6-26-2019

Efficient GPU Tree Walks for Effective Distributed N-Body Simulations

Jianqiao Liu
Purdue University

Michael Robson
University of Illinois Urbana-Champaign, mrobson@smith.edu

Thomas Quinn
University of Washington

Milind Kulkarni
Purdue University

Efficient GPU Tree Walks for Effective Distributed N-Body Simulations

Jianqiao Liu
Purdue University, USA
liu1274@purdue.edu

Michael Robson
University of Illinois Urbana-Champaign, USA
mprobson@illinois.edu

Thomas Quinn
University of Washington, USA
trq@astro.washington.edu

Milind Kulkarni
Purdue University, USA
milind@purdue.edu

ABSTRACT

N-body problems, such as simulating the motion of stars in a galaxy, are popularly solved using tree codes like Barnes-Hut. ChaNGa is a best-of-breed n-body platform that uses an asymptotically-efficient tree traversal strategy known as a dual-tree walk to quickly determine which bodies need to interact with each other to provide an accurate simulation result. However, this strategy does not work well on GPUs, due to the highly-irregular nature of the dual-tree algorithm. On GPUs, ChaNGa uses a hybrid strategy where the CPU performs the tree walk to determine which bodies interact while the GPU performs the force computation. In this paper, we show that a highly-optimized single-tree walk approach is able to achieve better GPU performance by significantly accelerating the tree walk and reducing CPU/GPU communication. Our experiments show that this new design can achieve a 8.25× speedup over baseline ChaNGa using a one node, one process per node configuration.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**; • **Applied computing** → **Astronomy**.

KEYWORDS

N-body problems, Heterogeneous system, Distributed system, GPU, Tree traversal

ACM Reference Format:
Jianqiao Liu, Michael Robson, Thomas Quinn, and Milind Kulkarni. 2019. Efficient GPU Tree Walks for Effective Distributed N-Body Simulations. In *2019 International Conference on Supercomputing (ICS '19), June 26–28, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3330345.3330348

1 INTRODUCTION

One important class of simulation problems is the *n-body problem*, which computes interactions between particles in a system to evaluate the effects of forces between those bodies such as gravity, electrical charge, etc. Perhaps the most classic example of an n-body simulation is modelling the self gravity of astronomical bodies—stars in a galaxy, for instance. As with all n-body problems, the naïve approach to computing the gravitational force is the direct approach: for each body in the system, compute the force acting on it from the other $n - 1$ bodies, resulting in an $O(n^2)$ algorithm.

In 1986, Barnes and Hut proposed an approach that has since become a standard way of performing n-body gravitational simulations: a *tree code* [1]. This approach leverages the fact that the gravitational force of a group of bodies can be approximated by a multipole expansion[1], the lowest term of which is proportional to the total mass divided by the square of the distance, and higher order terms drop off with successively higher powers of the distance. Tree codes use a spatial tree (classically, an octree) to capture the spatial relationship between bodies. To compute the forces on a body, the body traverses the tree, computing approximate forces from bodies that are far away by interacting with a node of the octree that encompasses all of those far-away bodies, and exact forces from bodies that are close by. In this way, the $O(n^2)$ algorithm becomes an $O(n \log n)$ algorithm. At a high level, this algorithm can be thought of as two interleaved components: a *tree walk* portion that identifies what forces need to be computed for a body, and a *force computation* portion that computes the forces on that body. Note that a *body* is usually called a *particle* in gravitational simulation, and a *node* in the tree corresponds to a *cell* in the space. We use them interchangeably in this paper.

Because cosmological simulations can involve millions or even trillions of bodies, distribution is a key approach to scaling up computation. Over the years, there have been many frameworks developed to perform distributed tree code–based n-body simulations [22, 23, 25]. One of the most advanced, and most efficient, is *ChaNGa* [14]. This framework, based on Charm++ [10], works by breaking up the large spatial tree required by a tree-code into *tree pieces*. Each of these tree pieces represents a sub-tree of the overall spatial tree, and hence a subset of all the bodies in the simulation. These tree pieces can then be distributed and executed on different nodes in a system, combining the forces from "local walks" (bodies in a tree piece interacting with other bodies in the same tree piece) and "remote walks" (bodies interacting with remote tree pieces). More details of ChaNGa and the underlying Charm++ runtime system are in Section 2.

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330348

[1]The multipole expansion is the sum of spherical harmonics of the mass distribution. ChaNGa uses fourth (hexadecapole) order Cartesian multipole expansions in the force computation [21, 23].

The local walks in ChaNGa are implemented using a *dual tree* algorithm: rather than having each body perform separate tree walks, resulting in *particle-cell* interactions (when a body determines whether all the bodies in a given subtree are far enough away) and *particle-particle* interactions (when a body interacts directly with another body to compute forces), the dual-tree walk also leverages *cell-cell* interactions. These cell-cell interactions can quickly determine if *all* the bodies in one subtree are far enough away from the bodies in another subtree to allow for approximate computation (A variant of this approach is also used in another classic n-body algorithm, the Fast Multipole Method [4, 17]). This optimization further reduces the complexity of the tree walk to $O(n)$ compared to the original "single-tree" implementation. Note that this complexity difference affects only the tree-walk portion of the algorithm; the two variants perform asymptotically similar numbers of force computations.

While the dual-tree approach is highly effective for CPU-only computation, it suffers from several drawbacks when trying to take advantage of the GPUs that are increasingly a part of distributed systems. Dual-tree walks derive their asymptotic benefit from performing the tree walk for numerous points as part of a single computation. While this is the source of the asymptotic win versus single-tree implementations, it also reduces the amount of parallelism available in the computation (all of the single-tree walks are parallelizable, but the dual-tree walks for all the points must be done as part of a single computation), underutilizing a GPU's massive parallel resources. Moreover, the dual-tree algorithm is very control-heavy, leading to *control divergence* that compromises a GPU's SIMT execution model, further reducing utilization. As a result, ChaNGa does not directly implement a dual-tree walk on GPUs. Instead, ChaNGa separates the tree walk step from the force computation step: the *CPU* performs the dual-tree traversal to determine which bodies each other body needs to interact with directly, building *interaction lists*, then the *GPU* uses these interaction lists to perform the force computation. Because the interaction lists are dense, regular structures, the force computation step can be efficiently performed on the GPU.

ChaNGa's CPU/GPU approach is an effective way to exploit GPUs in its distributed computation. However, it means that the GPU's parallelism cannot be leveraged for the tree-walk portion of the computation. Moreover, because the CPU must perform the tree walks and then send the interaction lists to the GPU, a significant amount of time needs to be spent in this communication, further reducing efficiency.

Contributions. In this paper, we observe that it is important to match the algorithm to the target hardware. While dual-tree approaches have attractive asymptotic properties that are effective for CPU computation, these asymptotic behaviors are counterbalanced by the specific requirements of efficient GPU computation: the need for massive parallelism and regular control flow. Hence, in this paper, we make a key change to ChaNGa's GPU implementation: we use an efficient *single-tree* algorithm to perform local tree computations.

