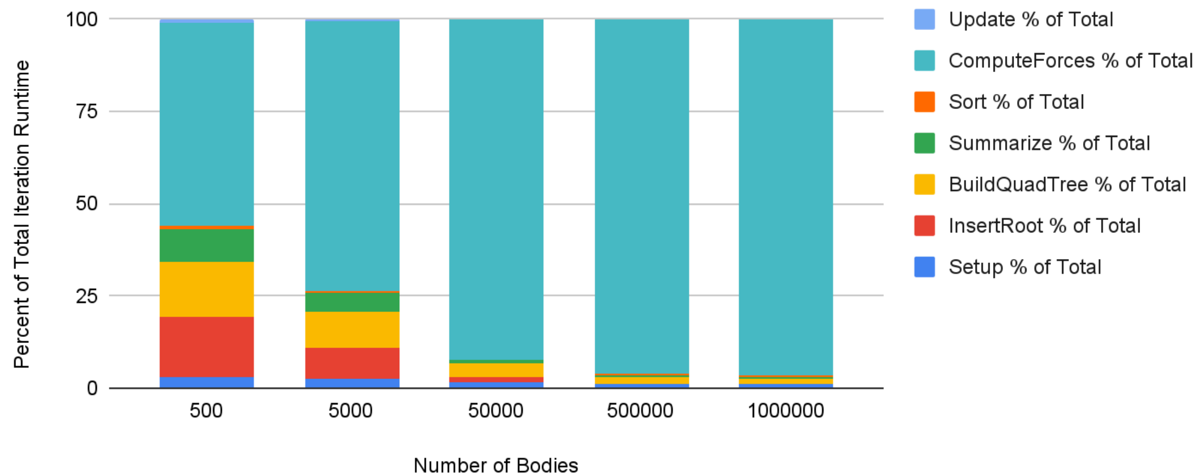

Figure 7 - `InsertRoot` represents the kernel for computing the bounding box. `Sort` represents the kernel 4 optimization. `Update` represents the kernel that updates point locations.

When we add the kernel 4 optimization which approximately sorts points spatially, we see that the percentage of runtime spent in the kernel computing the total force decreases significantly (seen in Figure 8 below). This optimization decreases thread divergence within warps, as threads in the same warp should be computing the force for points located near each other in the quadtree, so they should be performing very similar access patterns. This shows that thread divergence is in part responsible for the less than ideal speedup.

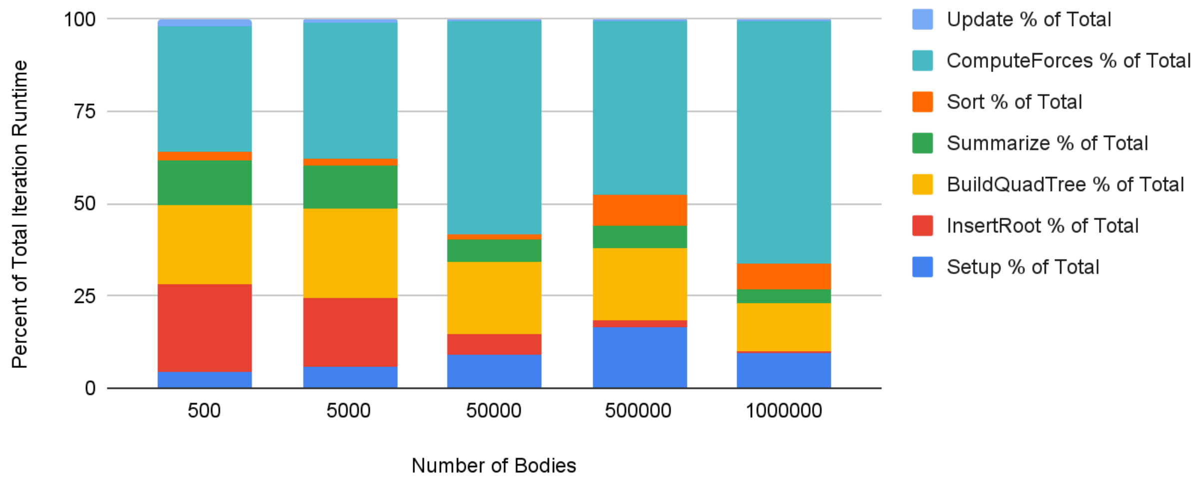

Figure 8 - `ComputeForces` takes a lower percentage of overall time with sorting enabled

We believe that locking and synchronization also adds overhead, as the kernel for building the quadtree takes a significant percentage of the runtime. In addition, the locking and synchronization introduces thread divergence and constant memory accesses as discussed earlier. During development, we found that without throttling threads by removing the synchronization points, we had a slowdown of around 1000 times, signifying that the memory accesses from trying to acquire and release locks produces a significant amount of overhead. Thus, we attribute the less than ideal speedup to locking, thread divergence, and constant memory accesses.

