OPTIMIZATIONS FOR N-BODY PROBLEMS ON HETEROGENOUS SYSTEMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Jianqiao Liu

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Milind Kulkarni, Chair

School of Electrical and Computer Engineering

Dr. Anand Raghunathan

School of Electrical and Computer Engineering

Dr. Mithuna S. Thottethodi

School of Electrical and Computer Engineering

Dr. Samuel P. Midkiff

School of Electrical and Computer Engineering

Approved by:

Dr. Pedro Irazoqui

Head of the School Graduate Program

This dissertation is dedicated to my parents, Shucheng and Changqin, and my wife, Zhenyan, who have unconditionally supported me morally and culinarily.

ACKNOWLEDGMENTS

I want to have my utmost gratitude to my fabulous advisor, Milind Kulkarni. He has provided insightful guidance in my research and enlighten me to explore unknown areas. Whenever I got stuck in difficulties, I could find a way out through his conversation and advice. Milind is a consummate communicator, and I'm honored to learn from him turning raw fact into shareable knowledge. I thank my advisory committee, Sam Midkiff, Anand Raghunathan and Mithuna Thottethodi for attending my examinations, and providing valuable feedback to improve my dissertation. And I thank Mithuna and T. N. Vijaykumar for their advisement about computer architecture and optimization during our joint research project.

I want to thank all my co-authors and collaborators. I have sincere gratitude to Tom Quinn for his guidance throughout our three years of collaboration. Tom provided elaborate instruction on ChaNGa and helped me immensely to grow intellectually. I thank Michael Robson, who helped me understand the Charm++ runtime system, and other members in ChaNGa community for incisive suggestions. Throughout my course of the research, I worked closely with Nikhil Hegde, who was also my highest frequency student co-author. Nikhil has pushed me to be a more complete and more thorough thinker.

I have been fortunate to spend three months working with Ian Brown, Ramki Ramakrishna, and Alex Wiltschko during my internship at Twitter. They taught me how to approach new projects in industry and skills in communicating with diverse people. I'd like to thank Hao Lin, my labmate in Purdue ECE and colleague at Facebook. Hao gave me a broadened horizon – there are so much to be learned in the world of industry.

I thank my labmates, Nour Jaber, Kirshanthan Sundararajah, Laith Sakka, Chris Wright, Jad Hbeika and Charitha Saumya. They are extremely talented and motivated researchers with an excellent sense of humor and adorable vigor. Thanks to my roommates, Yiyang.

His thinking and advice, about research and life, has made the tough spots bearable and the high point even better.

This research was supported in part by the DOE Early Career Award (DE-SC0010295) and an NSF funding (OAC-1550525 SI2-SSI: Collaborative Research: ParaTreet: Parallel Software for Spatial Trees in Simulation and Analysis).

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Liu Jianqiao Ph.D., Purdue University, May 2019. Optimizations for N-body Problems on Heterogenous Systems. Major Professor: Milind Kulkarni.

N-body problems, such as simulating the motion of stars in a galaxy and evaluating the spatial statistics through n-point correlation function, are popularly solved. The naive approaches to n-body problems are typically O(n^2) algorithms. Tree codes take advantages of the fact that a group of bodies can be skipped or approximated as a union if their distance is far away from one bodys sight. It reduces the complexity from O(n^2) to O(n*lgn). However, tree codes rely on pointer chasing and have massive branch instructions. These are highly irregular and thus prevent tree codes from being easily parallelized.

GPU offers the promise of massive, power-efficient parallelism. However, exploiting this parallelism requires the code to be carefully structured to deal with the limitations of the SIMT execution model. This dissertation focusses on optimizations for n-body problems on the heterogeneous system. A general inspector-executor based framework is proposed to automatically schedule GPU threads to achieve high performance. Essentially, the framework lets the GPU execute partial of the tree codes and profile threads behaviors, then it assigns the CPU to re-organize these threads to minimize the divergence before executing the remaining portion of the traversals on the GPU. We apply this framework to six tree traversal algorithms, achieving significant speedups over optimized GPU code that does not perform application-specific scheduling. Further, we show that in many cases, our hybrid approach is able to deliver better performance even than GPU code that uses hand tuned, application-specific scheduling.

For large scale input, ChaNGa is the best-of-breed n-body platform. It uses an asymptotically-efficient tree traversal strategy known as a dual-tree walk to quickly provide an accurate simulation result. On GPUs, ChaNGa uses a hybrid strategy where the CPU per-

forms the tree walk to determine which bodies interact while the GPU performs the force computation. In this dissertation, 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 one node, one process per node configuration. We also point out that ChaNGa's implementation doesn't satisfy the inclusion condition so that GPU-centric remote tree walk doesn't perform well.

1. INTRODUCTION

1.1 N-body problems

N-body problems consist of a class of important problems across computational geometry, statistics, computational physics, and machine learning fields. From simulating the formation of galaxies to recommending customers potential product according to their social network neighbors, n-body problems are popularly solved in the various fields.

The n-body problems share a common structure: each body in a large data set potentially needs to traverse all the rest bodies (one or more times) to update some attributes. One example is the *n-body simulation*, 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 modeling the self-gravity of astronomical bodies—stars in a galaxy, for instance. As withall 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, or even all the n-body problems: a *tree code* [1]. This approach takes advantage of the fact that the gravitational force of a group of bodies can be approximated by a multipole exansion [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

[1]The multipole expansion is the sum of spherical harmonics of the mass distribution in the Cartesian coordinates.

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. Most of the n-body problems are being solved along with the same strategy: using a recursive tree traversal code to reduce the computation complexity.

There are more n-body problems besides the n-body simulation. They consist but not limited to:

The n-point correlation function forms the theoretical foundation of several computational fields, e.g. bioinformatics and data mining, which are heavily using statistical analysis or simulation. Nowadays, it is being used by the cosmologists to validate the state-of-the-art cosmological model with the observation from the weak lensing and cosmic microwave background survey. These surveys image hundreds of millions of galaxies, which necessitates efficient methods to extract useful data. The point correlation functions supply fast and accurately spatial evaluation to them. A key question that point correlation functions are typically used to answer is that: for a given galaxy, the correlation functions describe the probability that another galaxy can be found within a certain distance [2].

The k-nearest neighbors (kNN) algorithm is mostly known as a non-parametric method in regression and classification. It doesn't make any assumption to the input data, which is quite useful to process the practical data who does not clearly obey the typical theoretical assumptions like the Gaussian mixtures or linearly separable etc. Similar to the Barnes-Hut algorithm, the kNN also adopts the tree structure to partition the space and accelerate the search process. Each point in the query data set traverses the reference tree to update a list (of length k) of its nearest neighbors. While most of the time the query and reference data set are the same, they may be quite different in application field like the computational geometry. The type of tree structure has a strong effect on search performance [3].

Smooth Particle Hydrodynamics (SPH) is a central technique in simulating the mechanics of fluid flows. It was first proposed in 1977 as an alternative way to simple grid-based approach to fluid dynamics. SPH is a particle-based method and thus usually implemented as tree code. It computes pressure from the weighted contribution of nearest neighboring

particles rather than solving linear systems of equations. The simulation result could be adjusted through variables like density/pressure. The abilities to provide adaptive resolution and simulate phenomena across many orders of magnitude make SPH ideal for astronomical simulations, such as galaxy formation and the structure of dark matter.

1.2 N-body problems on GPU platform

GPUs offer the promise of massive, energy-efficient parallelism, providing hardware that can execute hundreds of simultaneous parallel threads. As a result, the last decade has seen intense efforts towards mapping applications and kernels to GPUs to take advantage of that parallelism. Unfortunately, achieving such highly efficient parallelism requires compromises: GPUs present a somewhat limited SIMT (single-instruction, multiple-thread) execution model, where, to take full advantage of the parallel execution resources, threads that are executing simultaneously must perform the same computation (avoiding *control divergence*) and access memory in a predictable way (avoiding *memory divergence*). In other words, while GPUs appear to be well-suited to executing data-parallel algorithms, the full power of the GPU cannot be exploited unless the data-parallel tasks are similar to each other. As a result, most successful GPU algorithms are *regular*, with predictable control flow and memory access patterns.

There has been considerably less success tackling n-body problems on GPU. These algorithms, which feature data structures such as trees and data-dependent behavior, are much more difficult to map to GPUs, as the input-dependence precludes grouping together threads to minimize control divergence and the pointer-based data structures mean that even if threads are performing similar operations, their memory accesses are likely not predictable, increasing memory divergence. As a result of these difficulties, most attempts to map n-body problems to GPUs have been one-off implementations: for each new algorithm, a new, *ad hoc* implementation for GPUs must be developed [4–11].

Recently, Goldfarb *et al.* developed a general framework for mapping a class of n-body problems, to GPUs [12]. These applications, which include classic algorithms such

as Barnes-Hut [1] and kd-tree–based nearest neighbor searches [13], perform multiple recursive traversals of tree structures. The multiple traversals expose tremendous amounts of data parallelism. However, since the behavior of each traversal is input dependent, the data parallel tasks are not identical, making GPU execution challenging. If the tasks are merely mapped to SIMT threads, each thread ultimately experiences substantial memory and control divergence. To overcome this, Goldfarb *et al.* developed two transformations, *autoroping* and *lockstepping*, that restructure these tree traversal algorithms so that they execute effectively on GPUs (see Section 2.3 for more details).

Unfortunately, to achieve maximum performance using their framework, Goldfarb *et al.* used application-specific "sorting" optimizations. To minimize control divergence while avoiding memory divergence, Goldfarb *et al.* reorganized the set of traversals to be performed so that the traversals that were grouped into SIMT thread groups were likely to touch similar portions of the tree. Figuring out a fast, effective way to perform this sorting requires a careful understanding of an algorithm's behavior. In other words, this prior work relies on application-specific *sorting* to achieve high performance. What is missing is an approach to tackling these problems that does not rely on application-specific knowledge to be effective.

1.3 N-body problems on distributed system

N-body problems can involve millions or even trillions of bodies, distribution is a key approach to scaling up computation. This thesis focus on the n-body simulations for the distributed platform. Over the years, there have been many frameworks developed to perform distributed tree code–based *n*-body simulations [14–16]. One of the most advanced, and most efficient, is *ChaNGa* [17]. The full name of ChaNGa is "Charm Nbody GrAvity solver". This framework, based on Charm++ [18], 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 the remote tree pieces).

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 tree walk performs a number of "cell-cell" interactions, which 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[2]. 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 this 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 force computation. Because the interaction lists are dense, regular structures, the force computation step can be efficiently performed on the GPU.

[2]A variant of this approach is also used in another classic n-body algorithm, the Fast Multipole Method [19, 20].

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.

1.4 Our approaches

To make effective optimization for n-body problems on heterogeneous systems, we spend efforts on two main directions:

1.4.1 Hybrid CPU-GPU scheduling and execution

For a single computer, we take advantage of a key insight about the behavior of tree traversal algorithms that allows us to perform effective scheduling without performing application-specific sorting: even though traversals are data-dependent and hence inherently unpredictable, if the behavior of a single traversal is examined part of the way through its execution, its future behavior is highly correlated with its past behavior. In other words, two traversals that have behaved similarly for the first half of their execution are likely to behave similarly for the second half, as well.

The insight that past traversal behavior is correlated with future behavior has been exploited before, in narrower, or application-specific contexts by Zhang *et al.* [21] and Pingali *et al.* [22], and in a more general tree-traversal context by Jo and Kulkarni [23, 24]. Fundamentally, these approaches all interleave the scheduling component (tracking past behavior and reorganizing computations based on that behavior) with the execution (perform the next phase of computations). This tight coupling has an advantage: by interleaving tracking and scheduling with execution, the schedule can be continuously adapted in response to the profiling information. However, these approaches also have a serious disadvantage: because the tracking and scheduling occur continuously, and are highly irregular processes, this tightly-coupled approach is ill-suited to execution on a GPU.

In this work, we introduce a completely different way of automatically scheduling tree traversals. We base our approach on an extension of the prior insight about traversal behavior. The depth-first nature of recursive traversals means that the behavior of traversal as it explores the "lower half" of the tree (*i.e.*, nodes at depth more than half the tree height) is largely determined by its behavior in the "upper half" of the tree (*i.e.*, which nodes in the upper half of the tree the traversal visits, and in which order). In other words, two traversals that have similar behaviors in the upper half of the tree will behave similarly in the lower half of the tree, as well. However, the vast majority of the *work* performed by the traversal occurs in the lower half of the tree. As a result, it is possible to examine the behavior of traversals as they visit the upper half of the tree, and use *just that information* to reschedule the traversals as they execute the lower half of the tree.

Crucially, since this upper-half execution is a small fraction of the overall execution time, it is not burdensome to perform that execution more than once. This fact suggests a *automatic*, *hybrid*, *insepctor-executor* approach that can be readily mapped to a GPU. We perform the "top half" of all of the traversals on the GPU once, to collect profiling data about the behavior of each traversal. This *inspector* stage is highly data-parallel, with a small memory footprint, and is well-suited to the GPU. The GPU then transmits that profiling data back to the CPU, which performs the highly irregular scheduling process to determine a *new* schedule of execution. This new schedule of execution is then used to *execute* the original tree traversal algorithm on the GPU, using strategies such as Goldfarb *et al.*'s. Even though the inspector and scheduler phases add additional overhead to the application, the gains from the optimized schedule during the execution phase win out.

1.4.2 GPU single tree walk

For n-body problems on a distributed system, we make the observation 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:

a desire for massive parallelism and a desire for regular control flow. Hence, in this dissertation, 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. [12] and Liu et al. [25] that show that it is possible to implement these tree walks in a way that nevertheless limits its control divergence and hence 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, that on a CPU-heavy system our implementation is 4.85× faster than the best prior configuration of ChaNGa in the best case, and 3.66× faster on average, and on a GPU-heavy system, our implementation is 8.07× faster in the best case and 6.04× faster on average.[3]

1.5 Contributions and organization

The dissertation starts with the hybrid CPU-GPU scheduling and execution for n-body problems on a single machine, then we make an insightful observation to the state-of-the-art n-body simulation implementation and contribute our optimization techniques to it. The primary contributions are:

1. We introduce a hybrid, inspector executor–based, dynamic scheduling algorithm that performs partial traversals on the GPU, then reschedules the traversals on the CPU, before completing the work on the GPU. We develop optimized versions of this scheduling algorithm that exploit structural properties of traversal algorithms to further improve our dynamic scheduling.

[3] On a single node with 1 CPU/GPU, we compute the average speedup over different input datasets and bucket sizes.

2. We implement a framework that performs this dynamic scheduling in a general, application-agnostic manner, allowing programmers to produce hybrid CPU-GPU implementations of tree traversal algorithms.

3. We develop a new skeleton for writing the GPU kernel portion of tree traversals that minimizes unnecessary memory accesses.

4. We observe that dual tree walk is not suitable for GPUs and point out that its asymptotic behaviors are counterbalanced by the specific requirements of efficient GPU computation.

5. We implement GPU gravitational force computation kernel using above skeleton and integrate it with the ChaNGa, which is the state-of-the-art distributed computational astrophysics platform.

6. We point out that ChaNGa's implementation doesn't satisfy the inclusion condition so that GPU-centric remote tree walk doesn't work well.

2. N-BODY PROBLEMS ON GPU

This chapter describes the necessary background for the remainder of the dissertation. We briefly cover the n-body code, the SIMT execution model of GPUs and Goldfarb et al.'s approach [12] for mapping traversal algorithms to GPUs.

2.1 n-body codes

The naïve approach to perform the 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 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 represented by the node's spatial property. Otherwise, the particle needs to traverse each child of the node. The process is repeated until no more nodes remain (Figure 2.1).