By using a single-tree computation, we can move both the tree walk and force computation steps to the GPU, eliminating the communication bottleneck inherent in transferring interaction lists between the CPU and GPU. While single-tree walks are *still* irregular, we adapt recent developments in GPU tree walks from Goldfarb et al. [6] and Liu et al. [13] that show that it is possible to implement these tree walks in a way that limits control divergence and hence can be made highly efficient on the GPU. By combining these two effects—reduced communication costs and reduced control divergence—our approach is able to overcome the higher asymptotic complexity of the single-tree approach to deliver a more efficient GPU implementation of ChaNGa, representing the fastest known configuration of ChaNGa. We show, across several benchmarks, our implementation is 11.29× faster than the original ChaNGa in the best case, and 8.25× faster on average (using a one node, one process per node configuration).

```
void BarnesHut(particle, node) {
  if (far_away(particle, node)) {
    calculateGravity(particle, node); }
  else if (isBucket(node)) {
    for (p : node.particles())
      calculateGravity(particle, p); }
  else {
    for (child : node.children())
      BarnesHut(particle, child); }}
```

Figure 1: Barnes Hut pseudocode

2 BACKGROUND

This section provides background on n-body codes in general, and Barnes-Hut in particular. It also describes Charm++ [10], the distributed runtime system that ChaNGa (described in the next section) is built on.

2.1 n-body codes

The naïve approach to perform an n-body simulation is to have each particle in the space directly compute the gravity with all the rest of the particles. Each particle requires $O(n)$ computation, and the overall complexity is $O(n^2)$. The Barnes Hut algorithm (BH) uses an *octree* in three dimensional space to organize the particles. The topmost node (the *root*) represents the whole space, and its eight children represent the subspaces. The space is recursively divided until the number of particles in each node is below a threshold. In the simulation, a particle traverses the space from the root. If the center of mass of one internal node is sufficiently far away from the particle, the particles contained by that node are treated as a single particle whose position and mass are captured by the internal node's attributes. Otherwise, the particle needs to traverse each child of the internal node. The process is repeated until no more nodes remain (Figure 1).

2.2 Charm++

Charm++ is an adaptive runtime system based on the principles of asynchrony and overdecomposition. Programs written in Charm++ are comprised of migratable C++ objects, known as chares. These chares can be grouped into collections, known as chare arrays. Charm++ is a message-driven model in which these chares communicate via asynchronous messages, or entry methods. This asynchronous model automatically enables adaptive overlap of communication and computation, including on the CPU and GPU [8].

These features of adaptivity, asynchrony, migratability, and overdecompositon, etc., when combined with the introspection in the runtime system, enable various capabilities such as dynamic load balancing and fault tolerance, both of which ChaNGa leverages. The Charm++ system is a mature scalable parallel programming model that underlies several successful applications.

GPU Manager. Charm++'s GPU Manager library was initially designed for use in ChaNGa to accelerate various kernels [8]. The library is focused on offloading GPU kernels and data copies without blocking CPU code. It is able to achieve this by asynchronously invoking various stages of a kernel's execution (data copy in, kernel launch and execution, data copy out) between the execution of entry methods. The GPU manager utilizes streams to create overlap between these different phases of different kernels. This is done to promote overlap of communication and computation both on the device and between the host and the device. An important note is that the existence of this library does not prevent interested users from independently taking advantage of the entire CUDA platform [16]. The motivation behind this library is to provide the user with various useful functionalities without impeding GPU performance. These extensions include stream creation, data movement and management, kernel launch, synchronization, and notification. The GPU manager library also initializes a pinned memory pool that is made available to accelerate copies to and from the device. Finally, there are extensions to the API for integration with Charm++'s profiling and visualization tools. Most of these features were developed to accelerate ChaNGa code through a process of co-design.

To use the GPU manager library, the program registers its kernel functions and data buffers with the Charm++ runtime system. The user still has complete control over the launch parameters of the kernel. The runtime system then leverages its introspective nature and automatically initiates data copies greedily both in and out of the device and overlaps these copies with kernel launch and execution. When the kernel is finished executing the runtime system sends a notification message in the form of a callback function when the data is copied back and ready for reuse.

3 CHANGA

This section describes ChaNGa, the n-body simulation code built on top of Charm++. We begin by describing the high-level structure of ChaNGa. We then describe its dual-tree approach to traversing trees, and explain the unsuitability of GPUs for performing dual-tree walks. Finally, we explain how ChaNGa *currently* supports GPU execution.

3.1 ChaNGa structure

ChaNGa implements a large number of features necessary for modelling astrophysical problems including periodic boundary conditions, individual timestepping, and gas dynamics via Smooth Particle Hydrodynamics (SPH) along with a variety of equations of state. Here we focus on the implementation of the gravity calculation within ChaNGa. The gravity calculation proceeds by first decomposing particles into spatial domains, then building a tree over the whole volume, and then traversing the tree with a Barnes-Hut-like algorithm to calculate the gravitational forces on each particle. Since each domain contains a contiguous part of the tree, domain construction and tree building are intimately linked.

Tree construction. ChaNGa divides up the computational volume with an octree similar to that described in the original Barnes-Hut paper. One difference is that the tree is implemented as a binary tree with divisions alternating in the X, Y, Z dimensions. The root of the tree is constrained to be cubical, and every third level of the tree contains cubical nodes. The leaves (referred to here as "buckets") of the tree contain a small number of particles (at least 12, by default, referred to here as the "bucket size"). All nodes of the tree contain multipole moments, up to hexadecapole, of the mass distribution of the particles within that node, which are used to calculate forces due to all contained particles when the node satisfies the *opening criterion*. The *opening criterion* determines when to continue traversing a portion of the tree. Alternatively, its converse can be thought of as a *truncation criterion* that determines when to stop traversal.

The tree is constructed by first assigning each particle a key corresponding to a location on a space-filling curve (by default the Peano-Hilbert curve). The particles are then placed in tree-order via a parallel sort. During the sort, particles are also divided among *tree pieces* (each of which is a Charm++ chare). The Charm++ run time system then distributes these tree pieces among the processors in order to balance the computational load. Because each tree piece has its boundary information, it is able to independently construct a tree containing its particles all the way up to the root (see Figure 2).

3.2 Dual-tree traversals

The high-level pseudocode for n-body codes presented in Section 2.1 is for *single-tree* traversals. The structure of the algorithm is, roughly, "for each body in the system, traverse the spatial tree to compute forces"; in other words, there is one tree that is used to accelerate the traversals. One way to think about how this computation is structured is that it is based around *particle-cell* interactions, where a body visits an interior node of the octree to determine whether it should continue visiting that portion of the space; and *particle-particle* interactions where, once a body reaches a leaf node of the octree, it directly interacts with the bodies in that node to compute forces.

An algorithmic advance that reduces the complexity of the algorithm is to replace the *outer loop* of the high level n-body algorithm with a different approach. Instead of simply looping over all the bodies in the system to traverse the tree, *dual tree* algorithms take advantage of the fact that the bodies are already arranged in a spatial tree to accelerate the process. Dual-tree algorithms use two trees (which are often identical): the *body* tree that represents the bodies in the "outer loop" of the force computation, and the *target* tree (inner tree) that represents the original spatial tree.