Kernel Breakdown Across Problem Sizes

The Burtscher and Pingali[1] paper broke a simulation timestep into six kernel launches, and our implementation used seven. We split the paper's kernel launch to compute the bounding box and insert a root node into two separate steps. Our kernels work as follows:

`Setup → InsertRoot → BuildQuadTree → Summarize → Sort → ComputeForces → Update`

Here, we analyze the impact that each kernel has on the amount of time needed to do a timestep iteration. We will analyze these results with and without the sorting kernel doing useful work, as it's primarily an optimization that coalesces memory accesses in the `ComputeForces` kernel.

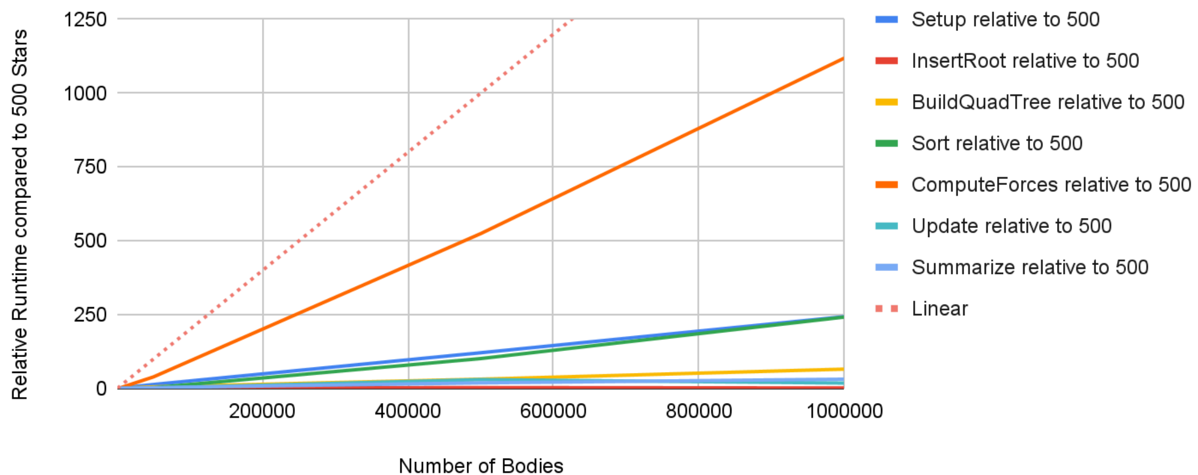

Figure 9 - All kernels scale roughly linearly, but at different rates

Figure 9 shows the relative increase in a kernel's runtime as the problem size increases. We chose 500 bodies as our base amount, so the runtimes for each kernel `k` are plotted relative to the time that `k` took on 500 bodies. So, a point at (1000000, 250) on the `Summarize` and `Setup` plots indicates that these kernels each took 250 times longer on 1 million bodies than they did on 500 bodies. If their workload increased linearly with the number of bodies, that would be the dashed `Linear` line.

What this plot shows us is that *certain kernels scale better than others*, but that they all scale roughly linearly (technically `nlogn` although that isn't easily seen in the plot). The

ComputeForces kernel takes the most time within each kernel run (as shown above in Figures 7 and 8) and scales the worst without the Sort kernel to speed it up. Introducing the Sort kernel significantly improves the scaling of ComputeForces:

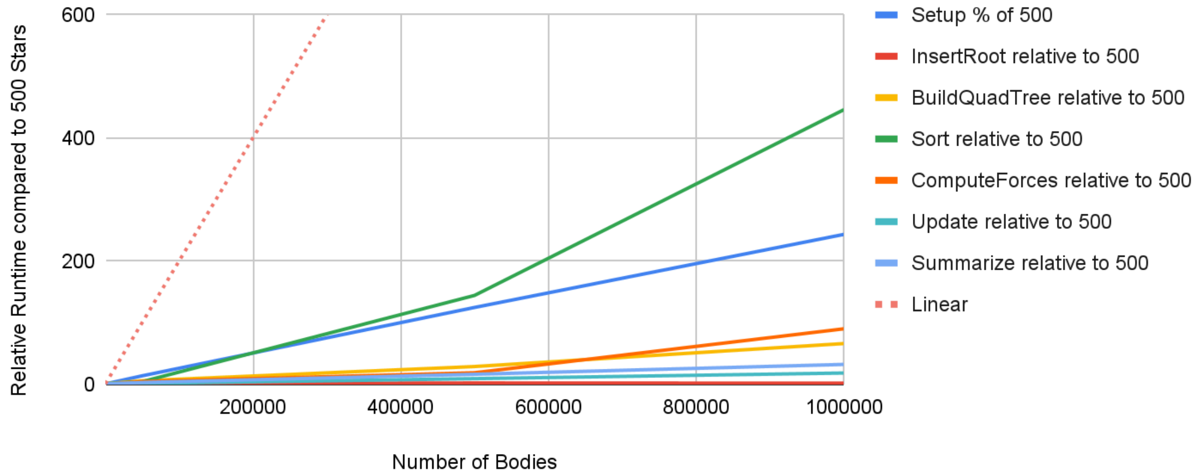

Figure 10 - ComputeForces scales significantly better with sorting

As one last illustration of the power of the Sort kernel to dramatically increase the raw speed of ComputeForces, Figure 11 shows how long ComputeForces takes in milliseconds for different numbers of bodies.

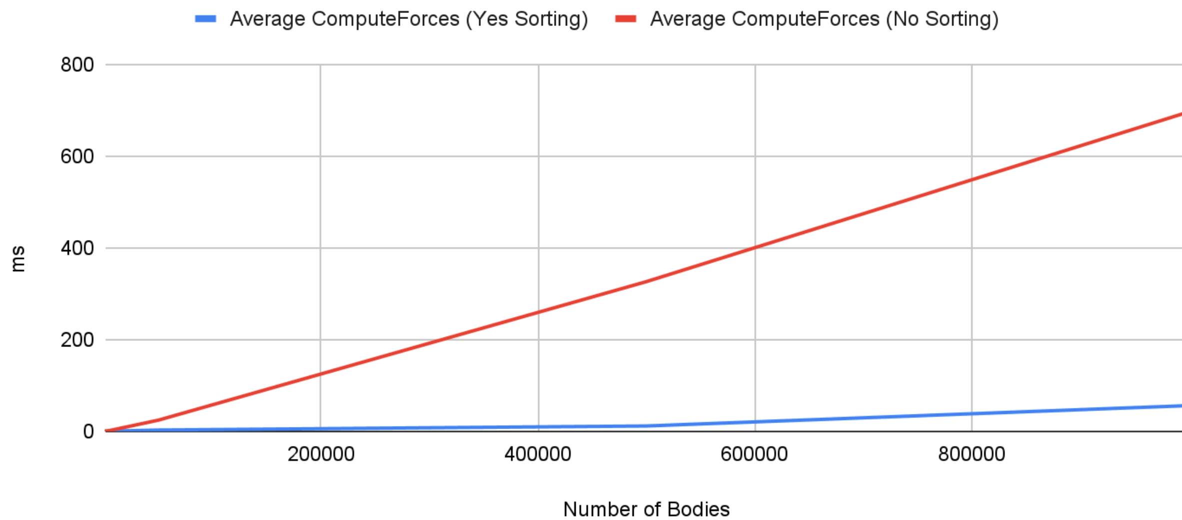

Figure 11 - Sorting dramatically increase the raw speed of `ComputeForces`

Iteration Time as a Function of θ Across Problem Sizes

We also checked how our average iteration time changed across problem sizes for different values of θ. As θ increases, more bodies can be ignored and approximated by the center of mass of a parent node. Because the force calculations work by traversing down from the root of the quadtree until they either hit a body or an interior node that can be used as an approximation of its subtree, increasing θ decreases the traversal length as a traversal hits an interior node that can be used much sooner.

We can see this in a plot of the relative runtime growth across problem sizes for different values of θ. This is a log vs log plot so that the separation between values of θ is apparent and not squished by large values.