2.2 GPU execution model

GPUs use a SIMT (single-instruction, multiple-thread) execution model that allows multiple threads to execute efficiently in parallel. SIMT execution is, essentially, vector execution: multiple threads can execute in parallel provided that all the threads are performing *the same instruction*. In the simplest case, consider a group of threads that each perform exactly the same operation on different pieces of data in an array. In a SIMT ex-

```
1  void BarnesHut(particle, node) {
2    if (far_away(particle, node)) {
3      calculateGravity(particle, node);
4    } else if (isBucket(node)) {
5      for (p : node.particles())
6        calculateGravity(particle, p);
7    } else {
8      for (child : node.children())
9        BarnesHut(particle, child);
10   }
11 }
```

Fig. 2.1.: Barnes Hut pseudocode

ecution, some number of threads will be combined into a single group (called a "warp" in NVIDIA parlance, and a "wavefront" by AMD; for brevity, we will use the term "warp" hereafter). These threads will execute in lockstep, each executing the same instruction simultaneously. As long as all the threads perform the same instruction, and all memory accesses performed by the threads are well-structured (*e.g.*, adjacent locations in an array), the GPU will deliver large amounts of efficient parallelism.

The key to the SIMT execution model, which both lends it its ease of use and hides a series of performance pitfalls, is how it deals with situations when threads *do not* perform exactly the same instruction (*control divergence*), or do not access memory in well-structured ways (*memory divergence*). In this case, the GPU hardware automatically manages execution by masking out threads in the warp so that the threads that *do* execute still execute in a straightforward, vectorized manner. So, for example, if the threads in a warp encounter a branch, with some taking the branch and the others falling through, the warp will conceptually split into two groups, with the taken threads executing together while the fall-through threads are stalled, then vice-versa, until all the threads reconverge after the branch. This masking behavior under *control divergence* reduces parallelism, often significantly. Similarly, if the threads in a warp do not access memory locations in a structured way—which allows the hardware to coalesce the memory accesses into a single operation—some of the threads will stall until all the accesses can be completed. This *memory divergence* also re-

duces parallelism. Fundamentally, achieving good performance under the SIMT execution model requires structuring code to minimize control and memory divergence.

In the presence of divergence, GPU utilization can drop precipitously, at which point the parallelism advantages of a GPU are moot: execution on a CPU can often be faster. Unfortunately, irregular applications often incur both types of divergence. Because of data-dependent behavior, threads often suffer from control divergence. Because of the dynamic memory allocation inherent in pointer-based data structures, threads often access unpredictable memory locations on loads, leading to memory divergence. As a result, most attempts to map irregular applications to GPUs have required very careful, application-specific tricks and techniques to achieve good performance.

2.3 Autorope and lockstep traversal

Goldfarb *et al.* described an approach for mapping a general class of irregular applications—those that perform repeated tree traversals—to GPUs [12]. These applications are characterized by the following structure: a set of "points" each traverse a single tree in a recursive, depth-first manner. However, GPU implementations of recursive tree traversals suffer from a specific performance pitfall. After a thread finishes traversing along a particular path in the tree, it must return to upper nodes in the tree to reach other branches. Thus the interior nodes are repeatedly traversed, an overhead that is compounded by the expense of numerous recursive calls on a GPU. To mitigate this overhead, Goldfarb *et al.* proposed a transformation called *autoropes* [12].

Ropes are a common technique for mapping traversal algorithms to GPUs. Rather than letting threads discover which nodes to visit through a series of recursive calls, ropes are additional pointers installed in the tree that directly point to the next node to be traversed (*e.g.*, to a sibling node in the tree), avoiding the expense of revisiting interior nodes. Ropes provide a linearization of the tree. Unfortunately, the particular targets of rope pointers are application specific and are complicated when multiple traversal orders are possible. Autoropes is an application-agnostic transformation that uses a stack of dynamically-

instantiated rope pointers to linearize trees. When visiting a node using autoropes, the thread pushes pointers to the children nodes onto a *rope stack* in the reverse order they will be traversed. Then, instead of making recursive calls, the autorope traversal just iterates over the rope stack, eliding the overhead of recursion, and ensuring that each node is visited just once.

Autoropes replaces the recursive call stack with a simple iteration over the rope stack. As a result, threads experience significantly less control divergence (because they are all simply iterating over a stack). Unfortunately, this means threads in a warp can diverge in the tree, with different threads touching very different portions of the tree, resulting in unnecessary memory traffic. *Lockstepping* mitigates this problem by introducing additional control flow that keeps threads in sync in the tree during traversal. When a thread is truncated at certain node $\mathfrak{n}$, it doesn't move to the next node through autorope stack directly. Instead, if other threads in the warp want to continue the traversal to the subtree rooted at node $\mathfrak{n}$, the thread will be carried along by others, masked out from any computation. A warp only truncates its traversal when all its threads in the warp have given up the traversal. To ensure that lockstepping does not result in many threads being dragged along through the tree doing no useful work, it is important to carefully schedule the computation so that threads with similar traversals get grouped together into a warp. Together with autoropes, lockstep traversal delivers high performance for well-scheduled inputs [12].

3. HYBRID CPU-GPU SCHEDULING AND EXECUTION

As mentioned in the introduction, scheduling is an important issue for tree traversals because GPU's SIMT execution model requires structuring code to achieve high performance. The threads in the same warp have to perform exactly the same instruction and access the memory location in a predictable way. Otherwise, the control divergence (threads *do not* perform exactly the same instruction) and memory divergence (threads *do not* access memory in well-structured ways) pop up and steal over the GPU utilization. Unfortunately, n-body codes often incur both types of divergence. Because of data-dependent behavior, threads often suffer from control divergence. And because of the dynamic memory allocation inherent in pointer-based data structures, threads often access unpredictable memory locations on loads, leading to memory divergence. Thus, mapping n-body codes to GPUs requires very careful scheduling techniques. However, due to the high input-dependency and complex traversal patterns, the scheduling for n-body codes is extremely hard.

In this chapter, we demonstrate a hybrid, inspector-executor based dynamic scheduling framework that allows programmers to efficiently implement tree traversal algorithms on CPU-GPU platform. We begin with the formulation of the scheduling problem and its NP-hardness provement. Then we explain our design in detail, including optimization for different traversal patterns. We also include some key points in our implementation and a full evaluation.

3.1 Traversal Scheduling

As explained in the introduction, efficiently mapping tree traversals on GPUs requires carefully *scheduling* those traversals so that traversals that are grouped together into the same warp are as similar as possible. This ensures that *lockstep* traversal is able to exploit substantial commonality in the memory accesses performed by traversals while not overly

expanding the amount of work done by a warp. This section shows that the general scheduling problem, Sched is NP-hard, necessitating the use of heuristics. It then summarizes prior scheduling and sorting heuristics for tree traversal algorithms.

3.1.1 Sched is NP-hard

The general scheduling problem for tree traversals, which we call Sched, is simple to define. Given a point p that represents a traversal, define $t(p)$ as the set of nodes visited during p's traversal of the tree. For two points, p_i and p_j, define the *difference* between the traversals, $\delta(p_i, p_j)$ as $t(p_i) \cup t(p_j) - (t(p_i) \cap t(p_j))$—in other words, the nodes that exist in one traversal but not in the other. Note that these are the nodes that result in non-convergent computation, as only one point needs to visit them.

Sched is the following problem. Given a set of points, $\{p_1, ..., p_n\}$, produce a sequence s of those points that minimizes:

$$\Delta = \sum_{i=1}^{n-1} |\delta(s_i, s_{i+1})|$$

In other words, Sched minimizes the total differences between consecutive points in the sequence—it produces a *sorted* sequence.

Theorem 1 Sched *is NP-hard.*

Proof To show that Sched is NP-hard, we reduce from Hamiltonian Path: given an undirected graph $G = (V, E)$, find a path that visits each vertex once. We show how to design a tree traversal problem based on G where solving Sched for that problem solves the Hamiltonian Path problem.

First, build a tree with $|E| + 2$ leaves. Let the first leaf in the tree be x and the last leaf in the tree be y. Label each of the other $|E|$ leaves in the tree according to the edges in E; call these *edge-based* leaf nodes. Then, attach subtrees with $4|E|$ nodes to x and y. For each vertex $v_i \in V$, we specify a point p_i with a traversal defined as follows: p_i visits all of the nodes in the tree *except* the subtrees rooted at x and y and any edge-based leaf node

```
1 void recurse(node root, point pt) {
2   if (!can_correlate(root, pt))
3     return;
4   if (is_leaf(root))
5     update_correlation(root, pt);
6   else {
7     recurse(root.left, pt);
8     recurse(root.right, pt);
9   }
10 }
```

Fig. 3.1.: Point correlation

corresponding to an edge *not* incident on v. The only leaf nodes visited by p_i correspond to the edges incident on v_i, $E(v_i)$. Then define two additional points p_x and p_y, which truncate at x and y, respectively, and otherwise visit all of the other nodes in the tree except the edge-based leaf nodes.

Note that for any two vertices v_i and v_j, $|\delta(p_i, p_j)| = |E(v_i)| + |E(v_j)| - 2|E(v_i) \cap E(v_j)|$. Note that the last term is zero unless v_i and v_j share an edge. Also, for all vertices v_i, $|\delta(p_x, p_i)| = |\delta(p_y, p_i)| = 4|E| + |E(v_i)|$.

Now we solve Sched across all the points—those corresponding to the vertices of G as well as p_x and p_y. Note, first, that minimizing Δ requires that p_x and p_y be scheduled first or last—otherwise their $4|E|$ difference penalty is accounted for twice. Each other point appears in two pairs in the scheduled sequence. Every edge therefore is accounted for four times—twice for each vertex it is incident on—unless it is incident on both vertices of a pair. We thus have that Δ is:

$$8|E| + 4|E| - 2Q$$

Where Q is the number of point pairs in the sequence produced by Sched that share a leaf node—in other words, the number of vertex pairs that share an edge. Δ is minimized when Q is maximized. In other words, Δ is minimized when all vertex pairs in the sequence share an edge—a Hamiltonian Path. Hence, if a Hamiltonian Path exists, Sched will find it. ■

Note that SCHED merely refers to an arbitrary set of tree traversals. However, we are not interested in arbitrary tree traversals—we are interested in tree traversals that are generated from recursive tree traversal algorithms, as Point Correlation algorithm in Figure 3.1. It is straightforward to construct such an algorithm for a given graph. When building the tree for the graph, color each of the nodes that should be visited by *all* of the points white, all of the nodes visited only by p_x red, all of the nodes visited only by p_y green, and all of the other nodes blue. It is clear that the `can_correlate` predicate in Figure 3.1 can be modified to ensure that the points visit exactly the nodes they are supposed to: if a node is white, then `can_correlate` is always true; if a node is red or green, `can_correlate` is true only if the point is p_x or p_y, respectively; if a node is blue, `can_correlate` looks up whether the graph vertex associated with the point is incident on the edge associated with the node. This modified recursive algorithm, when presented with a set of points derived from the vertices of the graph in question, and a tree built as specified above, produces exactly the set of traversals needed for SCHED to find a Hamiltonian path if one exists.

3.1.2 Prior sorting heuristics

Because SCHED is NP-hard, we must instead turn to heuristics to schedule traversals. The typical approach for tree-traversal applications is to use *ad hoc*, application-specific sorting heuristics, based on a programmer's understanding of the behavior of the tree-traversal algorithm. As a result, there have been several strategies proposed for specific traversal algorithms. For Barnes-Hut alone, researchers have proposed sorting using space-filling curves [26], Z-curves [27], orthogonal bisection [28], or the structure of the Barnes-Hut tree itself [29]. For ray tracers, researchers have suggested various ray-reorganization techniques [30–34]

Rather than devising new sorting strategies for each new traversal algorithm, several researchers have looked at using the past behavior of computations to predict their future tree accesses, and hence dynamically schedule them with minimal application-specific knowledge [21–24]. Most directly relevant, as they target the same types of algorithms as this

paper, is Jo and Kulkarni's *traversal splicing* work [23, 24]. Traversal splicing operates by tracking each traversal's behavior during execution. Each traversal is partially executed until it either truncates its execution at a node in the tree, or reaches some pre-specified maximum depth in the tree. Traversals that are truncated at the same node in the tree (including those that make it to the pre-specified maximum depth) are considered similar, and a new execution order is constructed based on this information. Then all of the traversals are again partially executed until they truncate again or reach another pre-determined stopping point, and the process repeats.

Traversal splicing is based on the insight that traversals that truncate at the same part of the tree are behaving similarly, and hence are likely to behave similarly in the future. Unfortunately, traversal splicing requires very careful bookkeeping, monitoring of traversals, sorting, and interleaving the execution of the traversals with the highly irregular scheduling process. As a result, traversal splicing incurs noticeable runtime overhead [23][1], and is very *ill-suited* to execution on GPUs.

Hence, we are left with a dilemma: known scheduling approaches are either application-specific, or require highly-irregular computation that is poorly matched to GPUs' SIMT execution model. In the next section, we present a novel hybrid scheduling approach that splits the tasks of scheduling and execution, and hence is substantially simpler than prior dynamic scheduling approaches, incurs less runtime overhead, and is well-suited to mapping to GPUs.

3.2 Design

This section describes our hybrid CPU-GPU scheduling strategy, a novel scheduling and execution technique for tree traversal algorithms that is both general and automatic. We do not rely on any application-specific or semantic knowledge. Instead, our technique uses two GPU kernels: one that runs a portion of the traversal code on the GPU while inspecting the behavior of individual traversals. The CPU then uses this information to dynamically

[1]Though this overhead is often mitigated by gains in locality.

reorder the traversals so that when the second kernel is called, threads grouped into warps perform similar work, improving SIMT efficiency. This strategy is, essentially, an instance of the inspector-executor model [35], where the initial GPU kernel acts as the inspector, the CPU is used to perform the re-scheduling, and the second GPU kernel acts as the executor. The key phases of the technique are:

1. *Profiling.* A carefully constructed GPU kernel runs a small portion of every traversal in the algorithm to collect behavioral information that is used during scheduling. (Section 3.2.1).

2. *Scheduling.* The CPU analyzes the profiling information and groups threads into different buckets. Threads in one bucket are more likely to access the same branches of the tree and hence exhibit better locality. (Section 3.2.2).

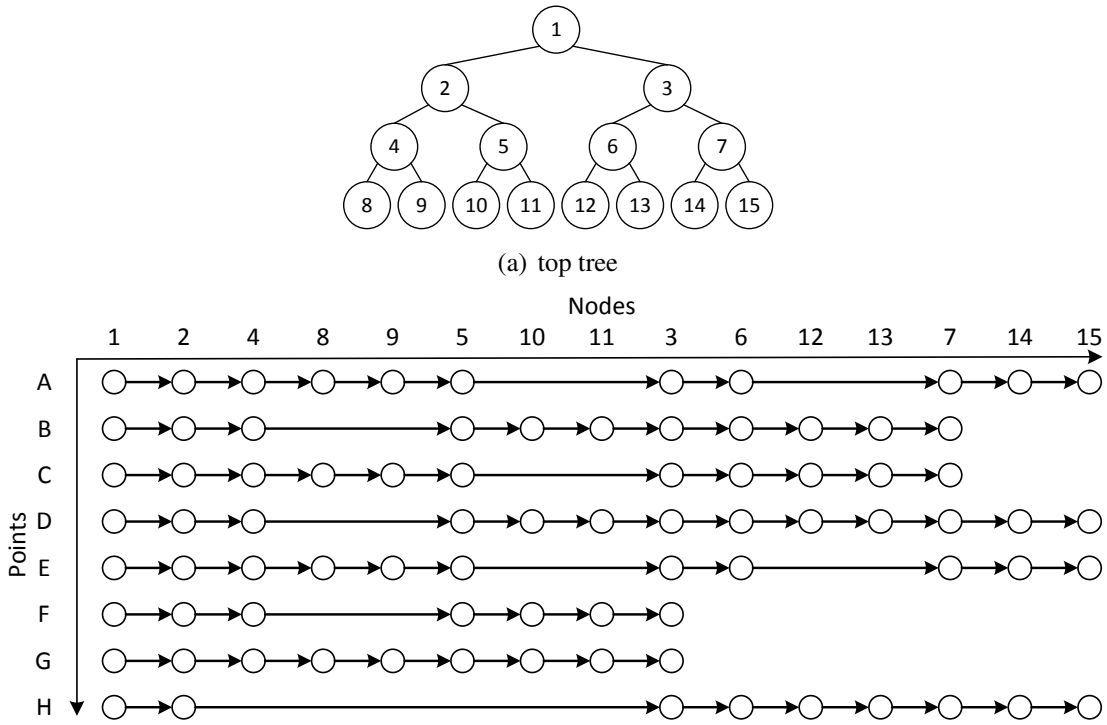
3. *Execution.* This schedule is then used to execute a second GPU pass that performs the rest of the traversal, using an optimized kernel (Section 3.2.3).