Dual-tree algorithms introduce a third kind of interaction to the particle-cell and particle-particle interactions in single-tree traversals: the *cell-cell* interaction. In this interaction, an interior node from the body tree (call it node X) interacts with an interior node from the target tree (call it node Y). If *none* of the bodies in node X need to interact with *any* of the bodies in node Y, then none of the bodies in node X need to continue visiting this portion of the spatial tree. Crucially, this computation can be done by using

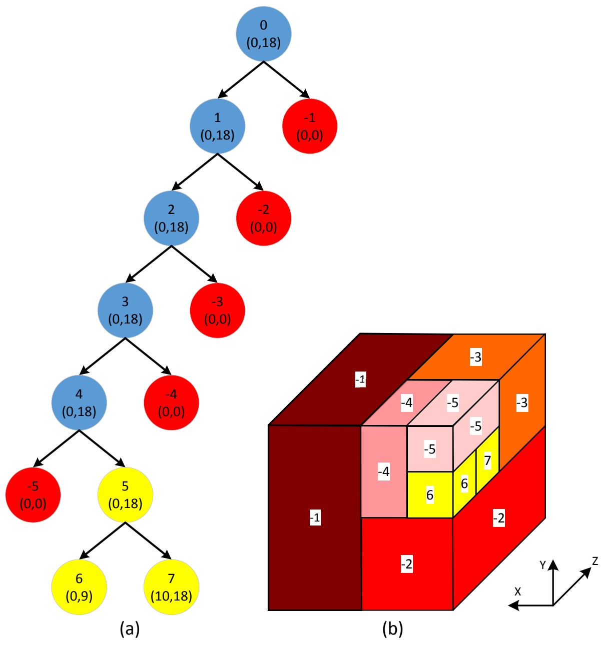

Figure 2: (a) Tree structure: Each chare builds its piece of the tree starting with internal nodes (yellow), eventually getting to boundary nodes (blue) which have non local children (red) and are duplicated on other tree pieces. The top boundary node is the root which is duplicated on all tree pieces. Numbers in the nodes in this figure correspond to node index (range of contained buckets). (b) Spatial representation: The cube represents node 0's space. The boundary nodes (blue) are replaced by their children (red and yellow). We differentiate a non local node's distance to the local tree-piece by its color. Darker colors indicate farther nodes. Nodes include their children (e.g., Node 5 includes Nodes 6 and 7.)

summary information about node X, amortizing the cost of this interaction across *all* the bodies in node X. If any body in node X needs to interact with Y, then the cell-cell interaction decomposes into new cell-cell interactions: the two child nodes of X perform cell-cell interactions with each of the child nodes of Y (i.e., four new cell-cell interactions are performed), effectively continuing the traversal of the tree.

Note that this recursive procedure means that the opening criterion for tree traversal can efficiently be calculated across multiple bodies in the system with a single cell-cell interaction. Note that the dual-tree opening criterion is a little bit looser than the opening criterion in the single-tree traversal, as cell-cell interactions stop only if *no* body in the two cells should interact. Nevertheless, even with this looser opening criterion resulting in larger walks of the tree, the amortization effect wins out: the complexity of dual-tree traversals drops from $O(n \log n)$ to $O(n)$. Figure 3 shows pseudocode for a dual-tree traversal. ChaNGa uses this style of traversal to accelerate its Barnes-Hut implementation, giving its CPU implementation attractive asymptotic complexity and beneficial cache behavior.

Why are GPUs unsuited for dual-tree traversals? A natural question is whether it makes sense to implement the dual-tree walk on GPUs, to take advantage of the GPU's greater parallelism compared

```
void dualTreeWalk(outerNode, checkList, interactionList){
  for (innerNode : checkList) {
    checkList.remove(innerNode);
    action = openCriterion(innerNode, outerNode);
    if (action == CONTAINED) {
      checkList.append(innerNode.children()); }
    else if (action == INTERSECT) {
      if (isBucket(innerNode)) {
        interactionList.plist.append(innerNode.particles
             ()); }
      else {
        checkList.append(innerNode.children()); }}
    else {   // TRUNCATED
      interactionList.clist.append(innerNode); }}
  if (isBucket(outerNode))
    calculateGravity(outerNode.particles(),
         interactionList);
  else {
    dualTreeWalk(outerNode.left, checkList,
         interactionList);
    dualTreeWalk(outerNode.right, checkList,
         interactionList); }}
```

Figure 3: Dual tree walk: the three possible actions from the opening criterion are 1) CONTAINED, meaning all particles in the outerNode will pass the acceptance criterion, 2) INTERSECT, meaning some may fail, and (implied) TRUNCATED, meaning all particles will fail the criterion.

to CPUs. Interestingly, the answer appears, at least for straightforward implementations, to be "no."

To understand why dual-tree traversals do not map well to GPUs, it is useful to recall why dual-tree traversals gain an asymptotic advantage over single-tree traversals in the first place: the cell-cell interactions. Their key advantage is that in a cell-cell interaction, *all* the bodies in a cell in the body tree can leverage a *single* interaction computation to determine whether they should continue traversing the target tree. Hence, work that would, in the single-tree case, need to be repeated across every body in the system, can be done just once. Consider, for example, the very first interaction, between a body and the root node of the octree. In the single tree walk, this interaction must be calculated for every body in the system, even though every body will determine that it must continue traversal; in the dual tree walk, this computation happens exactly one time. Fundamentally, this optimization saves on computation *at the cost of parallelism.*

The GPU gets its performance advantage from massive parallelism. Each individual thread on the GPU is relatively weak, with in-order execution, many hardware stalls, and, crucially, a SIMT (single instruction, multiple thread) execution model that penalizes control divergence, where different threads execute different instructions. All of these combine to mean that to get real performance gains out of a GPU, computations need large amounts of parallel threads that are all performing similar work. That is exactly what a dual-tree execution *does not* provide. A single thread performing a cell-cell interaction to amortize the cost of computing many particle-cell interactions provides no parallelism at all—indeed, the other hardware threads on a GPU are effectively wasted, when they could have been used to compute particle-cell interactions.

There are thus two drawbacks to a dual-tree execution on a GPU. First, as mentioned, the reduced parallelism means that a

GPU is performing slow single-threaded (or low–thread count) computation instead of highly multithreaded computation. Second, this low–thread count computation is still using the looser stopping criterion of a dual-tree walk, which means that it pays an additional penalty of traversing more of the tree than a single tree walk would. We build on both of these insights in Section 4 in our turn to single tree walks to improve performance. Of course, this still leaves the key problem for GPUs of minimizing control divergence, which our approach also addresses.

3.3 Hybrid CPU/GPU execution in ChaNGa

Given the unsuitability of dual-tree computations for GPUs, how *does* ChaNGa leverage GPUs in its current implementation? It does so by splitting the computation into two phases: a *tree walk* phase, which performs the dual-tree walk to determine which leaf nodes of the body tree need to interact with the target tree, building *interaction lists* that give, for each body, which other bodies and cells it needs to interact with, and a *force computation* phase which processes the interaction lists to compute the forces [8].