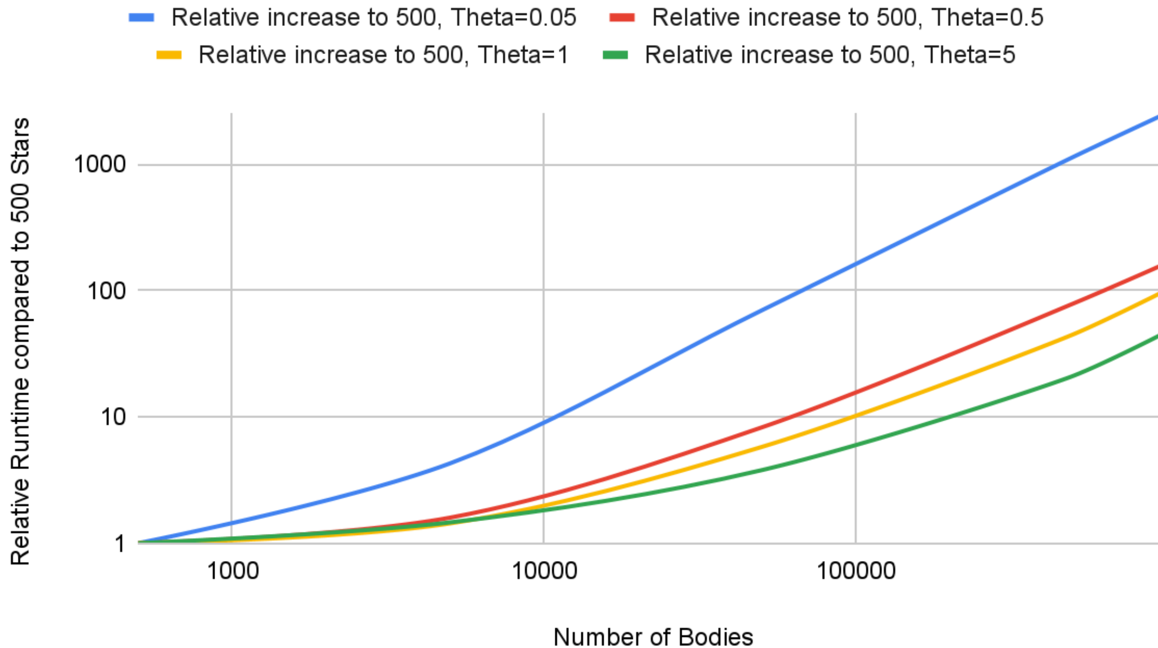

Figure 12 - Lower values of θ scale worse than larger values of θ

This shows that *lower values of θ scale worse* than higher values of theta. This makes sense and matches our expectations, as small values for θ cause the behavior of Barnes-Hut to approach a naive $O(n^2)$ solution as very few nodes can be approximated. Large values of θ cause most nodes to end their quadtree traversals very quickly, and thus the program scales better as the number of bodies increases.

Just to demonstrate the effect that θ can have on the absolute runtime of the program, here's how the runtime of a timestep iteration on 50,000 bodies changes with θ:

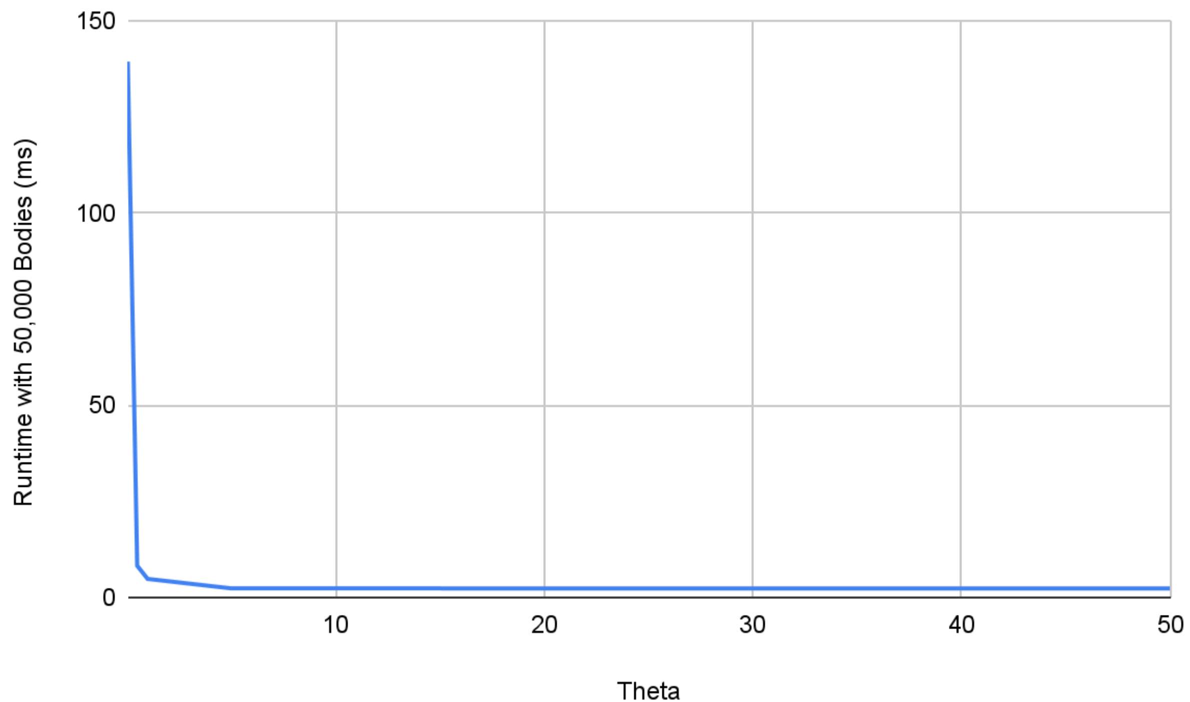

Figure 13 - Increasing θ dramatically increases raw runtime with 50,000 bodies

Accuracy as a Function of θ Across Problem Sizes

Our baseline for measuring accuracy is the CPU implementation for θ=0. For these simulations, we perform the cost calculation described above after 20 timestep iterations. We generated sets of 100, 1000, 5000, and 10000 points with initial velocity set to zero, randomly located in the box bounded by (-1024, -1024) and (1024, 1024), with mass randomly assigned between 0.1 and 100. We tested against the following values of θ: 0.05, 0.5, 1.0, 5.0, 50.0. Running the sequential version with θ=0 for 20 timestep iterations as a baseline wasn't reasonable for much more than 10000 points, as the CPU implementation is a recursive $O(N^2)$ implementation. The graphs displaying the position and velocity cost, calculated as described above, are shown below. We'd like to note that this magnitude of cost is to be expected given that points are

initially up to 2048 units of distance away, and these costs sum up squared distance of all points.

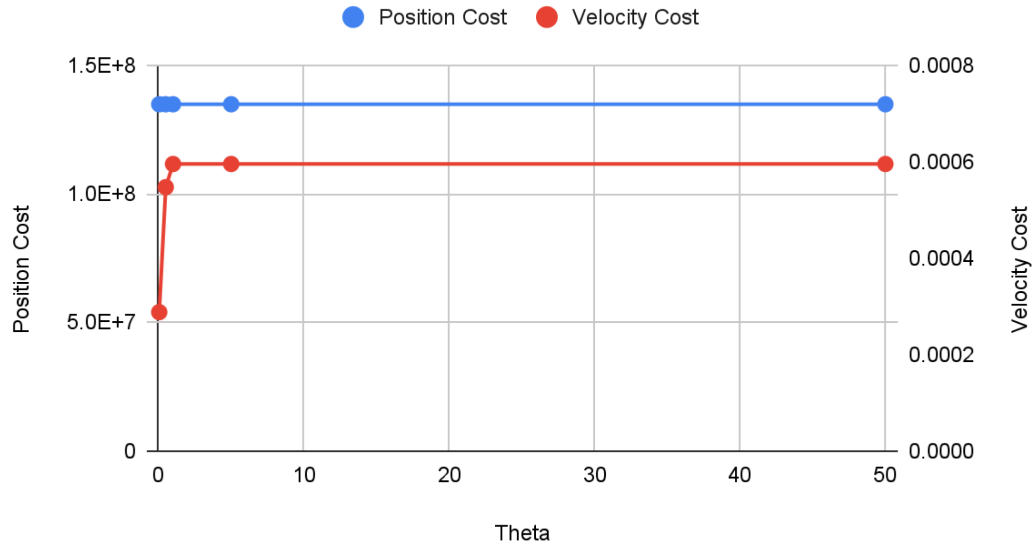

Figure 14 - Position and Velocity accuracy for 100 particles across varying theta after 20 simulation iterations