In Section 3.2.4, we argue that this scheduling strategy is sound.

3.2.1 Profiling

In the profiling stage, we run a GPU kernel that performs each traversal *only in the top half of the tree*. Because the top portion of the tree is small relative to the rest of the tree, this profiling step accounts for a very small proportion of the overall computation, and hence even if the points are poorly scheduled, the overall impact on performance is small.

Figure 3.2(a) shows the top portion for a binary tree, with nodes indexed in heap order. In the following sections, we call this top portion the *top-tree*. Figure 3.2(b) shows the traversals of eight points using the algorithm shown in Figure 1. The vertical axis shows different input points that would traverse the tree, while the horizontal axis records which nodes a point may visit. Each circle in the diagram represents a computation step during execution. Note that each point does not visit all the nodes.



(a) top tree

(b) point traversal

Fig. 3.2.: Tree traversal algorithm

```
1 void profiling(node root, point pt) {
2   stack stk = new stack();
3   stk.push(root);
4   while (!stk.is_empty()) {
5     root = stk.pop();
6     if (!can_correlate(root, pt))
7       continue;
8     // update information here
9     if (is_leaf(root)) {
10      matrix[pt.id][root.id] = root.id;
11    } else {
12      stk.push(root.right);
13      stk.push(root.left);
14    }
15  }
16 }
```

Fig. 3.3.: Profiling for point correlation

During profiling, the traversal of each thread is traced and recorded into a *scheduling matrix*. The scheduling matrix has one row for each point, and one column for each node

Table 3.1.: Scheduling matrix for point correlation

	1	2	4	8	9	5	10	11	3	6	12	13	7	14	15
A	1	2	4	8	9				3	6			7	14	15
B	1	2	4			5	10	11	3	6	12	13	7		
C	1	2	4	8	9				3	6	12	13	7		
D	1	2	4			5	10	11	3	6	12	13	7	14	15
E	1	2	4	8	9				3	6			7	14	15
F	1	2	4			5	10	11	3						
G	1	2	4	8	9	5	10	11	3						
H	1	2							3	6	12	13	7	14	15
Result				**ACEG**			**BDFG**				**BCDH**			**ADEH**	

in the top tree. When a point visits a node, the appropriate cell in the table is marked. Figure 3.3 shows how the point correlation code is augmented with this profiling data. Note that the stack manipulation in lieu of recursive calls to visit different portions of the tree is due to the autoropes transformation (Section 2.3). Figure 3.1 shows the resulting scheduling matrix for the set of traversals in Figure 3.2(b). Note, we only need the leaf node columns, which are marked in gray, for scheduling. This matrix is transferred back to the CPU for use during scheduling, as described in the next section. The profiling overhead is discussed in Section 3.4.

Guided traversals In some algorithms, such as nearest neighbor, the particular order a point visits nodes is governed by point-specific data. For example, based on characteristics of a point, one point might visit the tree root's left child before its right, while another might visit the right child before the left. Following Goldfarb *et al.*'s terminology, we call algorithms that have this property *guided* traversals, in contrast to algorithms like point correlation that are *unguided* [12]. Note that whether a traversal is guided or not can be determined by a simple static analysis that determines whether the order of recursive calls is control dependent on any point-specific data.

Because the traversal order of each point in a guided traversal is different than in an unguided traversal, our profiling code must also encode that traversal order. Figure 3.4 shows that code. Note, first, that the particular order of tree traversal is determined by the predicate `closer_to_left` (line 13), making the traversal guided. Second, the structure of the scheduling matrix is now different. Rather than each column representing a particular

```
1 void profiling(node root, point pt) {
2   stack stk = new stack();
3   stk.push(root);
4   int index = 0;
5   while (!stk.is_empty()) {
6     root = stk.pop();
7     if (!can_correlate(root, pt))
8       continue;
9     // update information here
10    if (is_leaf(root)) {
11      matrix[pt.id][index++] = root.id;
12    } else {
13      if (closer_to_left(root, pt)) {
14        stk.push(root.right);
15        stk.push(root.left);
16      } else {
17        stk.push(root.left);
18        stk.push(root.right);
19      }
20    }
21  }
22 }
```

Fig. 3.4.: Profiling for nearest neighbor

node in the tree, column *i* represents the *i*th node visited by a particular point. Line 11 shows how the particular traversal order of a given point is encoded into the matrix.

Profiling overhead The profiling step is essentially partial execution of the whole computation task but truncated at certain depth. The deeper this truncation depth, the more information is collected during profiling. However, as we explain in Section 3.2.3, the computation performed during profiling is re-computed during execution, and hence doing more work during profiling can result in wasted work. Balancing the effects of more profiling information with more profiling overhead is a classic problem in any profile-guided optimization. In our case, we choose a profiling depth of around one third of the tree. Because the bottom half of the tree contains the bulk of the tree nodes, our profiling overheads are very low. Moreover, we also adopts CUDA streams to overlap the profiling execution and data transmission. Especially, when the schedule matrix is oversized, splitting the input and multi-streaming can perfectly achieve good performance.

3.2.2 Scheduling

Scheduling unguided traversals

The profiling matrices we generate in the profiling step provide information that lets us reason about the behavior of the points. In Table 3.1, we see that points **ACEG** all visit the leaf nodes ⑧ and ⑨ of the top tree. Since all of these points reached the same leaf nodes of the top tree, it is more likely that they will behave similarly as they traverse the rest of the tree. Similarly points **BDFG** all visit nodes ⑩ and ⑪, so we expect them to behave similarly in the rest of the tree. (Note that point **G** shows up in both groups; we conclude that it behaves somewhat similarly to both groups of points).

We can scan the *columns* of the scheduling matrix to construct scheduling buckets. The last row of Table 3.1 shows the resulting buckets. Note that even though the top tree has eight leaf nodes, there are only four buckets. This is because sibling leaf nodes have the same information in our example.

These buckets represent points that have some similarity of behavior. Scheduling according to this information can greatly improve locality. However, points in the same bucket are still unoptimized. Since each bucket may contains millions of points, the divergence in a single bucket can still be considerable. Indeed, intra-bucket scheduling can be even more important than inter-bucket grouping. We hence perform intra-bucket scheduling while building each bucket.

At a high level, the idea behind intra-bucket scheduling is simple. In the result row of Table 3.1, **A** and **E** appear in two buckets together, while they only appear in one bucket with **D** and **H**. Hence, in the bucket where all four points appear, ⑭, we would like to execute **A** and **E** consecutively, and then **D** and **H**. In other words, nodes that appear in several buckets together should be considered more similar, and hence scheduled together. Unfortunately, building the buckets and then searching for such similarity is very expensive. We thus use an intra-bucket scheduling algorithm that orders the points on the fly.

Intra-bucket scheduling Rather than treating the construction of each bucket as a separate process, we consider these steps a continuous process: we use the outcome of building one

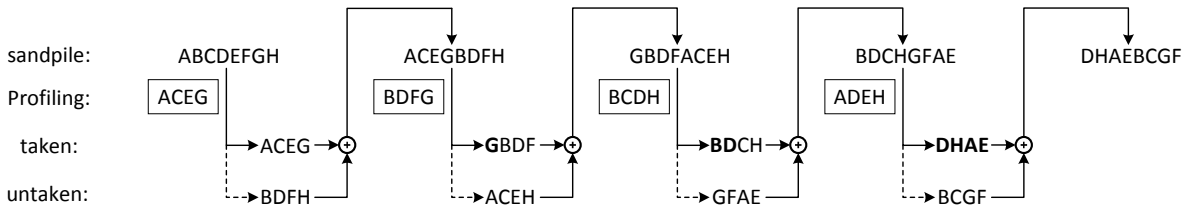


Fig. 3.5.: Intra-bucket scheduling example

```
1 void schedule() {
2   vector<int> *sandpile = {A, B, ..., H};
3   foreach(Node n in nodes) {
4     vector<int> *taken, *untaken;
5     for (i = 0; i < npoints; i ++) {
6       point_id = sandpile[i].id;
7       if(matrix[n->id][point_id])
8         taken->push_back(point_id);
9       else
10        untaken->push_back(point_id);
11    }
12    foreach (j in untaken)
13      taken->push_back(j);
14    delete sandpile, untaken;
15    sandpile = taken;
16  }
17 }
```

Fig. 3.6.: Intra-bucket scheduling code

scheduling bucket as a guide to the construction of the next. The procedure is illustrated in Figure 3.5. The vector *sandpile* records scheduling result of each scheduling step. Before first step, it is initialized with original input points's index, from **A** to **H**.

In the scheduling process executed on node ⑧, *sandpile* is filtered into two subsets: *taken* and *untaken*. The *taken* subset contains points that would traverse node ⑧ while *untaken* collects the rest. Then we join *taken* and *untaken* together and take the new formed *sandpile* as the input for node ⑩. Notice that the new *sandpile* still has the same elements as the old one, but already contains scheduling information.

In node ⑩, we check whether points visit this node in the order they lay in the *sandpile* which is transferred from node ⑧. Point **A**, **C** and **E** show no record in matrix, so we push them into *untaken* subset. Point **G**, **B**, **D** and **F** visited node ⑩ and should be pushed into *taken*. After we insert point **H** into *untaken*, we join the two subsets again and create

another new *sandpile*. Figure 3.6 shows the pseudo-code of the partition-join process. At the end of the scheduling, the *sandpile* is re-arranged as **DHAEBCGF**. Points **D** and **H** are put together because both of them visit node ⑭, as are **AE** (node ⑧), **BC**(node ⑫) and **GF**(node ⑩).

By continuously refining the schedule as we build the scheduling buckets, the *sandpile* eventually yields a final schedule that captures the similarity of points across multiple scheduling buckets. This process is quite efficient, as building a schedule for one million points across 256 scheduling buckets takes just a few tenths of a second.

Scheduling guided traversals

The scheduling matrix for guided traversals keeps both the node ID and traversal order for each point. Points' traversal are represented by a sequence of numbers, like the gene. What we would like to do is group together points with similar sequences together, as they will perform similar work. To do this, we iterate through each traversal sequence, partitioning points based on the order in which they visit nodes.

In Figure 3.7, the input points are filtered into four buckets based on the first leaf node they access: points **ABFH** would traverse node ⑧ first, while **CE** prefer node ⑩, **D** goes to node ⑫ and **G** visits node ⑮. Node ⑧'s bucket contains so many elements that we need to schedule them in finer granularity. In the second step[2], both point **A** and **F** traverse node ⑪ first then node ⑩ while **B** and **H** visit node ⑩ then ⑪. Point **A** and **F** present more similarity and thus should be arranged closer than others, so as point **B** and **H**. We could implement the same scheduling iteration multiple times until the end of the row, but we need to consider the cost. If there are only a few points left in a bucket, the divergence would be tolerable. We cut off our recursive scheduling process when the number of points is below some limits. A common example of such limit is thirty-two, the number of threads in a warp (line 17 in Figure 3.8).

[2]We define the traversal of two sibling nodes with the same parent as a step. A step that visits node ⑧ then node ⑨ is recognized as different from a step traverses node ⑨ then node ⑧.

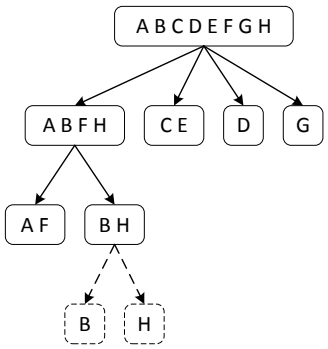


Fig. 3.7.: Guided optimization example

```
1 int *buffer = new int [npoints];
2 vector<int> sandpile = {A, B, ..., H};
3 schedule(sandpile, buffer, 0, 0);
4
5 void schedule (sandpile, buf, offset, index) {
6   clusters = new vector<int> [nnodes];
7   for (i  = 0; i < sandpile.size(); i ++) {
8     pos = sandpile[i];
9     temp = matrix[index][pos];
10    if(!temp)
11      cluster[temp].push_back(sandpile[i]);
12    else
13      buffer[offset++] = sandpile[i];
14  }
15  for(j = 0 ; i < nnodes; j ++) {
16    if(clusters[j].size() == 0) return;
17    if(index >= nnodes || clusters[j].size() <= 32)
18      for(k = 0; k < clusters[j].size(); k ++)
19        buffer[offset ++] = clusters[j][k];
20    else
21      schedule(cluster[j], buffer, offset, index + 2);
22  }
23 }
```

Fig. 3.8.: Guided optimization code

Figure 3.8 shows the pseudo-code for guided traversal optimization. Assume that the number of input points is **npoints**, and the *top tree* has **nnodes** leaf nodes. We initialize a vector with **npoints** well-aligned elements that presents the original order of input points, and an empty sequence as the output. According to the first traversed node, we distribute **npoints** points into **nnodes** buckets. For the points in a bucket, we repeat the schedule until

the number of points falls below a threshold. After all the points in the one bucket is well sorted, the program moves to the next bucket.

Since the number of nodes that a point would traverse varies, the length of a row may also change. In above pseudo-code, each node has an ID from 1 to **nnodes**, while the cells in matrix are initialized as zero. When the entry in a row changes from non-zero to zero, that means the traversal of a thread finishes. The thread that ends up earlier than others should be removed from further scheduling. We insert it in the bottom of the output buffer (line 12).

3.2.3 Execution

The final stage of our process is the execution phase. We simply run the original GPU kernel (with the addition of our optimized lockstep skeleton, described in the next section) for the traversal algorithm using the order of points determined during scheduling. Since this order of points is based on points that we expect to behave similarly, this has an analogous effect to when Goldfarb *et al.* ran their kernels on *sorted* input points.

Note that we re-run the *entire* traversal algorithm using the new schedule of points. In the profiling step, we execute the tree traversal algorithm for certain depth. Since the profiling work performs some of the work of the original algorithm, we could store the computation result of profiling, and restore from these break points in execution step. However, this requires communicating tremendous amounts of data back from the GPU after the profiling step. Instead, updates to points that occur during profiling on the GPU are *not* communicated back to the CPU. All we communicate back is the scheduling matrix.

While this strategy does result in redundant work, as the top tree is traversed a second time during the execution phase, the amount of time spent in the top tree overall is negligible, and hence this has very little impact on performance. Indeed, our experiments have shown that the expense of communicating point data back and forth and restore from the partial traversal results can actually *slow down* performance.

3.2.4 Correctness

Correctness is far more important than performance for any kind of code transformation and scheduling. We argue that our hybrid-scheduling is sound. Note that although scheduled code walks through the tree in a different order from original, for each point, scheduling does not change the nodes it visits, nor the order it visits them. For a given node, points that visit it are the same set, and the place where the value is updated is also preserved.

3.3 Implementation

This section describes an alternate lockstepping implementation that performs far fewer memory accesses than Goldfarb *et al.*'s original lockstep kernel, substantially improving performance.

Goldfarb *et al.*'s lockstep kernel operates as follows: each warp has a bit vector with one entry per thread called the *mask vector*. If a thread wants to truncate at a node, its bit in the mask vector is set. If any thread in the warp *does not* truncate (*i.e.*, not all bits in the mask vector are set), all threads continue recursion. The mask vector is used to supress the computation of threads that are "carried along" with the warp even though they wanted to truncate higher in the tree.

While this approach successfully implements lockstepping, it has a performance penalty. The mask vector needs to be preserved throughout a warp's traversal. As a result, the mask vector is treated as an argument to the recursive method, and hence is pushed and popped as part of the function call stack, or, in the case of autoropes, the rope stack. Because this stack can get quite large, it is stored in slower memory. Hence, the repeated pushes and pops of the mask vector introduces substantial overhead.