Notably, the tree walk phase is highly irregular, but consumes relatively little of the overall computation time, while, once the interaction lists are computed, the force computation phase is highly *regular*: the interaction list for each body can be stored in a dense array, and processing each interaction list requires the same arithmetic operations and can be readily parallelized. This leads to a natural separation of concerns: ChaNGa performs the tree walk on the CPU and sends the computed interaction lists to the GPU, and then performs the force computations on the GPU. Further improving matters, the interaction lists can be computed by multiple CPUs and sent asynchronously to the GPU to compute the forces[2].

While this is a very attractive approach to exploiting GPU parallelism, it does come with a drawback: the interaction lists are quite large, and as a result, communicating them from the CPU to the GPU is a significant expense. Indeed, the GPU often spends upwards of 46% of its total runtime merely transferring data from the CPU. Hence, even though the interaction list computation is highly regular and parallel, utilizing the GPUs resources well, the communication overheads still result in underutilization.

A second drawback of this approach is that it does not leverage increasing GPU resources well. The large amounts of parallelism in the GPU means that the force computation phase can complete very quickly. Indeed, in the hybrid approach, only 5% of the time is spent in GPU computation versus CPU computation. This means that adding more CPU cores can speed up the overall computation, by speeding up the tree walk phase that is the bottleneck. Conversely, however, a simple Amdahl's law argument shows that adding more GPU resources (either multiple GPUs per node or more powerful GPUs per node) without commensurately adding CPU resources will not result in much performance improvement. As GPUs are energy efficient (and cost effective) compared to CPUs, this precludes adding GPUs as an efficient way of improving performance.

[2]We note that recent distributed GPU implementations of the fast multipole method use a similar approach, where the interaction lists are built on the CPU (during tree construction) and then processed on the GPU [3, 12]. Because FMM builds its interaction lists differently, we would not expect our GPU traversal approach to be as effective (see Section 7).

What we would like is an alternative strategy for exploiting GPUs in ChaNGa. One where 1) we do not underutilize the GPU by spending significant amounts of time in communication; and 2) the GPU is the computational bottleneck, opening up avenues for performance improvement by increasing the number of GPUs. We discuss exactly this strategy next.

4 DESIGN

This section describes the key change we make to ChaNGa to more effectively utilize GPU resources: rather than using the hybrid CPU/GPU approach, we instead offload the entire local tree walk to the GPU.

4.1 Offloading the local tree walk

As we discussed before, moving ChaNGa's dual-tree algorithm to the GPU is unlikely to effectively leverage the GPU's parallelism. Thus, we return to the simple *single tree* computation, and offload that to the GPU. Because a single-tree walk has each particle traverse the tree independently, it features abundant parallelism, taking better advantage of a GPU's execution resources than a dual-tree walk would.

Importantly, a single-tree walk dramatically reduces the amount of data that needs to be communicated to the GPU compared to the interaction-list approach. In the interaction-list approach, the body data (positions, etc.) needs to be sent to the GPU to facilitate force computation. *In addition*, for each body an interaction list needs to be sent to the GPU. It is this extra data that leads to the 46% communication overheads on GPU side seen in the interaction-list approach. In contrast, when the entire tree walk is offloaded to the GPU, *only* the body data, in the form of the octree, needs to be sent to the GPU. Interaction lists do not need to be sent as they are computed as part of the GPU tree walk. As we will see in Section 6, this change means that by offloading the entire tree walk to the GPU reduces communication overheads from 46% to 5%.

Communication overheads are not the whole story, though. In the interaction-list approach, the force computation performed on the GPU is highly regular, matching the execution model of the GPU. On the other hand, a tree walk is inherently irregular, whether dual-tree or single-tree: even if the tree is laid out in a dense fashion (e.g., by linearizing the tree), the key feature that makes tree walks efficient is the opening criterion that prevents bodies from traversing the entire tree. As a result, bodies will make distinct, data-dependent decisions about which parts of the tree to traverse, leading to memory divergence—as different bodies touch different parts of the tree—and control divergence—as different bodies make different truncation decisions. Without addressing this divergence, moving the entire tree walk to the GPU is likely to result in severe underutilization of the GPU.

Recent developments in tree walks on GPUs give hope to this approach [2, 6, 13]. Burtscher et al. showed that a single-tree walk can be effectively placed on a GPU without incurring severe divergence through performing *warp*-level truncation instead of body-level truncation [2]. Rather than each body independently deciding whether to stop traversing a specific part of the tree, all bodies that are packed into a single GPU warp vote on truncation. If *any* body in the warp wants to continue traversing the tree, all bodies do. This prevents memory divergence, as all bodies in the warp consistently access the same parts of the tree.

Goldfarb et al. [6] generalize this approach through an optimization they call *lockstepping* that uses the same warp-level voting mechanism as Burtscher et al. but with explicit masks to block out threads that do not need to perform computation. They also add a technique called *autoropes* that uses an explicit *rope stack* to maintain the traversal order of the tree and avoid redundant visits to interior nodes as a traversal moves up and down through the octree. Finally, Liu et al. [13] refine Goldfarb et al.'s approach to minimize the amount of state needed to track the masks and the rope stack, reducing the number of registers required to perform tree traversal, thus allowing the GPU to perform more concurrent traversals.

Putting all of these advances together, we find that replacing the CPU-based tree traversal and GPU-based force computation with a *GPU only* single-tree traversal leads to better performance overall, dramatically reducing communication overhead, while controlling divergence sufficiently such that the overall GPU computation time for doing the entire tree walk is comparable to the time for the force computation only.

4.2 Complexity concerns

One notable drawback to switching to a single-tree traversal on the GPU is that we sacrifice the asymptotic complexity advantages of the dual-tree traversal, which the interaction-list approach preserves. For many inputs, the ability of our GPU-only approach to fully exploit a GPU's massive parallelism outweighs the asymptotic complexity disadvantage (since, of course, constant factors matter). However for large inputs, it is possible that the interaction-list approach will win out. We note, however, that the design of ChaNGa, and the principles of Charm++, limits this: as the input scales up, we expect the input to be broken up into more tree pieces, limiting the size of the local tree walks, and hence limiting the downside of the $O(n \log n)$ complexity of the single-tree approach. Typically, there are more tree pieces than the number of processors to benefit from overlapping CPU/GPU computation and load-balancing. Thus the tree piece size is smaller than the GPU memory limit.

We want to emphasize that the dual- vs. single-tree complexity comparison is mainly about the tree walk part. The force calculation itself is not affected by the tree walk strategy. Interestingly, although the dual-tree walk wins in complexity, our results show that single-tree walk requires less particle-to-particle calculation due to the tighter opening criterion (Table 2).

4.3 Further optimizations

Our current implementation only examines how offloading local tree walks to the GPU can improve performance. We make minimal changes to other portions of ChaNGa. However, our strategy does open up further opportunities for performance improvement. While we have not yet implemented these techniques, we discuss them here.

Overlapping with remote work. Our approach to offloading tree walks to the GPU applies only to the local tree walk performed by a tree piece. The bodies in that tree piece still need to account for forces acting on them by remote parts of the tree. Because these remote computations are relatively small portions of the overall computation, we leave these computations completely to the CPU, and preserve their dual-tree nature. This design fits naturally into the existing ChaNGa architecture, which separates local and remote computations.