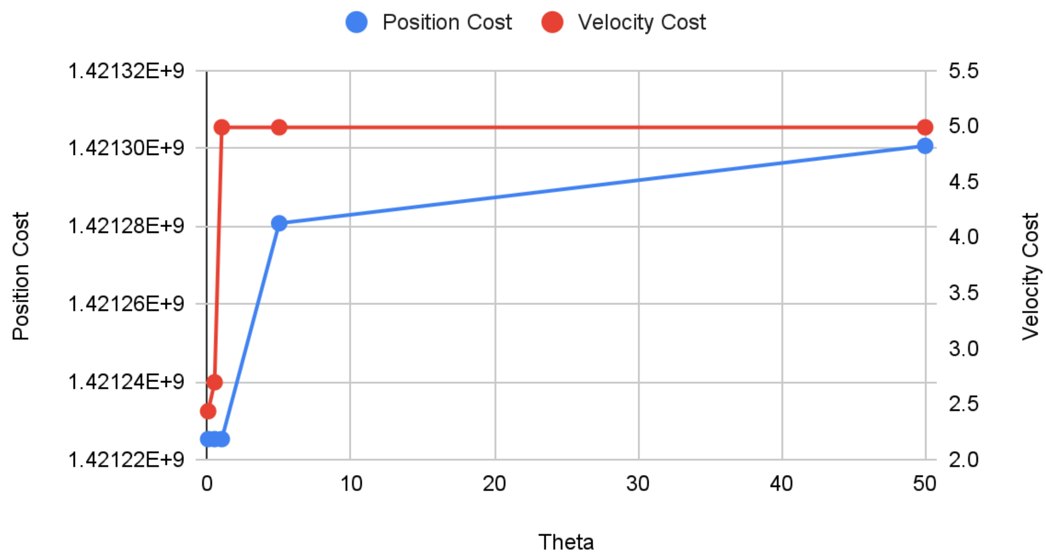

Figure 15 - Position and Velocity accuracy for 1000 particles across varying theta after 20 simulation iterations

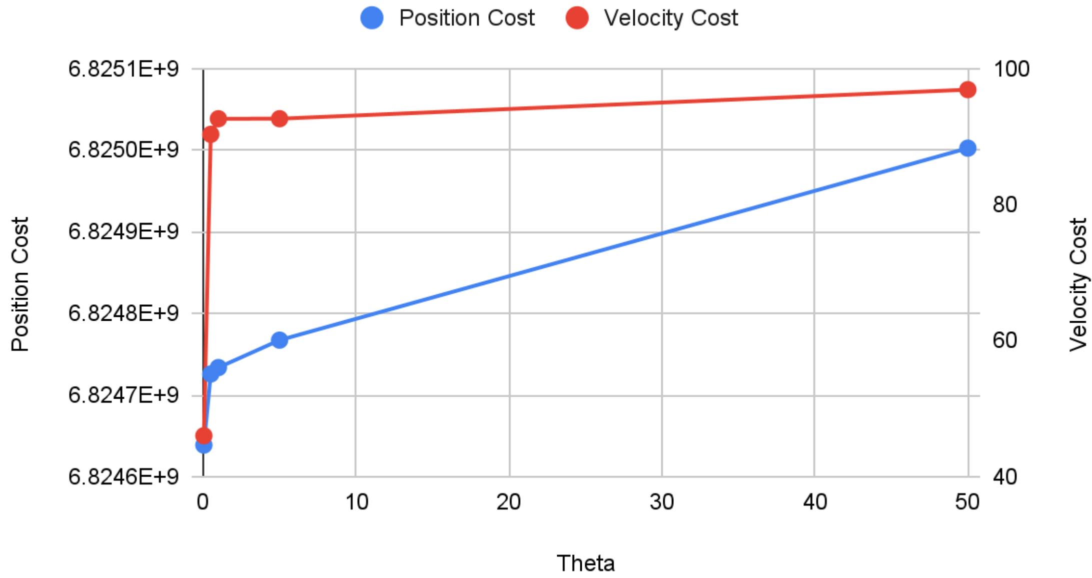

Figure 16 - Position and Velocity accuracy for 5000 particles across varying theta after 20 simulation iterations