We propose a new lockstep kernel that, rather than using a mask vector that must be preserved on the (function or autoropes) stack can instead be maintained in a register, dramatically reducing memory accesses. While the warp traverses the tree, each thread conceptually runs a finite state machine (FSM) with two states: *truncated* and *traversing*. A

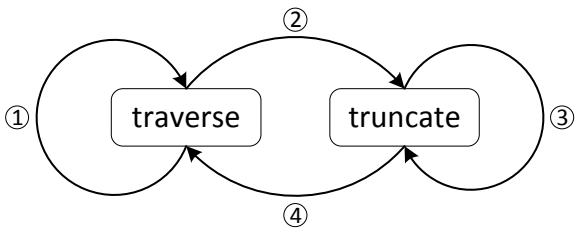


Fig. 3.9.: FSM state transfer

thread only performs computation while it is in traversing state. Figure 3.9 shows this state diagram and Figure 3.10 shows the pseudocode for the lockstep kernel, indicating when state transitions are taken.

All the threads in a warp are initially in the traversing state. When a thread reaches a condition that leads it to truncate its traversal (line 15), it transitions to the truncate state. The question is, when does the thread transition back to the traversing state? To keep track of this, the thread stores the current depth of the stack, `sp` in a local variable `critical`. When the stack depth returns to `critical`, the thread transitions back to traversing state (line 7).

The FSM lockstep kernel has two key features. First, as long as a thread is truncated, it performs no computation (line 30). Second, unlike Goldfarb *et al.*'s lockstep kernel, we need not track a mask vector. Instead, each thread simply tracks whether it is truncated. Nevertheless, all the threads are still carried along through the tree, preserving the memory coalescing benefits of lockstep traversal.

We note one other feature of the lockstep kernel. If some threads in the warp want to take one path through the traversal, while other threads want to take a different path, lines 20–26 determine which direction the majority of the non-truncated threads want to go, and send all the threads in that direction. This behavior ensures that all threads in the warp *dynamically* select a single path through the tree.

Example In previous unguided traversal example, the warp diverges at node ④ when the depth of stack is 2. Point **B** is truncated (*critical = 2*) and wants to visit node ⑤, but the warp wants to traverse the subtree rooted at node ④. The warp pushes node ⑧ and ⑨ into the *stack* and increases the *sp* to 4. The *stack* keeps growing when node ⑧ or node ⑨ also

```
1 __global__ void gpu_kernel() {
2   flag = 1; critical = INT_MAX; cond = 0;
3   sp = 1; //current depth of the warp-shared ropes stack
4   for(pidx = blockIdx.x * blockDim.x + threadIdx.x;
5     pidx < npoints; pidx += blockDim.x * gridDim.x) {
6     while (sp >= 1) {
7       if (sp <= critical)
8         flag = 1;                 // transition 4
9       sp --;
10      if (flag) {
11        //compute condition
12        cond = ...;
13        if (!__any(cond))
14          continue;
15        if (!cond) {              // transition 2
16          flag = 0;
17          critical = sp;
18        } else {                  // transition 1
19          // compute branch condition
20          cond_left = ...;
21          cond_right = ...;
22          vote_left = __ballot(cond_left);
23          vote_right = __ballot(cond_right);
24          num_left = __popc(vote_left);
25          num_right = __popc(vote_right);
26          if (num_left > num_right)
27            // stack operation
28          else
29            // stack operation
30        }
31      } else {}                   // transition 3
32    }
33  }
34 }
```

Fig. 3.10.: FSM lockstep kernel

has children, but the node ⑤ cannot be visited unless all these nodes above it have been popped. In other words, the traversal of point **B** would never resume before *sp* drops below *critical* value.

3.3.1 Profiling division

The profiling step needs a substantial amount of memory to store the profiling matrix. Suppose we have one million points as input, and the *top tree* has 1024 leaf nodes. And the

application is guided algorithm, which means matrix elements should record *int* value. The memory cost could be up to 4GB (4B * 1K * 1M)! For large inputs, this profiling matrix can easily outstrip a GPU's memory. We partition the input points into several groups to fulfill the memory requirements. We assign groups into different streams and overlap the data transfer with kernel execution. The profiling matrix for groups are aggregated on the CPU side before scheduling phase. Thus this partition won't hurt the correctness and scheduling performance.

3.4 Evaluation

3.4.1 Methodology

Platform We evaluate our benchmarks on a server with two AMD Opteron 6164 HE Processors, each of which contains 12 cores running at 1700MHz. The GPU is an nVidia Tesla K20C with 5120 MB GDDR5 memory and 2496 CUDA cores. The system has 32GB system memory and runs on Red Hat 6.6 with Linux kernel v2.6.32.

Benchmarks We evaluate our scheme on six benchmarks:

Point Correlation (PC) is an important algorithm in bioinformatics and data mining field. The two-point correlation can be computed, for each point in a data set, by traversing a kd-tree to count the number of other points that fall within a certain radius.

Nearest Neighbor (NN) finds the nearest neighbors of points in a metric space. NN builds a kd-tree over a set of input points. It then takes a set of *query* points, and for each query point traverses the kd-tree to find its nearest neighbor.

k-Nearest Neighbor (kNN) is a non parametric instance-based learning algorithm widely used for classification and regression. Unlike NN, which finds the nearest neighbor of a query point, kNN finds the k nearest neighbors [13].

Ball Tree (BT) is a variation of nearest neighbor that uses ball trees, where the multi-dimensional space is partitioned by hyperspheres.

Vantage Point (VP) is a variation of nearest neighbor search that uses vantage point trees rather than kd-trees: subspaces are split according to distance from a chosen vantage point.

Barnes-Hut (BH) performs an n-body simulation [1]. BH recursively divides the set of n bodies into groups by sorting them in an octree. Then each body traverses the tree to calculate the gravity acting upon it. We replace the force computation part of the original code with our GPU variants, but leave the remainder of the code the same. We time the force computation phase of a single iteration.

Inputs The first five benchmarks are evaluated with four inputs: Covtype, Mnist, and Rand_Dim7, each of which contains 400,000 7-dimensional points, and Geocity, which contains 400,000 2- dimensional points. For the four nearest-neighbor problems (NN, kNN, VP, BT)[3], we partition the 400,000 points into two subsets, S_1 and S_2. S_1 is used to build the tree, while the points in S_2 are the query points[4]. BH is evaluated with two 1-million bodies inputs: Plummer, which uses the Plummer model to generate bodies, and Rand_Dim3, with uniformly randomly generated bodies.

Evaluation methodology For each benchmark, we evaluate five variants: the original kernels (Goldfarb *et al.*'s original lockstep code) and our new FSM lockstep kernel on both unsorted and sorted inputs, and our hybrid scheduling approach on the unsorted input. The hybrid scheduling variant uses the FSM lockstep kernel. We choose the original kernel with unsorted input as the baseline, and take the application specific sorting cost as the baseline overhead.

File I/O, tree building, etc. are not targets for optimization, so we do not include those components in our timing. We measure the runtime spent in GPU execution of tree traversals, as well as time spent in profiling and scheduling.

3.4.2 Results

[3]The implementation of BT we use does not work with two-dimensional data, so we do not evaluate BT on the Geocity input.

[4]In the results we present, S_1 and S_2 each contain 200,000 points. We evaluated different random subsets of points, as well as different sizes for S_1 and S_2 and found qualitatively similar results.

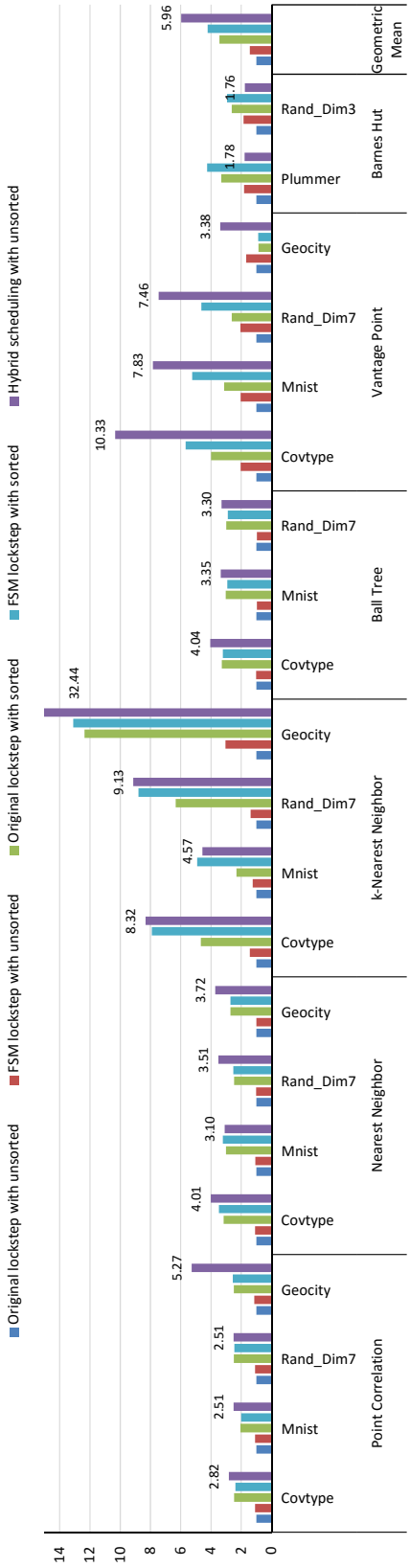


Fig. 3.11.: Speedup comparison

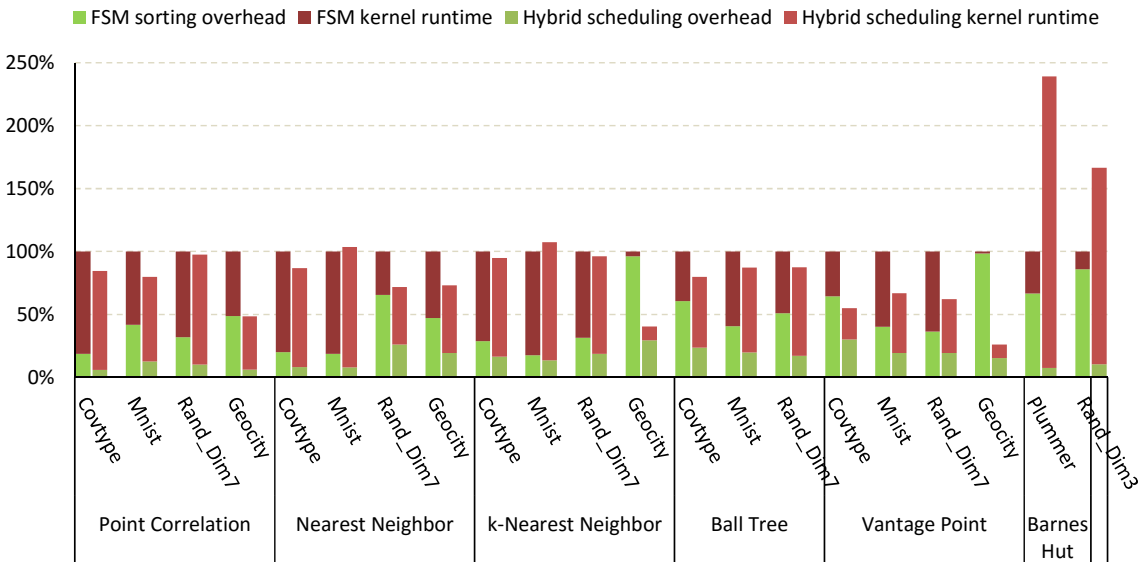


Fig. 3.12.: Overhead cost ratio comparison

We begin by comparing the performance of the different implementations of tree traversal algorithms. Figure 3.11 shows the speedup of all of the implementations over the baseline. The columns in every benchmark are arranged in the following order: the original lockstep with unsorted input, the FSM lockstep with unsorted input, the original lockstep with hand-sorted input, the FSM lockstep with hand-sorted input and the FSM kernel with hybrid scheduling strategy. The percentage value presents the speedup of hybrid scheduling over the baseline.

The primary comparison is between Goldfarb *et al.*'s baseline (Original lockstep with unsorted) and our full system (Hybrid scheduling with unsorted), which implements both our hybrid scheduling algorithm and our optimized traversal kernel. We see that our hybrid scheduling algorithm yields, even in the worst case, a 1.76× speedup over the baseline. ***On average, our technique is 5.96× faster than the baseline.*** Hence, when presented with the original inputs, our new approach delivers enormous gains over the previous, best-known general implementations.

We also isolate the benefits of our optimized kernel by using the original schedule, but using the optimized kernel (FSM lockstep with unsorted). We see that across the board, the optimized kernel is faster than the original kernel, though some times the gains are

minimal. Ultimately, we can conclude that most of the performance gains from our new techniques come from the optimized schedule.

Finally, to show the effectiveness of our automatic scheduling approach over a hand-tuned approach that uses application-specific sorting routines to derive schedules for each application, we compare our technique (Hybrid scheduling) to using the optimized kernel on *sorted* inputs (FSM lockstep with sorted). We see that, with the exception of Barnes-Hut, *our technique is always faster*. In other words, on average, **our automatic technique is 1.41× faster than hand-tuned implementations**!

Barnes-Hut is the one outlier in our comparisons to the baseline (yielding the smallest improvement) and to hand-tuned schedules (where the hand-tuned schedules are faster). In other words, our hybrid scheduling is less effective for Barnes-Hut than for other benchmarks. In Barnes-Hut, traversals each visit a larger fraction of the tree than in other benchmarks. In other words, there is relatively *less* divergence between traversals in the tree. Because our scheduling algorithm relies on thread divergence as a signal for sorting, Barnes-Hut's lack of divergence limits the effectiveness of our scheduling. Sections 3.4.3 and 3.4.3 explore this result in more detail.

3.4.3 Performance breakdowns

The following subsections explore the performance results in more detail. First, we investigate the performance of our techniques relative to the hand-tuned techniques, studying both the cost of scheduling and the cost of execution. Second, we explore how our techniques' effectiveness varies with the depth of the profiling phase. Third, we isolate the performance of our optimized lockstep kernel with Goldfarb *et al.*'s original traversal kernels. Finally, we directly assess the effects of hybrid scheduling on divergence.

Scheduling time ratio

Figure 3.12 shows the ratio that the overhead of hybrid scheduling strategy and hand-tuned sorting over the whole runtime. In our hybrid scheduling, the extra work includes

the profiling execution, matrix transmission from GPU to CPU, CPU scheduling, and final result transmission back to the GPU[5]. For hand-tuned sorting, the extra work is only the application-specific sorting procedure.

Our hybrid scheduling strategy shows consistent advantages over application-specific sorting in the majority of benchmarks. We note that in many cases, the advantage of our hybrid scheduling strategy comes not from better schedules, but from the fact that the overhead of sorting in the hybrid strategy is far smaller. This result justifies our decision to perform profiling on the GPU, where the parallelism advantages of GPU execution win out.

Profiling depth sensitivity analysis

Our hybrid scheduling tunes the execution order of the original input sequence according to the information collected during the execution of top tree. That means that the larger the top tree is, the more knowledge our scheduling may learn, and thus the better the resulting schedule, at the cost of additional inspection overhead. Balancing the benefits of more profiling information against profiling overhead is a classic problem in any profile-guided optimization.

Figure 3.13 shows the sensitivity of our benchmarks' performance to how deep in the tree the inspection phase runs for the binary tree–based benchmarks (*i.e.*, all but BH). In all cases, we see the expected U-shaped curves: as depth increases, more profiling information leads to better schedules, until the overhead of profiling outweighs the scheduling benefits. Interestingly, we see that the optimal depth is roughly the same in all cases, regardless of benchmark or input. This suggests that for binary tree–based benchmarks, the profiling depth can be set in an application- and input-independent manner.

We further explore the overhead/effectiveness tradeoff in Figure 3.14, which separately measures the runtime of the inspection and execution phases. Figure 3.14(a) shows, for kNN, the expected behavior: as profiling depth increases, kernel execution time decreases,

[5]Note that although the profiling matrix is large in size, we may overlap its transmission with profiling kernel execution through CUDA streams.