We note that offloading the local tree walk to the GPU does offer up a potential performance advantage, as shown in Figure 4. Rather than the CPUs devoting time to performing local tree walks and computing interaction lists, in our approach, CPU resources can be fully devoted to remote tree walks, leading to a potentially large performance improvement from overlapping remote and local work. Our current implementation has not modified ChaNGa's scheduler, however, so we currently do not exploit this opportunity.

Intermediate time steps. A real-world n-body simulation will typically simulate the behavior of bodies for multiple time steps. From one time step to the next, most bodies have not moved significantly, and hence ChaNGa adopts an incremental strategy where in "major" time steps, all bodies are simulated, while in intermediate time steps, only a subset of bodies have their full forces recomputed.

In the current implementation the interaction lists for all bodies in a bucket are constructed and sent to the GPU for intermediate time steps (because a dual-tree walk naturally computes all of them), even though only some of the bodies have their interaction lists processed for force computation. Thus, ChaNGa's current implementation pays a large CPU-to-GPU communication overhead for all time steps, even when only some bodies are being updated.

Our single-tree approach is a natural fit for exploiting this discrepancy: we only perform tree walks for whichever bodies are sent to the GPU, allowing us to only compute interaction lists for the subset of bodies that are updated in the intermediate time step.

5 IMPLEMENTATION

This section describes some of the implementation details of our single tree walk, culminating in pseudocode specifying the GPU implementation.

Bucket-to-node opening criterion. The opening criterion, or truncation condition, is the computation that decides whether one or a batch of particles need to traverse the subtree rooted at a node. It is the key factor of any tree traversal algorithm. To conduct the force computation efficiently, ChaNGa adopts a node-to-node opening criterion. When walking the tree to compute the force, the opening criterion is computed between the inner and outer nodes, and all the particles under the inner node should satisfy it. When moving to a single-tree walk, where we are concerned with individual particles traversing the tree, this node-to-node opening criterion is not optimal: it may lead a particle to traverse a node because the other particles under the same inner node want to traverse that node, even though the particle itself does not. In our implementation, we use a bucket-to-node opening criterion: each particle tests the opening criterion between the bucket it belongs to and the outer node, leading to more precise traversals.

In our measurements, this bucket-to-node opening criterion results in significantly fewer opening criterion calculations than the original looser criterion, as seen in Table 1. Furthermore, while the two open criteria result in similar interaction list sizes, the

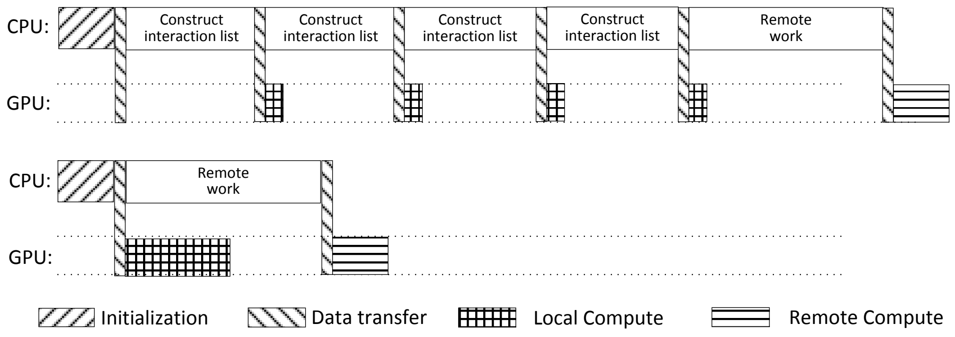
CPU:
Construct interaction list
Construct interaction list
Construct interaction list
Construct interaction list
Remote work
GPU:
CPU:
Remote work
GPU:
Initialization
Data transfer
Local Compute
Remote Compute

Figure 4: Strategy comparison: (1) the upper part is the original ChaNGa. The whole interaction list building process is chopped into pieces to overlap CPU/GPU computation time. When the number of lists reaches a threshold, the CPU sends the built lists to the GPU for computation then resumes the building process. (2) The lower part is the new ChaNGa design. The GPU handles both the tree walk and force computation. (3) The remote gravity computation in both strategies is simplified as a single walk plus a single computation block.

Benchmarks	Simulated original ChaNGa	Single tree walk
lambs	6.16E+09	2.41E+09 (-3.75E+09)
lambb	3.42E+11	1.18E+11 (-2.24E+11)
dwf1	8.75E+09	3.51E+09 (-5.24E+09)
dwf1.6144	1.18E+11	4.46E+10 (-7.31E+10)

Table 1: Number of target tree interactions with original opening criterion vs. single tree condition.

Bench-marks	node-to-node		bucket-to-node	
	cell	particle	cell	particle
lambs	9.55E+08	4.30E+09	1.00E+09 (+4.5E+08)	3.37E+09 (-9.3E+08)
lambb	5.30E+10	1.17E+11	5.33E+10 (+3E+08)	9.61E+10 (-2E+10)
dwf1	1.49E+09	4.52E+09	1.52E+09 (+3E+07)	4.02E+09 (-5E+08)
dwf1.6144	1.98E+10	4.29E+10	1.99E+10 (+1E+08)	4.12E+10 (-1.7E+09)

Table 2: The comparison of the total size of interaction lists for different opening criterion strategies. The "particle" represents the particle-particle interaction list, and the "cell" implies the particle-cell interaction list.

bucket-to-node opening criterion conducts more truncations than the node-to-node open criterion. It requires more particle-cell computation, but reduces particle-particle computation by much more (Table 2). Note that our new implementation achieves the same level of accuracy as the original ChaNGa in standard accuracy tests.

Reducing the latency in tree walk. The gravity computation is complicated and requires massive memory resources. Each tree node contains several pieces of information, such as the mass, position, radius, hexadecapole expansion, etc. Even worse, we need to track more data to enable the local tree walk on the GPU, such as information about child nodes, etc. The compiler has to assign each thread enough registers to handle its work, which can limit GPU occupancy (the number of threads that can be in context simultaneously) and hence lead to poor parallelism. This makes the gravity calculation a latency bound problem.

Note, though, that as the particles walk the tree, not all the data in the tree node is necessary. At most nodes, the particles just check the opening criterion, or compute with the particles under these nodes. Only a few nodes require the remaining information to compute forces. This leads to an optimized tree node data management strategy. In each iteration, the threads only load a subset of data (48 bytes) needed for the opening criterion, instead of the whole tree node (164 bytes). The full hexadecapole expansion is only accessed when the threads need to compute the gravity between particles and the node.

Tree walk pseudocode. Figure 5 shows pseudocode for the GPU tree walk. For the most part, this pseudocode tracks the single-tree Barnes-Hut pseudocode in Figure 1. We note a few key details.

Because GPUs do not manage recursion particularly efficiently, recursion is instead implemented through an explicit stack, `stk`. As we see in line 22 of the pseudocode, traversal is implemented by pushing children nodes onto the stack. This has the added benefit that, because this is a pre-order tree walk, we do not need to spend any time returning back from recursive calls. This captures the *autoropes* approach of Goldfarb et al. [6]. The core of autoropes is to use a stack of dynamically-instantiated pointers to linearize trees.