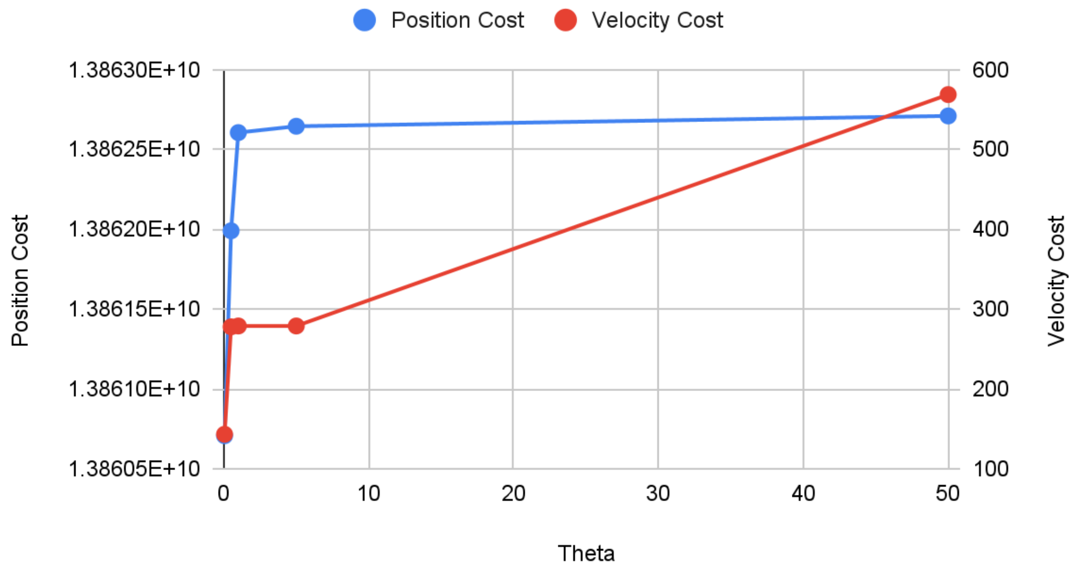

Figure 17 - Position and Velocity accuracy for 10000 particles across varying theta after 20 simulation iterations

As expected, as parameter θ increases, the cost increases meaning that the approximation is straying further from the true result. In addition, as the problem size increased, the magnitude of the cost increased proportionally. Interestingly, for each of the problem sizes, the cost increases in a sigmoid curve shape as theta increases. At small theta, the cost stays around the same, before hitting some inflection point and increases significantly, before once again staying around the same. We believe this is because at some theta, a cluster of particles are grouped as 'far enough' away, leading to the sudden increase in cost as more points are suddenly approximated. For smaller problem sizes, the cost is significantly less and also are less sensitive to changes in theta, because the points are more likely to be more dispersed, so the distances are far enough that they are classified as 'far enough' for most values of theta.

Conclusion

Given the speedup results we observed over a naive sequential CPU implementation, we believe that using the GPU is a good choice to achieve parallelism. Although there are difficulties with thread divergence and random memory accesses, the benefits from parallelizing over the GPU still outweighs the overhead from these issues.

References

[1]https://iss.oden.utexas.edu/Publications/Papers/burtscher11.pdf

[2]https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-31-fast-n-body-simulation-cuda

[3]http://arborjs.org/docs/barnes-hut

[4]https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Contribution Breakdown

We feel that the work was split evenly (50/50). Our contribution can be broken down as follows:

Contribution	Primary Contributor
Project Proposal	Aimee
Sequential CPU Barnes-Hut Implementation	Aimee
Snapshot visualization of CPU version	Aimee
Base CUDA Barnes-Hut Implementation	Jackson
CUDA Kernel 4 Optimization	Jackson
Thread throttling optimization	Aimee, Jackson
Visualization of CUDA version	Jackson
Midpoint Report	Jackson
Debugging race conditions in CUDA	Aimee, Jackson
Testing/experiment environment setup (creating test cases, cost metric script, instrument to input/output file)	Aimee
Final Report (minus analysis)	Aimee
Data Collection	Aimee, Jackson
Final Report - Data Analysis	Aimee, Jackson
Website	Jackson