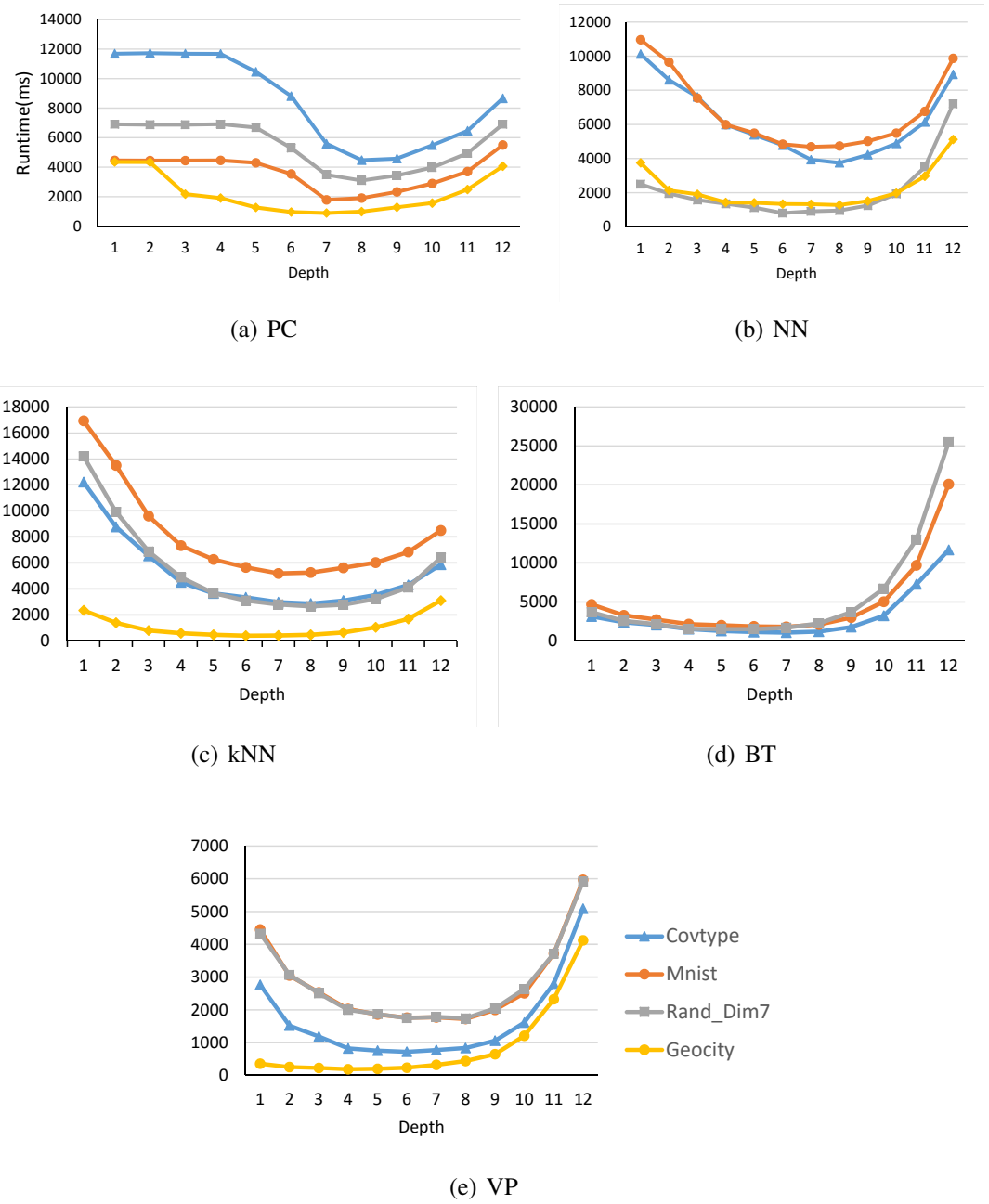
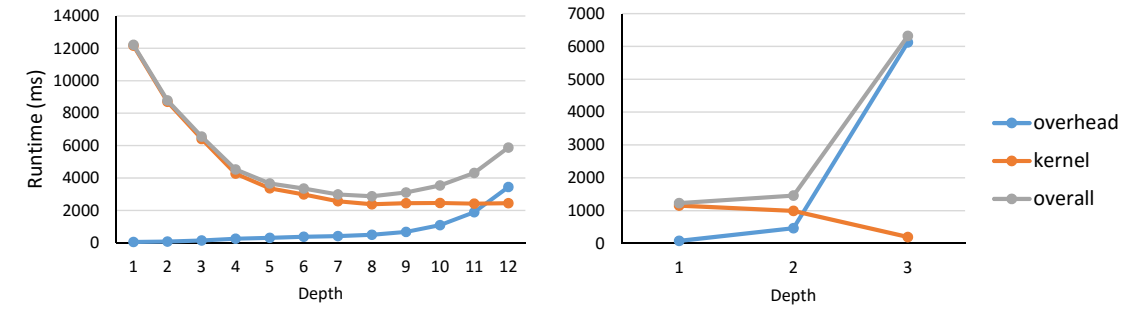


Fig. 3.13.: Depth sensitivity analysis

but overhead increases. Figure 3.14(b) shows the same breakdown for BH. Here, we see further evidence for why our hybrid scheduling underperforms for BH. While the trends match the other benchmarks, BH's octree means that overhead increases dramatically with



(a) kNN_Cov (b) BH_Rand_Dim3

Fig. 3.14.: Detailed depth sensitivity analysis

even a small increase in profiling depth. Hence, it is impractical to try and capture more profiling information, resulting in a poorer-quality schedule.

FSM kernel versus original kernel

Figure 3.11 lets us isolate the performance of our optimized lockstep kernel. With unsorted input, the FSM lockstep kernel presents similar or slightly better performance over baseline. The average speedup for FSM lockstep is 1.22×. When the input point set is hand-sorted, the FSM lockstep kernel shows significant advantage over baseline, and even the original lockstep traversal with the same hand-sorted input. The average speedup for FSM lockstep with hand-sorted input is 4.22×, while the original lockstep with hand-sorted input only provides 3.45× speedup.

Hybrid scheduling effects on divergence

The lockstep kernels, which force traversals to "stay together" in the tree, converts memory divergence (accessing different parts of the tree) into branch divergence (forcing traversals to "pause" while other traversals access a portion of the tree). Thus our hybrid scheduling's effect on reducing branch divergence can essentially addresses both concerns.

In the lockstep kernel, all threads in a warp traverse the same portion of the tree. We can thus evaluate the effectiveness of scheduling by measuring the total number of nodes

Table 3.2.: Scheduling effects on divergence

Benchmark	Input	Baseline (ms)		Average num of nodes visited by a warp		
		runtime	overhead	Unsorted	Sorted	Hybrid
k-Nearest Neighbor	Covtype	23904	861	147977	15785	31805
	Mnist	24048	861	173048	32002	53574
	Rand_Dim7	24261	870	158046	13321	27931
	Geocity	13526	994	57865	271	1240
Vantage Point	Covtype	8869	1016	105896	11758	10976
	Mnist	13694	1042	150307	36288	33919
	Rand_Dim7	13103	1043	142820	40924	34981
	Geocity	636	711	19178	337	507
Nearest Neighbor	Covtype	14996	850	256560	53813	132900
	Mnist	14681	848	265901	65607	136449
	Rand_Dim7	3335	864	70676	8783	17937
	Geocity	4655	808	131793	21779	103805
Ball Tree	Covtype	4305	806	74660	8263	24373
	Mnist	6081	847	112330	20478	42482
	Rand_Dim7	5004	882	92135	13611	28207
Point Correlation	Covtype	12752	994	256374	76047	80038
	Mnist	4835	987	95624	26021	31664
	Rand_Dim7	7811	1005	156050	37605	54155
	Geocity	4663	881	120388	22223	27312
Barnes Hut	Plummer	4395	680	22106	3346	19847
	Rand_Dim3	2163	630	10797	984	9291

visited *by a warp*. If a warp contains highly-convergent threads, it will visit fewer nodes than if the threads diverge and want to touch more of the tree. Table 3.2 shows the results. We see that hybrid scheduling results in a large decrease in the number of nodes visited per warp compared to the baseline: on average, warps scheduled using the hybrid approach traverse *7.15*× fewer nodes than the baseline, a substantial decrease in divergence.

We note that this evaluation shows that application-specific sorting yields even better convergence (about 2× better, on average)—an unsurprising result. However, as discussed in Section 3.4.3, while hand sorting yields more convergence and hence faster kernels, our hybrid approach spends less time in scheduling, yielding better performance overall.

Note that these results further explain hybrid scheduling's performance on BH. Not only is our profile-guided sorting relatively ineffective at reducing divergence (indeed, our 1.76× speedup comes almost entirely from the optimized kernel), application specific sorting is *highly* effective.

3.5 Related Work

Most of the research work about tree traversal focuses on GPU-only implementation, and is targeted at specific tree traversal algorithms. Foley and Sugerman propose two variations of kd-tree traversal that take advantage of bounding box to eliminate per-ray stack, and rely on kernel masking and scheduling to reduce overhead [5]. Horn *et al.* extends Foley *et al.*'s work by the usage of packet, restarting from lower subtree instead of the root and stack-based restart check [8]. Popov *et al.* develop another stackless kd-tree traversal approaches for GPU ray tracing by adding ropes to leaf nodes and using packets of rays [11].

Hybrid scheduling is mostly used in other irregular applications, such as clustering and map-reduce. Ren *et al.* use a hybrid approach for k-means computing with very large data sets [36]. They execute the map on the GPU, transfer the result back to the CPU, and execute the reduction on the CPU. Rather than assign map and reduce separately, Ravi *et al.* parallelize the computation by splitting the input data set into sections and distribute every section to either CPU or GPU [37]. Ravi and Agrawal also describe a scheduling framework for data parallel loop by configuring application with various behavior patterns for heterogeneous architecture through a *cost* model [38].

Zhang *et al.* propose a generic framework for handling irregular GPU computation called *G-Streamline* [39]. G-Streamline adopts a similar inspector-executor style approach as our hybrid scheduling: the CPU determines the set of data accesses that a set of GPU threads will perform and then remaps data or reorders GPU threads to improve convergence. G-Streamline primarily targets accesses through indirection arrays and relies on knowing (or approximating) which data a GPU thread will access prior to execution. In addition, G-Streamline's rescheduling heuristic relies on each thread's accessing a small amount of data. The tree applications our approach targets do not have these characteristics: we must generate a GPU profiling pass to predict the similarity of threads (unlike G-streamline, inspection is performed on the GPU to avoid high profiling overheads), and then use a more sophisticated scheduling algorithm to effectively handle tree applications.

Most other work on heterogeneous execution has focused on taking a set of tasks and dividing them between the CPU and GPU. Qilin is an API and runtime programming system that automatically maps computation to CPU and GPU cores [40]. It adopts offline profiling to analyze programs and adaptively maps them to an analytical performance model to determine actual job distribution. Kaleem *et al.* also propose two profiling based scheduling algorithms that automatically partition workload between CPU and GPU [41]. They showed that profile-based scheduling for integrated GPU has very little overhead and presents comparable efficiency with offline model.

3.6 Conclusions

We describe a general, application-agnostic scheduling framework for tree traversals that automatically uses partial execution to inspect GPU threads' behaviors, and then uses that information to reorder execution on the CPU prior to re-execution on the GPU. We show that the scheduling problem is NP-hard, and develop optimized version of scheduling according to structural properties of traversal algorithms. We also introduce a new skeleton for GPU kernels that uses a register-level finite state machine to minimize memory access. Our experiments show that our work significantly outperforms the baseline, and can even outperform hand-tuned, application-specific scheduling.

4. LOCAL TREE WALK ON DISTRIBUTED HETEROGENEOUS SYSTEM

In the previous chapter, we have presented that our new GPU kernels are able to deliver great performance over existing work by reducing memory usage and cutting off unnecessary traversal. In this chapter, We use our kernel to help ChaNGa, a dual tree based, interaction-list GPU implementation for n-body simulations. ChaNGa is the best-of-breed n-body platform targeting at large scale, large spatial and temporal ranges cosmological simulations. It uses an asymptotic-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. In this chapter, we describe that an efficient single-tree algorithm is able to perform local tree walk in large scale n-body simulations efficiently. We made an observation that the asymptotic advantage of the dual-tree algorithm is counteracted by its resistance to parallelism. Using our recent developments in GPU tree walks [25], we show that our approach is able to overcome the higher asymptotic complexity of single-tree walk and beat the dual-tree walk version ChaNGa [1].

We start with a brief introduction to ChaNGa and describe how we realize the dual-tree walk is not suitable for the heterogeneous system. Then we demonstrate our design and implementation and evaluate our work with several benchmarking inputs.

4.1 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.

[1]This part of work has been merged into ChaNGa master branch

4.1.1 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 size of a treepiece is a tradeoff between maximizing granularity for task parallelization (arguing for many small treepieces) and the booking-keeping overhead (arguing for few large treepieces). The Charm++ run

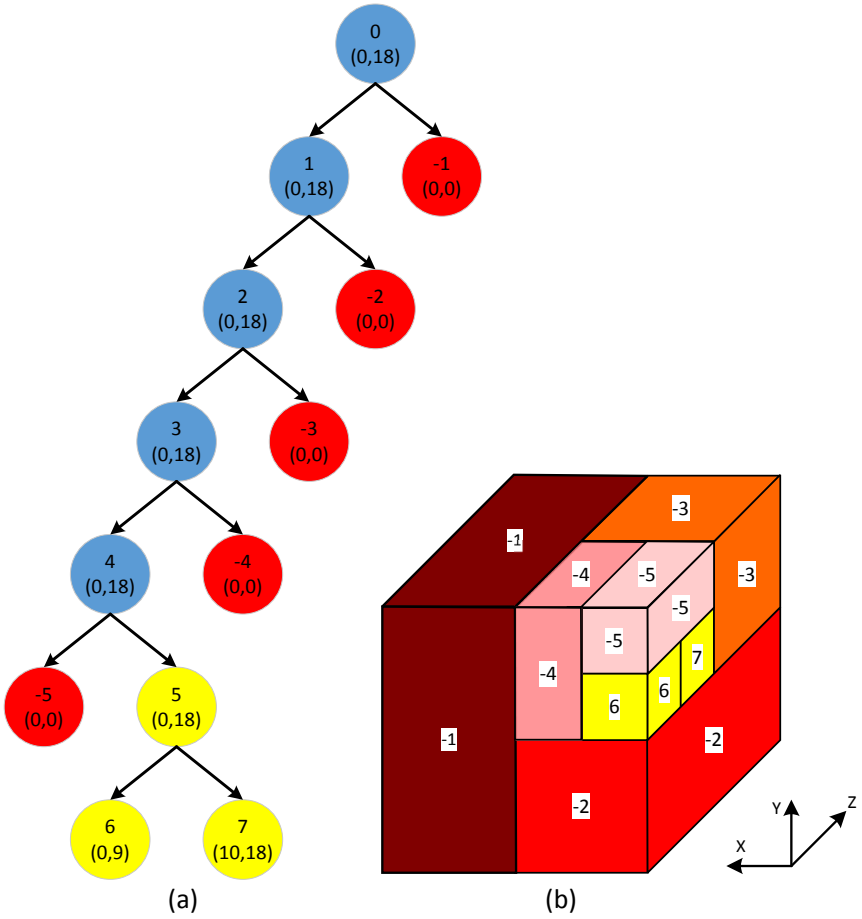


Fig. 4.1.: (a) Tree structure: Each chare builds its piece of the tree starting with internal nodes (yellow), eventually getting to boundary nodes (blue) which have non local children (red) and are duplicated on other tree pieces. The top boundary node is the root which is duplicated on all tree pieces. 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.)

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 4.1).

4.1.2 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 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.

```
1 void dualTreeWalk(outerNode, checkList, interactionList){
2   for (innerNode : checkList) {
3     checkList.remove(innerNode);
4     action = openCriterion(innerNode, outerNode);
5     if (action == CONTAINED) {
6       checkList.append(innerNode.children());
7     } else if (action == INTERSECT) {
8       if (isBucket(innerNode)) {
9         interactionList.plist.append(innerNode.particles());
10      } else {
11        checkList.append(innerNode.children());
12      }
13    } else {  // TRUNCATED
14      interactionList.clist.append(innerNode);
15    }
16  }
17  if (isBucket(outerNode))
18    calculateGravity(outerNode.particles(), interactionList);
19  else {
20    dualTreeWalk(outerNode.left, checkList, interactionList);
21    dualTreeWalk(outerNode.right, checkList, interactionList);
22  }
23 }
```

Fig. 4.2.: 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.