Line 19 of the pseducode uses the CUDA warp-wide voting mechanism `__any`. This returns true if any thread in the warp returns true. This is used to implement Liu et al.'s optimized *lockstepping* strategy [13], where if any thread in a warp wants to continue traversing the tree, all threads do. Note that it is critical that this decision is made only at the warp level, rather than across all the bodies in a cell. Because of the GPU's SIMT execution model, "carrying along" other bodies in a warp that do not actually want to continue traversal does not incur extra cost; the threads responsible for those bodies would have stayed idle anyway. Making this decision across more bodies than are processed by a warp, however, can cause some warps to do extra work even if none of the bodies in that warp want to visit part of the tree.

6 EVALUATION

This section evaluates the effectiveness of our GPU local tree walk in ChaNGa. We choose ChaNGa as our baseline because its performance has already been well-proven [9]. First, we compare the tree walks' performance with the original ChaNGa GPU approach. We analyze the details of why our implementation wins or loses in performance. Then we investigate scalability.

Platform We evaluate our application on the Comet cluster (at San Diego Supercomputer Center). Its GPU nodes consist of two

```
void gpuLocalTreeWalk() {
  stk = new stack();  // on GPU shared memory
  for(pidx = globalThreadIdx; pidx < numParticles;
    pidx += blockSize * gridSize) {
    init(stk); bucket = loadBucketNode();
    while (!stk.empty()) {
      target = stk.top(); stk.pop();
      if (current_thread_need_to_work()) {
        action = openCriterion(bucket, target);
        cond = isContainedOrIntersect(action);
        if (action == COMPUTE_NODE) {
          if (openSoftening(target, node))
            // evaluate as a MonoPole
          else
            // evaluate as a MultiPole }
        else if (action == COMPUTE_PARTICLES) {
          for (particle : target)
            // evaluate the force for particle }
        if (!__any(cond))
          continue;
        for (child : target.children())
          stk.push(child.Idx) }}}}}
```

Figure 5: GPU single tree walk

Intel Xenon E5-2680v3 processors, 128GB DDR4 DRAM and four NVIDIA P100 GPUs. We are able to get consistent access to 8 nodes in this cluster for our experiments.

Benchmarks We use the original ChaNGa GPU implementation as the baseline, and compare its performance with a variant of ChaNGa that uses our new GPU-only local tree walk. We use the following inputs. ***lambs*** is a 3 million particle representation of the final state (i.e., the current state of the Universe) of a cosmological simulation of a cubical volume 70 megaparsecs in size. ***lambb*** is an 80 million particle representation of that same volume. These simulations were originally used in [19]. ***dwf1*** is a 5 million particle zoom-in simulation. It represents a cosmological volume similar to the above benchmarks, but the particle sampling focuses on a single halo of roughly 1×10^{11} solar masses. ***dwf1.6144*** is a 50 million particle representation of that same halo.

The first two inputs, while clustered on small scales, are roughly uniform on the scale of the entire volume. The second two benchmarks have a non-uniform particle distribution on all scales, i.e., the variance of the number of particles in a volume is large, even for volumes comparable in size to the entire volume.

Performance comparison. ChaNGa is designed for a multi-node and multi-process scenario, and we evaluate the performance under different combinations. We choose a bucket (leaf node) size of 32 and 64 bodies because the GPU version of ChaNGa usually gets the best performance for buckets about that size. We also configure ChaNGa to use the minimal number of treepieces per process (typically one).

The results of running ChaNGa with several configurations are shown in Table 6 (confidence intervals are negligible). We first explore a simple, though unrealistic, configuration of 1 process on 1 node. In this configuration, there is no distribution, and hence all the work is in local tree walks. This maximizes the performance benefit of our new tree walk, as our improvements only target the local tree computation, and we see significant speedups across the inputs—on average 8.25× better.

Name	Total time	Number of calls
Interaction list construction (CPU)	~8s	1
CUDA memcpy HtoD	235.29ms	891
Particle interaction list process (GPU)	180.52ms	144
Cell interaction list process (GPU)	95.16ms	78
CUDA memcpy DtoH	4.59ms	1

Table 3: Runtime breakdown for original ChaNGa

Name	Total time	Number of calls
Interaction list construction (CPU)	0	0
CUDA memcpy HtoD	16.24ms	7
Local tree walk (GPU)	297.11ms	1
CUDA memcpy DtoH	4.54ms	1

Table 4: Runtime breakdown for new ChaNGa

When using 4 processes per node (the second group of results) the story changes. Here, the work is divided into 4 tree pieces. In original ChaNGa, each process handles a different tree piece and sends the interaction lists to its “private” GPU (remember there are 4 GPUs per node). Because the bottleneck in original ChaNGa is the CPU walk, this results in a significant speedup, while adding CPU resources does not help our implementation because it has no need to use these additional CPU resources. However, these resources could be put to use in other computations required by ChaNGa—such as SPH—though we do not explore that in this study. Further, the remote tree walk, which we do not target, starts to consume resources. Hence, the overall speedup of our new configuration over original ChaNGa drops to 2.13×. As we oversubscribe the GPU resource, with 8 processes per node (hence 2 processes per GPU), the GPU becomes the bottleneck, but our implementation is still 1.55× faster.

As we increase the number of nodes, we spend relatively less time on local tree walks, so, as expected, the advantages of our GPU tree walk are reduced. Nevertheless, when we use the maximum number of nodes possible and oversubscribe the CPUs, the GPU local tree walk is still 1.40× faster than the baseline on average. With larger input files, the GPU local tree walk shows better performance.

Performance breakdowns. Next, we break down the amount of time spent in different phases of the computation for the 1-process configuration (allowing us to use `nvprof` for profiling) with the **lambs** input. In original ChaNGa (Table 3), the dominant cost is the CPU building the interaction list: about 8 seconds. However, these interaction lists are large, so sending them to the GPU (`CUDA memcpy HtoD`) and performing the force computations takes 235ms and 276ms, respectively[3]. In contrast, our implementation (Table 4) requires no CPU computation time, and only needs to send the tree to the GPU (taking a mere 16ms). Then the GPU performs a full tree walk, encompassing both building the interaction lists and processing them, in only 297ms. Indeed, performing the *entire walk* on the GPU takes only 10% more time than *just processing the interaction lists* on the GPU in original ChaNGa, and only 25% more time than simply sending the interaction lists to the GPU. By the time the original implementation finishes transferring the interaction lists to the GPU, our implementation would almost be done with the whole computation!

[3]The "CUDA memcpy HtoD" takes 46% of GPU overall runtime (235.29 + 180.52 + 95.16 + 4.59 = 515.56ms) in original ChaNGa.