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. As a result, the complexity of dual-tree traversals drops from $O(n \log n)$ to $O(n)$. Figure 4.2 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 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, you 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 4.2 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.

4.1.3 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 [42].

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

[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 [**?**, **?**]. Because FMM builds its interaction lists differently, we would not expect our GPU traversal approach to be as effective (see Section 4.5).

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.

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.

4.2 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.

4.2.1 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 4.4, 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 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 [4, 12, 25]. 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 [4]. 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. [12] 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. [25] 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, allowing for more concurrent traversals to be performed by the GPU.

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.

4.2.2 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 the overlapping

CPU/GPU computation and the 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 4.2).

4.2.3 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 4.3. 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.

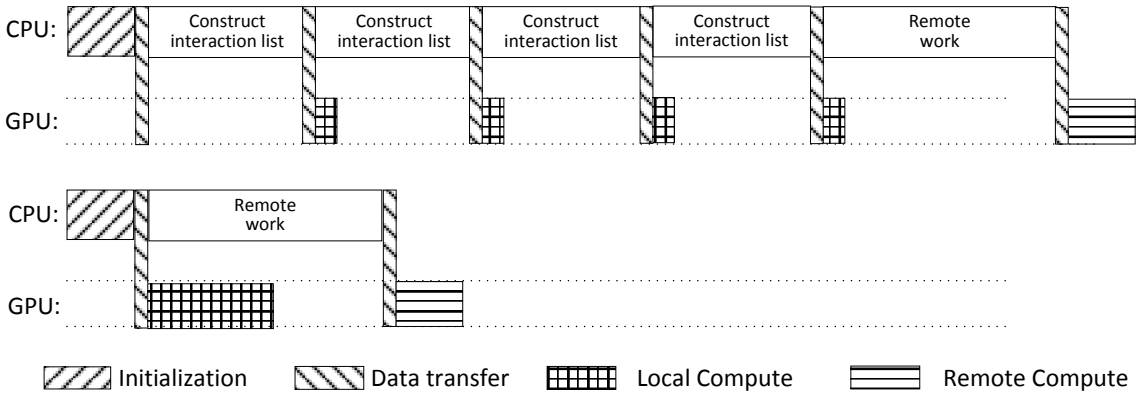


Fig. 4.3.: Strategy comparison: (1) the upper part is the original ChaNGa. The whole interaction list building process is chopped into pieces to overlap CPU/GPU computation time. When the number of lists reaches a threshold, the CPU sends the built lists to the GPU for computation then resumes the building 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.

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.

Table 4.1.: Number of target tree interactions with original truncation condition vs. single tree condition.

Benchmarks	Simulated original ChaNGa	Single tree walk
lambs	6.16E+09	2.41E+09
lambb	3.42E+11	1.18E+11
dwf1	8.75E+09	3.51E+09
dwf1.6144	1.18E+11	4.46E+10

4.3 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 open 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 open criterion: each particle tests the open 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 4.1. Furthermore, while the two open criteria result in similar interaction list sizes, the bucket-to-node open criterion conducts more truncations than the node-to-node open criterion. It requires more particle-cell computation, but saves much more particle-particle computation (Table 4.2). Note that our new implementation achieves the same level of accuracy as the original ChaNGa in standard accuracy tests.

Table 4.2.: The comparison of the total size of interaction lists for different open criterion strategies. The "particle" represents the particle-particle interaction list, and the "cell" implies the particle-cell interaction list.

Benchmarks	node-to-node		bucket-to-node	
	cell	particle	cell	particle
lambs	9.55E+08	4.30E+09	1.00E+09 (+4.5E08)	3.37E+09 (-9.3E08)
lambb	5.30E+10	1.17E+11	5.33E+10 (+3E08)	9.61E+10 (-2E10)
dwf1	1.49E+09	4.52E+09	1.52E+09 (+3E07)	4.02E+09 (-5E08)
dwf1. 6144	1.98E+10	4.29E+10	1.99E+10 (+1E08)	4.12E+10 (-1.7E09)

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 at the cost of low occupancy and hence 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 data (48 bytes) for opening criteria 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 4.4 shows pseudocode for the GPU tree walk. For the most part, this pseudocode tracks the single-tree Barnes-Hut pseudocode in Figure 2.1. 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. [12].

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* optimization [25], 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.

4.4 Evaluation

This section evaluates the effectiveness of our GPU local tree walk in ChaNGa. First, we compare the tree walks' performance with the original ChaNGa GPU approach. In Quinn *et al.* [43]'s work, they compared ChaNGa (our baseline) with the existing N-Body code PKDGRAV, and found comparable performance and better scaling. We analyze the details of why our implementation wins or loses in performance. Then we investigate scalability.

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

```
1 void gpuLocalTreeWalk() {
2   stk = new stack();  // on GPU shared memory
3   for(pidx = globalThreadIdx; pidx < numParticles;
4     pidx += blockSize * gridSize) {
5     init(stk); bucket = loadBucketNode();
6     while (!stk.empty()) {
7       target = stk.top(); stk.pop();
8       if (current_thread_need_to_work()) {
9         action = openCriterion(bucket, target);
10        cond = isContainedOrIntersect(action);
11        if (action == COMPUTE_NODE) {
12          if (openSoftening(target, node))
13            // evaluate as a MonoPole
14          else
15            // evaluate as a MultiPole }
16        else if (action == COMPUTE_PARTICLES) {
17          for (particle : target)
18            // evaluate the force for particle }
19        if (!__any(cond))
20          continue;
21        for (child : target.children())
22          stk.push(child.Idx)
23        }
24      }
25    }
26  }
27 }
```

Fig. 4.4.: GPU single tree walk

70 megaparsecs in size. ***lambb*** is an 80 million particle representation of that same volume. These simulations were originally used in [44]. ***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, 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 4.3 (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.

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`

Table 4.3.: Runtime Comparison on Comet (Time in seconds)

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

Table 4.4.: Runtime breakdown for original ChaNGa

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 4.5.: Runtime breakdown for new 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

for profiling) with the **lambs** input. In original ChaNGa (Table 4.4), 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. In contrast, our implementation (Table 4.5) 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!

Scalability Finally, we evaluate strong and weak scaling, with one process per node. For weak scaling (Figure 4.6) we use synthetic data with 1M–8M Poisson distributed particles. We do weak scaling with 1M particles per node. For strong scaling (Figure 4.5), 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.

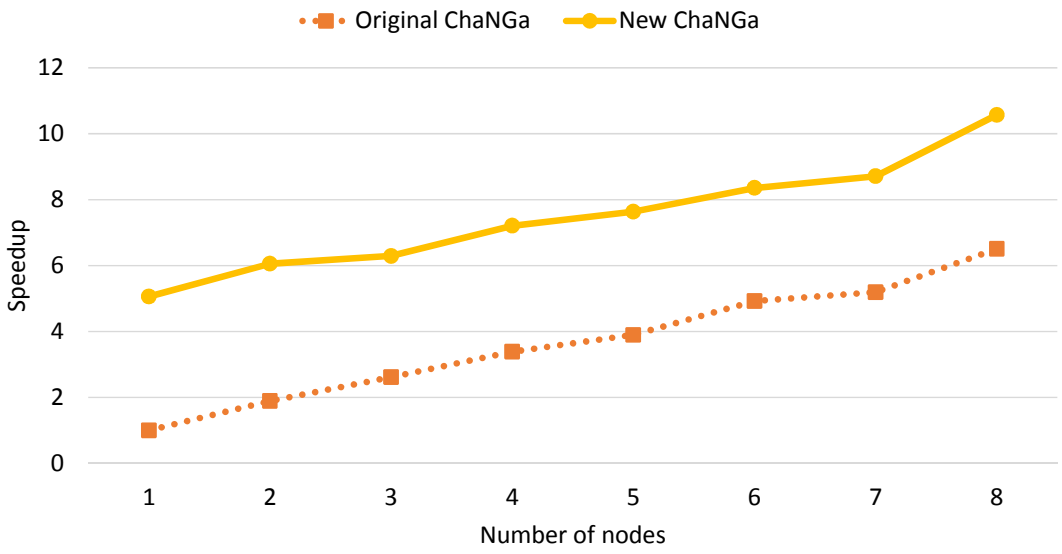


Fig. 4.5.: Strong scaling. Base line is original ChaNGa with 1 node.

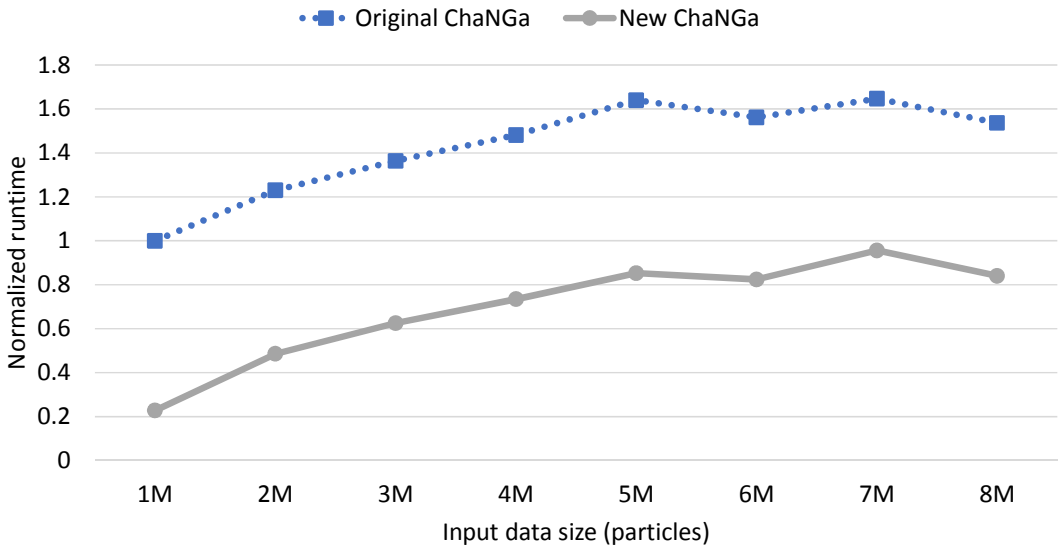


Fig. 4.6.: Weak scaling. Base line is original ChaNGa with 1 node.

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 `openCriteria` operation. 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 [45,46]. However, we want to study the potential of our approach and evaluate its performance under

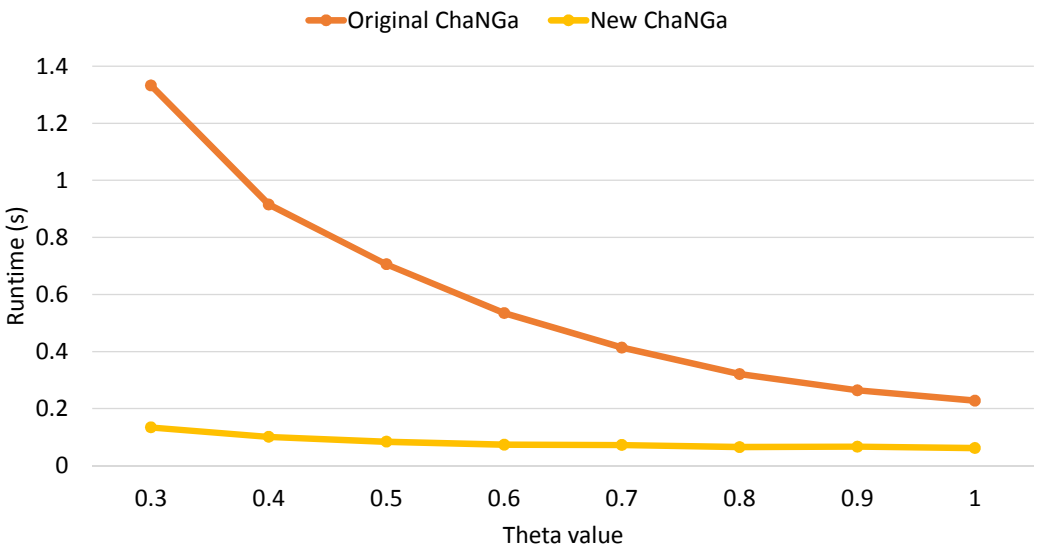


Fig. 4.7.: Runtime comparison under different theta values

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.

Performance model We admit that a performance model would have helped put the performance analysis on a firmer footing and also better communicate the underlying complexity and rate limiting steps. Unfortunately, such a model is hard to derive because of ChaNGas high input-dependency and system complexity. In above evaluation, weve provided as many data points towards a sensitivity analysis as we can, e.g. different input sizes, theta values, uniform/non-uniform data sets, though we acknowledge this is not as helpful as a good performance model.

4.5 Related Work

There has been much research about mapping tree traversals onto GPUs. Gieseke et al. [47] presented a novel data structure called a buffer kd-tree for massive nearest neighbor queries. It uses a shallow kd-tree and makes each leaf node contain a huge number of points. For a query point, it traverses the top kd-tree as normal but does an exhaustive search within each leaf node. They claim that the GPU can do the exhaustive search quickly and the irregularity of the kd-tree traversal has been greatly reduced. However, their work needs the CPU to manage the GPU's work frequently, which is expensive in message-driven distributed systems. Similarly, Liu et al.'s [25] work also relies on the CPU to schedule work for GPU execution. The CPUs in distributed systems usually have a heavy load doing message communication and thus are not always ready for the GPU. So neither of these works are particularly suitable for the distributed scenario.

Many people have devoted effort to tree or graph traversal for distributed systems. Hegde *et al.* [48] 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, the *SPIRIT* sends the particle to the target remote subtree, asking the subtree to do the computation and send back the result. Several papers parallelize the massive graph traversal from a vertex-centric perspective. Pregel [49] is a bulk-synchronous message passing abstraction that runs an all vertex-program in a lockstep way. After a **super-step**, a barrier is imposed to synchronize the action among all program instances. Low *et al.* [50] developed the *Distributed GraphLab* framework that using message passing to avoid the inefficiency associated with synchronization. None of above frameworks support GPU acceleration.

4.6 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, this dual-tree variant is ill-suited for GPUs, so 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 dissertation, 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.

5. REMOTE TREE WALK ON DISTRIBUTED HETEROGENOUS SYSTEM

5.1 Introduction

As the development of measurement and data collection techniques, people are facing the surge of data both in quantity and quality. The pressure of extracting useful information actuates people to borrow power from the distributed system and heterogeneous system. We have already seen that institutes and companies built larger, more powerful clusters (e.g. the latest Summit supercomputer in ORNL) to strengthen their ability. But efforts to related software support are still lagging, which we think is even more crucial to the performance.

Some researchers have already proposed approaches [12, 47] to map n-body problem on a single machine. And there are tremendous of papers about optimizations for n-body problems on the distributed system. However, to our best knowledge, there are few works focus on the optimization for n-body problems on the distributed heterogeneous system.

The fast development of GPU hardware makes the software design for distributed heterogeneous system significantly different from previous ones for distributed homogenous systems or even distributed systems with weaker GPUs. We are dealing with challenges from these aspects:

1. The performance gap between CPU and GPU is growing. Figure 5.1 [1] shows the trends of the GPU/CPU computational ability [51]. In 2007, the best GPU is 4× faster than the best CPU, while the strongest GPU runs 25× faster than the fastest CPU [2]. It's unreasonable to neglect the GPU's power and treat it simply as a co-processor.

[1] Figure source: http://www.irisa.fr/alf/downloads/collange/cours/ opt2018/opt2018 gpu 1.pdf
[2] We only consider desktop CPUs, and the performance gap between desktop and server CPU is "only" 4×.

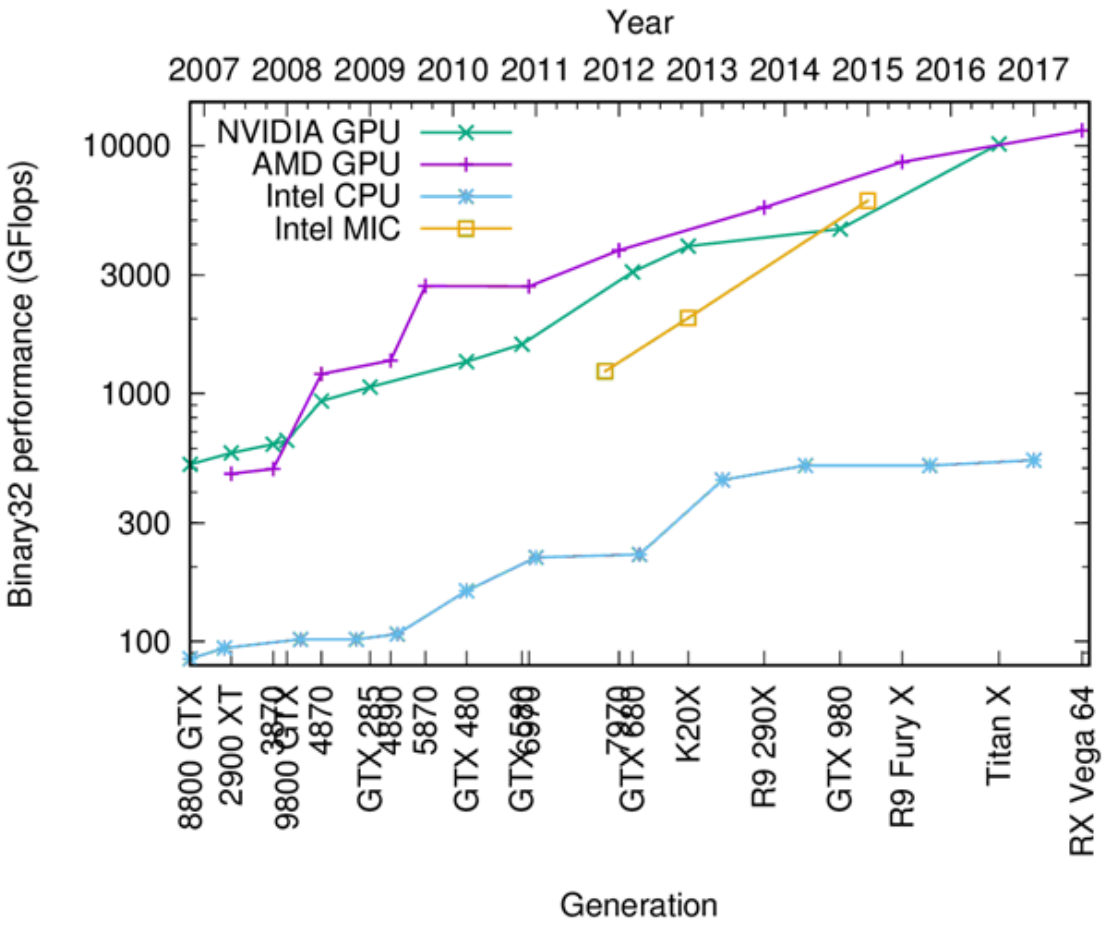


Fig. 5.1.: GPU CPU computational ability comparison

2. The designers for supercomputers and cloud service providers are using more GPU cards on each node. For example, the old Blue Waters (2007) cluster has 1 K20 GPU card + a 16-core AMD Opteron on each node, while the new Summit supercomputer (2018) has 6 Volta GPU card + a 42 cores IBM Power 9 per node – there are only 7 cores supporting each Volta! More aggressively, in the latest NVIDIA DGX-2 system, 2 Intel Xeon Platinum CPUs work together with 16 Volta GPUs – that is 3.5 CPU cores / GPU.

The rapid growth of GPU's ability forces us to realize: The balance between CPU and GPU computational ability continues to shift in favor of the GPU. The 1% workload from 10 years ago is now 60% or more in wall clock time. Any past code that has attempted to load balance work between CPU and GPU is today likely to be CPU bound. The lesson we

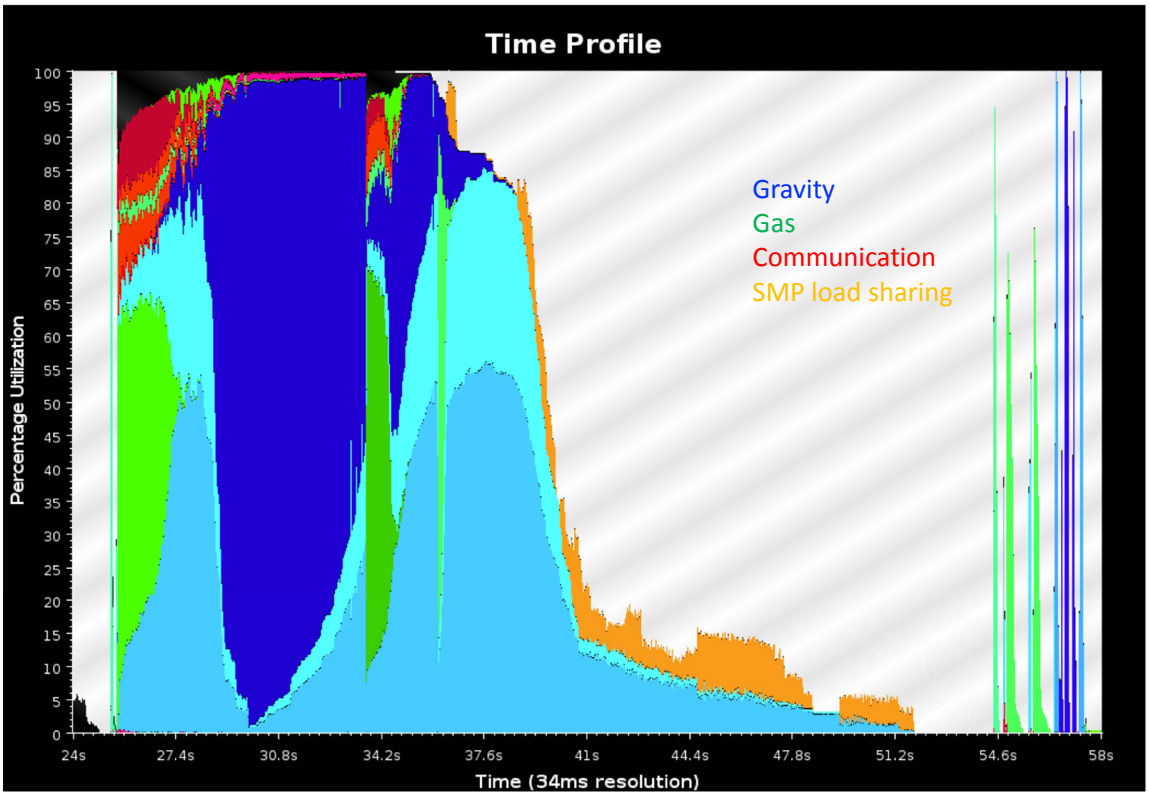


Fig. 5.2.: ChaNGa runtime profiling

learned from above is that we should apply GPU acceleration first to the most expensive part of the program.

Figure 5.2 shows the percentage utilization profiling of a typical ChaNGa simulation. The most expensive part is obviously the dark blue area, which means the local tree walk, and the secondary expensive, which is marked with light blue, is the remote tree walk. In this chapter, we will introduce the effort we have spent in offloading remote tree walk in ChaNGa to GPU.

5.1.1 Our approaches

Due to GPU's superior computational ability, we argue that for computation-intensive applications, such as ChaNGa, the GPU kernels should drive the design of the whole distributed parallel software. We want to offload the remote tree walk to GPU through two

approaches: 1) the GPU lead the process and conduct the CPU to prepare the data; 2) the CPU / GPU work independently but make sure the CPU always feed the data before GPU actually needs.

5.1.2 Challenges

In naïve CUDA programming model, the CPU takes charge of the memory copy from the host (CPU) to the device (GPU), kernel launch, CPU/GPU synchronization, and the memory copy back from the device to host. Before the GPU finishes its work, the CPU, in general, has to keep idle. For better performance, we need to extract the GPU management part out and assign to another process. The GPU manager process communicates with the computing processes in an asynchronous way.

The asynchronism becomes complicated as the number of cores/GPUs grows. Take one Summit node as an example, every 7 cores support one Volta card. To reach a better load balance (in current ChaNGa), we usually let each core processes multiple (e.g. 8) tree pieces. That means one GPU collaborate with 56 tree pieces and supports their local and remote tree walks. Although the GPU is super fast, its workload is also heavy. If we want the GPU to take heavier roles in the software, we have to carefully design the software to minimize the data latency.

In the distributed system, a single node usually doesn't have all the necessary data locally. The CPU should send the request to a remote node and receives the data. A hard question is: who decides which data to request and how to decide?

Besides, the remote tree walk is the core of ChaNGa. It connects with almost every module in the whole software. Offloading it to GPU means massive engineering effort.

5.2 Design

In the large scale n-body simulation, like ChaNGa, we distribute the whole global tree among all the nodes in the cluster. The CPU conducts the remote tree walk, builds interaction lists and sends to the GPU. When some remote data is needed, the CPU sends requests

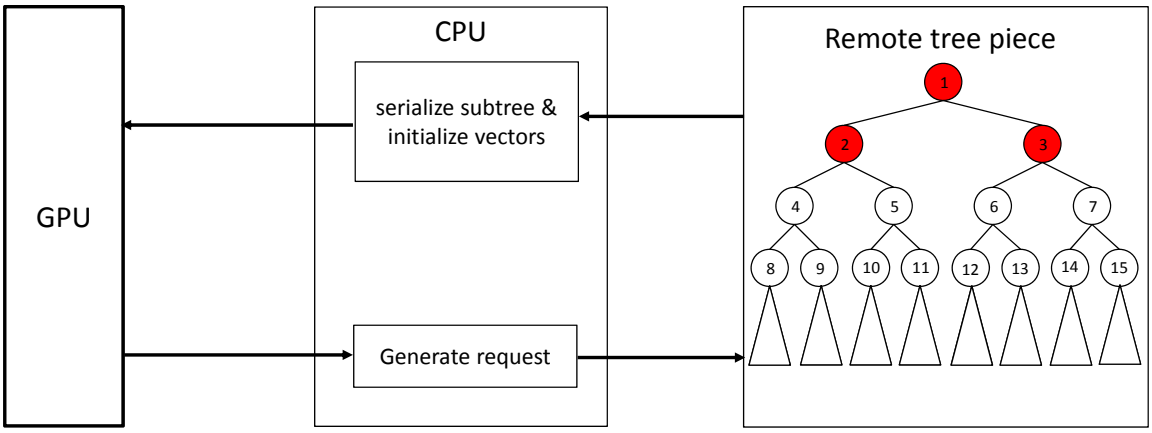


Fig. 5.3.: GPU-centric traversal. *nCacheDepth* equals 1.

to a remote tree piece. The remote tree piece replies with not only the requested tree node, but also a subtree with a certain depth (defined by the parameter "*nCacheDepth*".) under that node. When the remote tree walk is resumed on the requestor side, the CPU visits the subtree in a depth-first style. Since the subtree is stored in a software cache (on the requestor side), we'd like to call it "*cache subtree*" in the rest of the dissertation.

We want to explain a bit more about our GPU-centric and CPU/GPU collaboration approaches.

In the first approach (shown as Figure 5.3), once the CPU receives a cache subtree, it serializes data and sends to the GPU. Meanwhile, it also initializes a "leaves state array" whose size equals the number of leaf nodes in the cache subtree. Once the serialized data and array are sent to the GPU, the CPU resource switches to other tree pieces, or other tasks like the SPH. On the GPU side, the particles in the tree piece execute both remote tree walk and force calculation in parallel. If any particle wants to visit the subtree under a leaf node of current cache subtree, it sets the corresponding flag in the "leaves state array" – means "please send me that subtree next time". After GPU's work is done, the CPU scans the "leaves state array", and sends requests to remote tree pieces. This approach is a "GPU-centric" method – CPU only acts as a communication co-processor.

Figure 5.4 represents the second approach. The left tree is the local tree piece on one processor and the right tree is a typical cache subtree sent by a remote piece. Note that

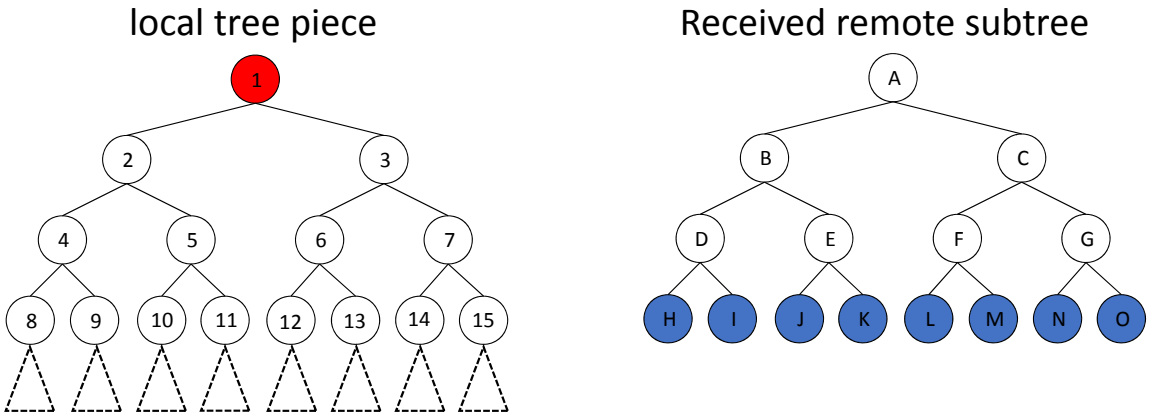


Fig. 5.4.: CPU approximated traversal

each leaf node on the left tree connects with a big subtree. The dual-tree walk in ChaNGa always descends along the local tree first, and most of the requests are generated by the leaf nodes (buckets). It is low efficient to launch the kernels by each request – the requests are very frequent while each request actually works for only a few buckets. Since the root's bounding box is the superset of its descendants', we assume such condition holds: if any leaf nodes in the local tree piece want to visit a remote tree node, the root of the local tree piece should want to do so as well.[3] Instead of descending along with the local tree piece, we treat the root as a "big" bucket and use the single tree walk. We admit that the root may send more requests than necessary, but that won't hurt performance badly. More aggressively, CPU doesn't even need to do the full single tree walk. It may directly perform the open criteria test with the leaf nodes of the received cache subtree – we call this as "approximated traversal". In Figure 5.4, it means the red node interacts with the blue nodes. The CPU only cares about which subtree to request, and the GPU handles the accurate tree walk + force computation. This approach is more efficient but may cause some CPU-GPU balance issue.

[3]The contrary statement doesn't hold because the union of its descendants' bounding box is typically smaller than the root's bounding box.

5.3 Implementation

The GPU-centric approach takes better usage of the GPU's computation ability and thus potentially promise better performance. However, it requires a big change to ChaNGa's architecture. Some of its features, like the CPU-GPU load balancing, even need support from Charm++ team. Considering the time management and implementation difficulty, we start from the second approach.

Each tree piece contains both its own information and the path to the root. We first let the CPU walks the global tree and send messages for missing tree nodes. The local tree nodes are skipped in this step. While the local tree piece waits for the remote data, the CPU serializes the local tree pieces and sends to the GPU. Once a cache subtrees is received, the CPU resumes the tree walk and sends out more messages. The CPU doesn't calculate gravity and use a simpler OpenCriterion for data fetching only. The CPU maintains a buffer for the received cache subtree and send it to the GPU when its size exceeds some threshold. To verify the correctness of this step, we still let the CPU walk the tree.

In the next step, we adopt the "approximated traversal". Given a cache subtree, the CPU only performs the open criterion test between the root and the leaf nodes of the cache subtree. The GPU conducts the accurate tree walk and produces the gravity calculation results. In this step, the CPU does not need to do the full prefetch walk. It skips $\sim 50\%$ OpenCriterion tests and just does array-based calculations (like GPU), so the CPU part is much faster. After this step is completed, we modify the CPU/GPU communication model in ChaNGa and switch to the "GPU-centric" approach.

5.4 Discussion

We thought we had taken everything into consideration and the GPU remote walk should work perfectly as the GPU local tree walk. Unfortunately, we made an incorrect assumption and could not reach a balance between accuracy and performance. We will talk about the problems we met and the efforts we had made in this section.