Tree walk strategies		Original ChaNGa		New ChaNGa				
Bucket Size		32	64	32		64		Average Speedup
		Runtime(s)	Runtime(s)	Runtime(s)	Speedup	Runtime(s)	Speedup	
1 node, 1 process per node	lambs	9.58	5.10	1.06	9.01x	0.85	6.01x	**8.25x**
	lambb	359.67	189.29	31.85	**11.29x**	26.01	7.28x	
	dwf1	16.89	9.16	1.71	9.86x	1.40	6.54x	
	dwf1.6144	194.84	103.93	19.69	9.90x	16.95	6.13x	
1 node, 4 processes per node	lambs	3.08	1.66	1.22	2.53x	0.89	1.88x	2.13x
	lambb	101.22	54.38	29.55	3.43x	23.18	2.35x	
	dwf1	6.26	3.42	3.15	1.99x	1.95	1.76x	
	dwf1.6144	67.52	37.07	40.73	1.66x	25.20	1.47x	
1 node, 8 processes per node	lambs	1.89	1.07	1.05	1.80x	0.77	1.38x	1.55x
	lambb	55.16	30.94	24.07	2.29x	19.83	1.56x	
	dwf1	3.49	1.90	2.40	1.45x	1.55	1.22x	
	dwf1.6144	38.40	20.71	26.75	1.44x	16.32	1.27x	
8 nodes, 1 process per node	lambs	1.92	1.04	1.07	1.80x	0.78	1.33x	1.80x
	lambb	49.49	27.47	15.41	3.21x	10.41	2.64x	
	dwf1	3.51	1.90	2.37	1.48x	1.55	1.22x	
	dwf1.6144	39.10	20.67	27.36	1.43x	16.56	1.25x	
8 nodes, 4 processes per node	lambs	1.50	0.88	0.90	1.67x	0.67	1.31x	1.53x
	lambb	41.11	22.13	16.94	2.43x	13.36	1.66x	
	dwf1	2.27	1.37	1.68	1.35x	1.20	1.14x	
	dwf1.6144	22.93	12.46	14.92	1.54x	10.49	1.19x	
8 nodes, 8 processes per node	lambs	0.80	0.57	0.57	1.39x	0.45	1.27x	**1.40x**
	lambb	21.55	11.70	10.15	2.12x	7.58	1.54x	
	dwf1	1.28	0.82	1.05	1.22x	0.74	1.10x	
	dwf1.6144	11.80	6.50	8.66	1.36x	5.43	1.20x	

Figure 6: Runtime Comparison on Comet (Time in seconds)

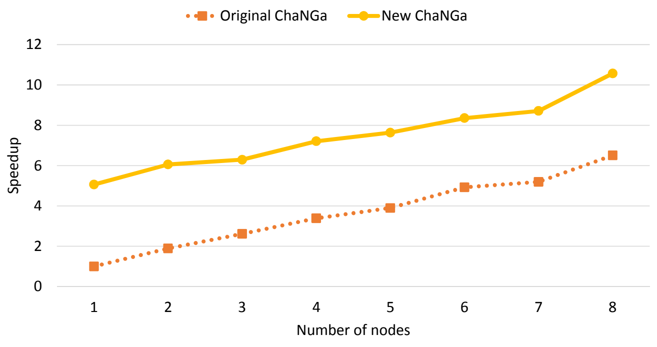

Figure 7: Strong scaling. Base line is original ChaNGa with 1 node.

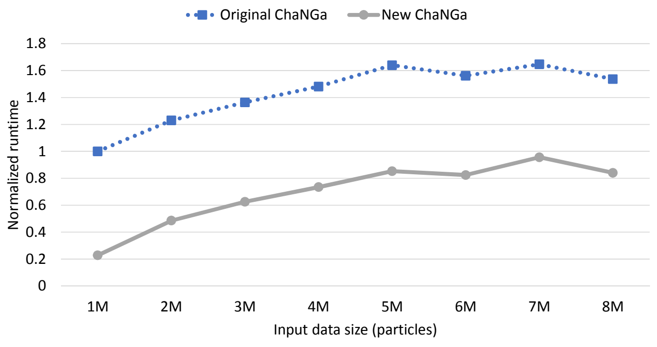

Figure 8: Weak scaling. Base line is original ChaNGa with 1 node.

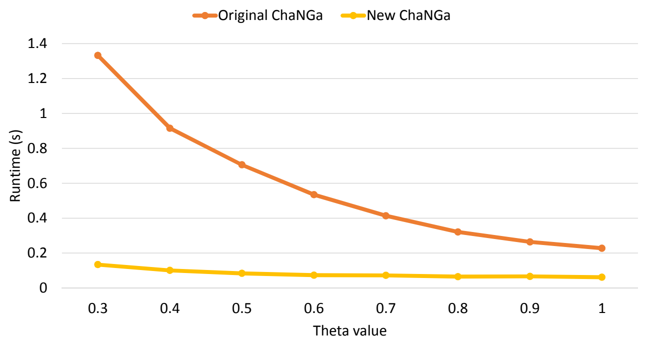

Figure 9: Runtime comparison under different theta values

Scalability. Finally, we evaluate strong and weak scaling, with one process per node. For weak scaling (Figure 8) we use synthetic data with 1M–8M Poisson distributed particles. We do weak scaling with 1M particles per node. For strong scaling (Figure 7), we use the synthetic 8M-particle data. In both cases, our new design scales similarly to the original ChaNGa. This makes sense: we would expect that as we increase the number of nodes, the local tree walk accounts for less of the computation, so the new ChaNGa will eventually scale the same as the original ChaNGa, which has been proven to be scalable.

Runtime comparison over theta. theta is the *opening angle* that determines when to traverse a node in a tree; it is a parameter to the opening criteria. With a smaller theta value, the algorithm opens more nodes, does more computation, and produces more accurate results. Most cosmology simulations set theta in the range 0.6–0.7 [18, 20]. However, we want to study the potential of our approach and evaluate its performance under even higher accuracy requirements. We vary theta with the 1-process configuration and standard theta-test input (30K particles). The dual-tree walk does the tree walk only once for all particles, so we expect it to perform better with smaller theta. However, our evaluation shows that its advantage is overwhelmed by the GPU's strong parallelism. The interaction list construction process on the CPU side becomes a bottleneck in original ChaNGa, and with small theta, our approach vastly outperforms it.

7 RELATED WORK

There has been much research about mapping tree traversals onto GPUs. Gieseke et al. [5] presented a novel data structure called a buffer kd-tree for massive nearest neighbor queries, which relies on a CPU to schedule a GPU's parallel execution of partial tree walks. Similarly, Liu et al.'s [13] work also relies on the CPU to schedule work for GPU execution.

Khatami *et al.* [11] presented a scalable n-body application implemented with HPX that uses futures to reduce synchronization between nodes. Like ChaNGa, SteinBusch *et al.* [24] use a dual-tree walk in their PEPC work, which is a distributed memory parallel BH tree code for electrostatic interactions. Pearce *et al.* [15] proposed a load balance technique that balances interactions rather than particles for distributed Barnes-Hut application. Hegde *et al.* [7] created a novel framework that selects the place to compute the remote force for particles differently. If a particle needs to compute the interaction with a remote subtree, *SPIRIT* sends the particle to the target remote subtree, asking the subtree to do the computation and send back the result. None of above works support GPU acceleration.

Recent work on hybrid CPU/GPU implementations of the Fast Multipole Method (FMM) [3, 12] exploit GPUs in a similar way to ChaNGa's original GPU implementation: an interaction list is built on the CPU, and then this list is processed efficiently on the GPU. FMM makes different tradeoffs than Barnes-Hut: the interaction lists in FMM are based on the structure of the tree and can be built during tree construction rather than requiring additional tree traversal. Moreover, if a tree is incrementally modified when bodies move, these interaction lists can be updated incrementally. In contrast, in Barnes-Hut, at each time step the tree must be traversed anew to reconstruct each body's interaction list, even if the tree can be incrementally rebuilt. As a result, FMM spends relatively less time in interaction list construction than does Barnes-Hut, so our approach of GPU-accelerating interaction list building may be less effective.

8 CONCLUSIONS

ChaNGa is a state-of-the-art distributed computational astrophysics platform that uses a dual-tree variant of Barnes-Hut to efficiently simulate gravitational forces. Unfortunately, ChaNGa's current method for targeting GPUs requires a split computation, where CPUs determine which force computations must happen while GPUs carry out those computations, an approach that underutilizes the GPU and leaves the CPU as a bottleneck. In this paper, we showed that an efficient *single-tree* GPU implementation, though it gives up some asymptotic complexity over the dual-tree approach, can more effectively utilize the GPU and gives consistent performance benefits over the original ChaNGa framework.

9 ACKNOWLEDGEMENTS

This research was supported in part by the NSF (OAC-1550234 and OAC-1550525) and a DOE Early Career Award (DE-SC0010295). This work used the Comet computational resource, which is supported by NSF grant ACI-1341698.

REFERENCES

[1] J. Barnes and P. Hut. 1986. A hierarchical O(N log N) force-calculation algorithm. *nature* 324 (1986), 4.

[2] Martin Burtscher and Keshav Pingali. 2011. An Efficient CUDA Implementation of the Tree-based Barnes Hut n-Body Algorithm. In *GPU Computing Gems Emerald Edition*. Elsevier Inc., 75–92.

[3] Jee Choi, Aparna Chandramowlishwaran, Kamesh Madduri, and Richard Vuduc. 2014. A cpu: Gpu hybrid implementation and model-driven scheduling of the fast multipole method. In *Proceedings of Workshop on General Purpose Processing Using GPUs*. ACM, 64.

[4] W. Dehnen. 2002. A Hierarchical *O*(N) Force Calculation Algorithm. *J. Comput. Phys.* 179 (June 2002), 27–42. https://doi.org/10.1006/jcph.2002.7026 arXiv:astro-ph/0202512

[5] Fabian Gieseke, Justin Heinermann, Cosmin Oancea, and Christian Igel. 2014. Buffer kd trees: processing massive nearest neighbor queries on GPUs. In *International Conference on Machine Learning*. 172–180.

[6] Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. 2013. General Transformations for GPU Execution of Tree Traversals. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 10, 12 pages. https://doi.org/10.1145/2503210.2503223

[7] Nikhil Hegde, Jianqiao Liu, and Milind Kulkarni. 2017. SPIRIT: a framework for creating distributed recursive tree applications. In *Proceedings of the International Conference on Supercomputing*. ACM, 3.

[8] Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V. Kalé, and Thomas R. Quinn. 2010. Scaling Hierarchical N-body Simulations on GPU Clusters. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/SC.2010.49

[9] Laxmikant V Kale and Abhinav Bhatele. 2016. *Parallel science and engineering applications: The Charm++ approach.* CRC Press.

[10] L. V. Kale and Sanjeev Krishnan. 1996. Charm++: Parallel Programming with Message-Driven Objects. In *Parallel Programming using C++*, Gregory V. Wilson and Paul Lu (Eds.). MIT Press, 175–213.

[11] Zahra Khatami, Hartmut Kaiser, Patricia Grubel, Adrian Serio, and J Ramanujam. 2016. A massively parallel distributed n-body application implemented with hpx. In *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. IEEE, 57–64.

[12] Ilya Lashuk, Aparna Chandramowlishwaran, Harper Langston, Tuan-Anh Nguyen, Rahul Sampath, Aashay Shringarpure, Richard Vuduc, Lexing Ying, Denis Zorin, and George Biros. 2009. A massively parallel adaptive fast-multipole method on heterogeneous architectures. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 58.

[13] Jianqiao Liu, Nikhil Hegde, and Milind Kulkarni. 2016. Hybrid CPU-GPU scheduling and execution of tree traversals. In *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2.

[14] Harshitha Menon, Lukasz Wesolowski, Gengbin Zheng, Pritish Jetley, Laxmikant Kale, Thomas Quinn, and Fabio Governato. 2015. Adaptive techniques for clustered n-body cosmological simulations. *Computational Astrophysics and Cosmology* 2, 1 (2015), 1.

[15] Olga Pearce, Todd Gamblin, Bronis R De Supinski, Tom Arsenlis, and Nancy M Amato. 2014. Load balancing n-body simulations with highly non-uniform density. In *Proceedings of the 28th ACM international conference on Supercomputing*. ACM, 113–122.

[16] James C. Phillips, John E. Stone, and Klaus Schulten. 2008. Adapting a Message-driven Parallel Application to GPU-accelerated Clusters. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 8, 9 pages. http://dl.acm.org/citation.cfm?id=1413370.1413379

[17] D. Potter, J. Stadel, and R. Teyssier. 2017. PKDGRAV3: beyond trillion particle cosmological simulations for the next era of galaxy surveys. *Computational Astrophysics and Cosmology* 4, Article 2 (May 2017), 2 pages. https://doi.org/10.1186/s40668-017-0021-1 arXiv:astro-ph.IM/1609.08621

[18] Chris Power, JF Navarro, A Jenkins, CS Frenk, Simon DM White, V Springel, J Stadel, and T Quinn. 2003. The inner structure of ΛCDM haloesâĂŤI. A numerical convergence study. *Monthly Notices of the Royal Astronomical Society* 338, 1 (2003), 14–34.

[19] D. Reed, J. Gardner, T. Quinn, J. Stadel, M. Fardal, G. Lake, and F. Governato. 2003. Evolution of the mass function of dark matter haloes. *MNRAS* 346 (Dec. 2003), 565–572. https://doi.org/10.1046/j.1365-2966.2003.07113.x arXiv:astro-ph/0301270

[20] Darren Reed, Jeffrey Gardner, Thomas Quinn, Joachim Stadel, Mark Fardal, George Lake, and Fabio Governato. 2003. Evolution of the mass function of dark matter haloes. *Monthly Notices of the Royal Astronomical Society* 346, 2 (2003), 565–572.

[21] John K. Salmon and Michael S. Warren. 1994. Skeletons from the Treecode Closet. *J. Comput. Phys.* 111, 1 (1994), 136 – 155. https://doi.org/10.1006/jcph.1994.1050

[22] V. Springel. 2005. The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society* 364 (Dec. 2005), 1105–1134. https://doi.org/10.1111/j.1365-2966.2005.09655.x arXiv:arXiv:astro-ph/0505010

[23] J. G. Stadel. 2001. *Cosmological N-body Simulations and their Analysis*. Ph.D. Dissertation. Department of Astronomy, University of Washington.

[24] Benedikt Steinbusch, Marvin-Lucas Henkel, Mathias Winkel, and Paul Gibbon. 2015. A Massively Parallel Barnes-Hut Tree Code with Dual Tree Traversal.. In *PARCO*. 439–448.

[25] M. S. Warren and J. K. Salmon. 1993. A parallel hashed Oct-Tree N-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing (Supercomputing '93)*. ACM, New York, NY, USA, 12–21. https://doi.org/10.1145/169627.169640