5.4.1 Open criterion and inclusion condition

Let's start with the *open criterion* because we need serval derivative concepts in the following discussion. The open criterion is the key factor of tree traversal. It decides whether a particle or a group of particles, continue to traverse part of the tree or not. ChaNGa adopts sophisticated steps in open criterion test to satisfy convergence, symmetry and achieve high accuracy. Figure 5.5 provides a simplified version for this chapter's discussion. Beyond this version, ChaNGa's original test also handles multipole expansion and gravitational softening. We point out that a node in the tree represents a bounding box in the space. They are interchangeable in this dissertation.

The open criterion function checks whether any part of the bucket node's bounding box overlaps with the **open circle** of the target node. The open circle is a circle rooted at the center of the target node's center, with radius equals the radius of the node's excircle divided by the *gFactor*. The input parameter *gFactor* is a pre-defined constant and typically smaller than 1.0 [4]. Figure 5.6 shows an example of tree walk. The bucket node should truncate at Node C but continue to traverse Node A and Node B.

We define the inclusion condition as this:

Definition 5.4.1 *If one bucket truncates its traversal at a parent node, it should truncate at the descendant nodes as well, or the continuing of traversal shouldn't affect the result.*

If one traversal satisfies the inclusion condition, our previous design is able to guarantee the correctness and deliver a good performance.

5.4.2 The break of inclusion condition

In our experiment, we realize that the inclusion condition we mentioned above doesn't hold in ChaNGa. We want to show the break of the inclusion condition through a series of figures. An octree is hard to draw. Let's consider 2D plane and quad-tree instead. And we set *gFactor* as 1.0 for simplicity – the open circle of a node reduces to its excircle.

[4]The *gFactor* is an expression of *theta*.

```
1   Bool openCriterion(Node& target, Node& bkt, double gFactor) {
2       double openRadius = target.radius / gFactor;
3       vector3D<double> s = node.center;
4       double rsq = openRadius * openRadius;
5       double delta;
6       if((delta = bkt.lesser_corner.x - s.x) > 0)
7           dsq += delta * delta;
8       else if((delta = s.x - bkt.greater_corner.x) > 0)
9           dsq += delta * delta;
10      if(rsq < dsq)
11          return false;
12      if((delta = bkt.lesser_corner.y - s.y) > 0)
13          dsq += delta * delta;
14      else if((delta = s.y - bkt.greater_corner.y) > 0)
15          dsq += delta * delta;
16      if(rsq < dsq)
17          return false;
18      if((delta = bkt.lesser_corner.z - s.z) > 0)
19          dsq += delta * delta;
20      else if((delta = s.z - bkt.greater_corner.z) > 0)
21          dsq += delta * delta;
22      return dsq <= rsq;
23  }
```

Fig. 5.5.: Simplified open criterion

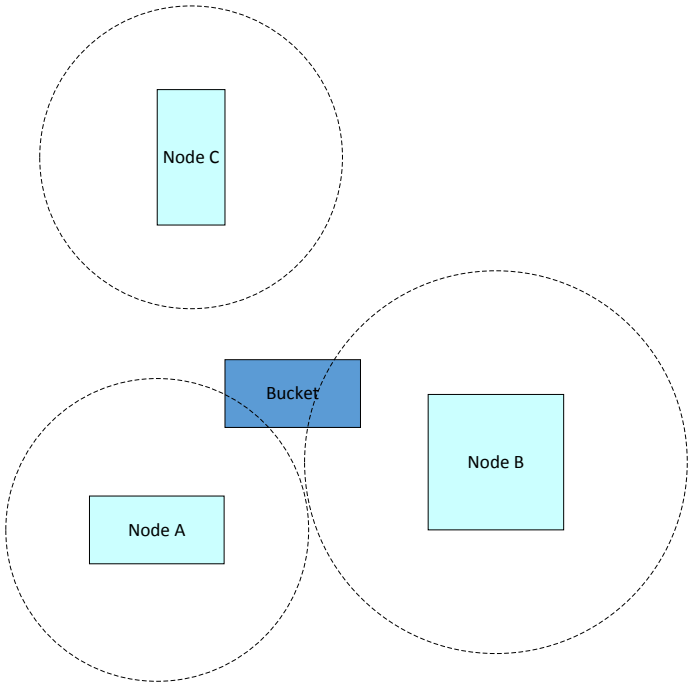


Fig. 5.6.: Open radius in 2D plane

ChaNGa builds the spatial tree in a bottom-up way. We usually start with the lowest level bounding box then grow up. Suppose we have a bounding box, which is a square with edge length equals b. Its open radius is $\frac{\sqrt{2}}{2}b$. After we finish another bounding box along the y-axis, these two squares form an upper-level bounding box (a rectangle). The center of its open circle falls on the border of the two squares and its open radius equals $\frac{\sqrt{5}}{2}b$. The "Area 1" is the space that belongs to C1 but is not covered by the C2 (Figure 5.7).

The Area 1 causes critical issues in our remote tree walk. Considering that our policy is that *if any bucket of a tree piece wants to visit any slice of another remote tree, the CPU will send out requests and eventually get that slice back. Then CPU will pass the slice to the GPU so that* ***every bucket*** *can check whether the slice is necessary.* Suppose we are executing remote tree walk on another tree piece. There are two buckets A and B. Bucket A overlaps with both C1 and C2 – it will trigger two requests (asking for the rectangle and the top square separately, suppose the cache depth is one). Bucket B only overlaps with C1. When the message containing the rectangle arrives the CPU, both A and B perform the open criteria test. Bucket A decides to "KEEP" (wait for the top square in the remote tree piece, no actual force calculation happens). The bucket B fails the open criteria test with the rectangle so that it doesn't need to continue the traversal (action == "COMPUTE"). The particles under bucket B directly compute with the tree node representing the rectangle. However, when the message encapsulating the top square comes, the bucket B passes the open criteria test with the top square and its particles need to compute with each particle within the square. Note that the tree node representing the rectangle contains multipole moments of the mass distribution of the particles within the whole rectangle. The mass under the square actually is computed **twice**. Similarly, when we use two rectangles to form an upper-level square, the "Area 2" could be another problem. In standard ChaNGa accuracy test, these areas result in 0.4% *repeated* particle-cell and 2.2% *repeated* particle-particle interaction list computation.

We want to emphasize that the break of inclusion condition is the characteristic of ChaNGa(Barnes-Hut)'s implementation. For other applications, such as Point Correlation

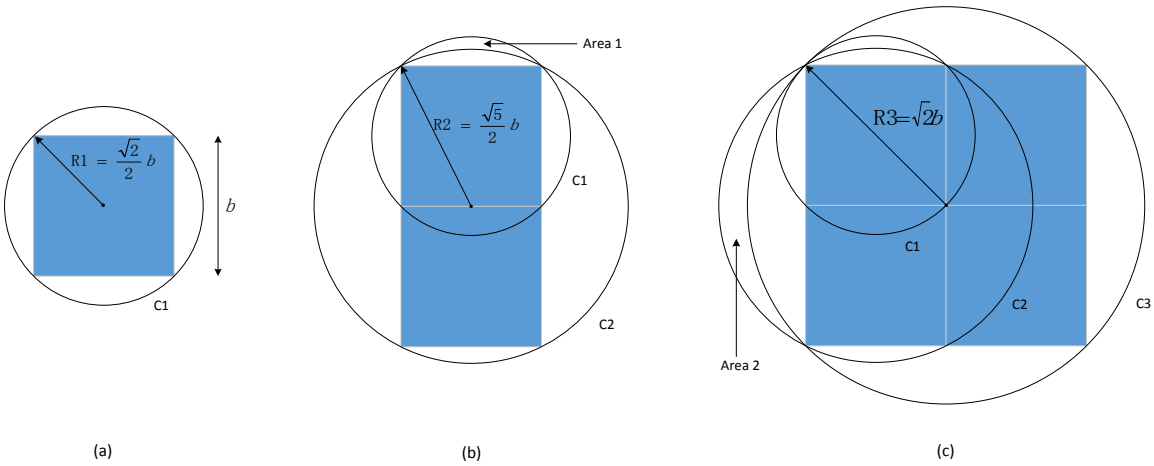


Fig. 5.7.: Original Problem

and k-Nearest Neighbor search, repeated computation doesn't affect the result, thus the inclusion condition is naturally satisfied.

We have developed several methods to fix the break of inclusion condition but none of them reaches a good balance between accuracy and performance.

5.4.3 Approach 1: bottom-up open radius calculation

The current ChaNGa calculates the *open radius* directly from the bounding box. The *open radius* is defined as the Euclidean length of the vector consisting of maximum distances from anywhere in the bounding box to its center along each axis. A parent node's open radius is independent of its descendants, thus a parent node's circle doesn't completely cover its descendants' circle. We want to restore the inclusion condition through different open radius calculation method.

Our first approach is to integrate the radius calculation into the bottom-up tree construction process. The parent node's radius is defined as the maximum of the sum of the distance to its child and that child's radius:

$$Parent.radius = max(distance(parent, child) + child.radius) \tag{5.1}$$

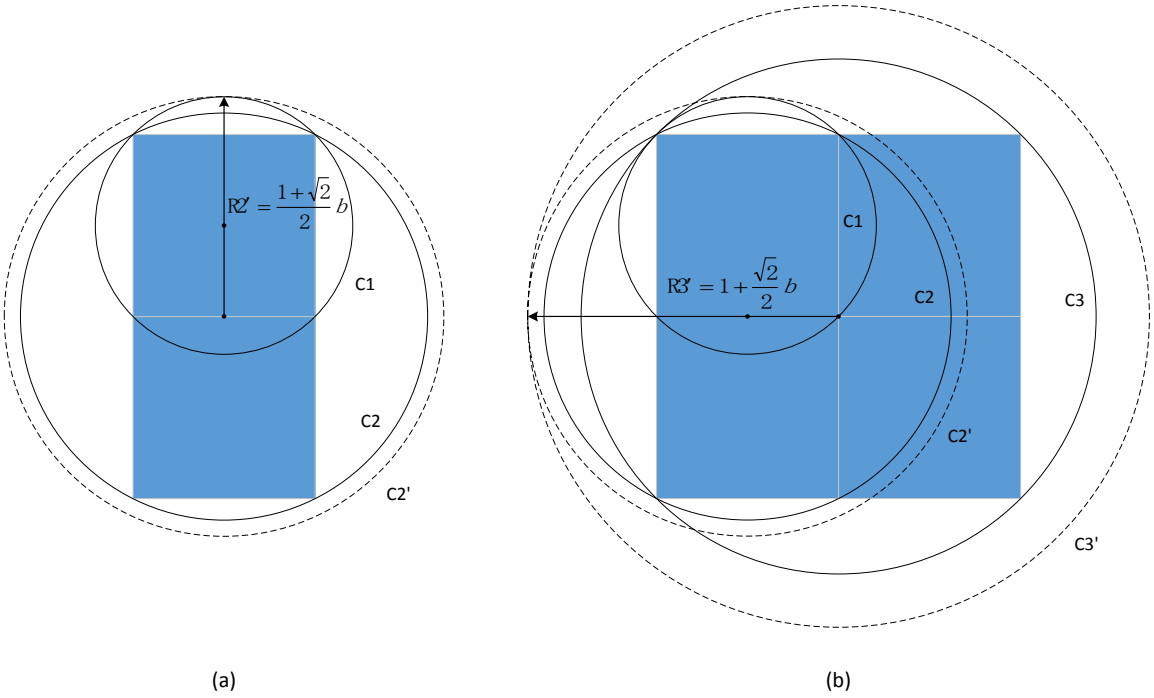


Fig. 5.8.: Bottom up radius calculation

In Figure 5.8 (a), the radius of $C2'$ ($R2'$) equals to the distance from E1's center to $C2'$'s center, plus C1's radius. The C2' circle apparently covers all the field of C1. Similarly, the C3' circle covers all the space of the C2' and C2. This approach satisfies the inclusion condition and guarantees the correctness. However, it brings up the performance issue.

According to Equation 5.1, the open radius of the outer square ($R3'$) should be the distance from the center of $C3'$ to the center of $C2'$, plus the radius of $C2'$ ($R2'$). Note that we have already added an increment from R2 to ($R2'$), the same increment has been propagated to ($R3'$) as well:

$$R2' = 1.07 \times R2$$

$$R3' = 1.2 \times R3 = 1.12 \times (1.07 \times R3)$$

The growth of the incremental factor is exponential. As the larger open radius results in the traversal opening more nodes, the total amount of computation explodes rapidly. Even though the GPU remote tree walks shows some speedup with small input data set (30K particles), its advantage could be easily eaten by the extra computation. Our experiment

results show that the approach runs 70× slower than the baseline for the ***dwf1.6144*** input file.

5.4.4 Approach 2: top-down open radius calculation

A natural question is that: can we eliminate the exponential growth of the open radius and the inclusion problem in one shot? We thought we could. Our second approach aims at a top-down open radius calculation. In the 2-D plane, we notice that the excircle of the outer square automatically covers "Area 1" – the calculation of the outer square's open radius only needs to consider "Area 2". In other words, we just need to stretch one node's open radius according to its children, not its grandchildren or further descendants. Figure 5.9 shows the new excircle of outer square ($C3''$), and the comparison to ($C3'$). The radius grows linearly, instead of exponentially, in the top-down approach.

A key factor here is the definition of the "center" of the bounding box. This approach works under the assumption that each bounding box's open circle rooted at its geometric center. Unfortunately, ChaNGa's force calculation equation is built atop the center of mass, instead of geometric center. "To minimize the maximum error due to the multipole expansion, it is best to use the center of the bounding sphere of the particles in the local volume, but for symmetry reasons, the ChaNGa decides to use the center of the mass of the particles within the bounding box" [52]. The center of a bounding box's open circle could be anywhere in the box. In Figure 5.10, the center of C1 falls onto the top left corner of the top left square so that the calculated $C3''$ can only cover part of C1's space. Similar to the original design, the top-down approach simply fails the accuracy test.

5.4.5 Approach 3: bookkeeping

Another approach is to do the bookkeeping work: for each bucket, we record which remote node (and its descendants) it needs to traverse. The first challenge is: where and how to store the bookkeeping information – its size could be quite large.

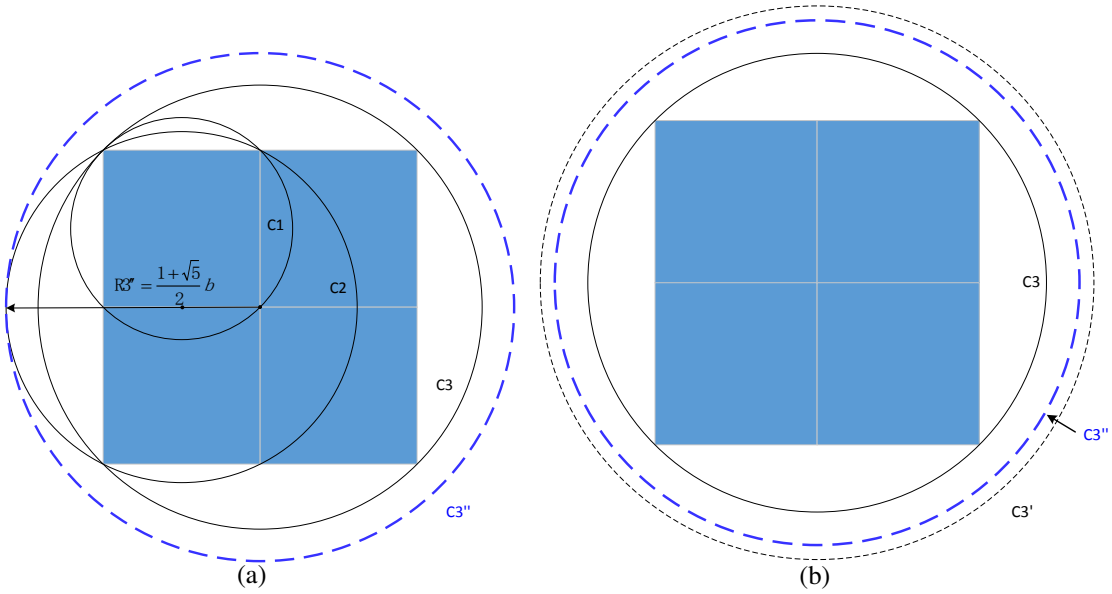


Fig. 5.9.: Top-down radius calculation

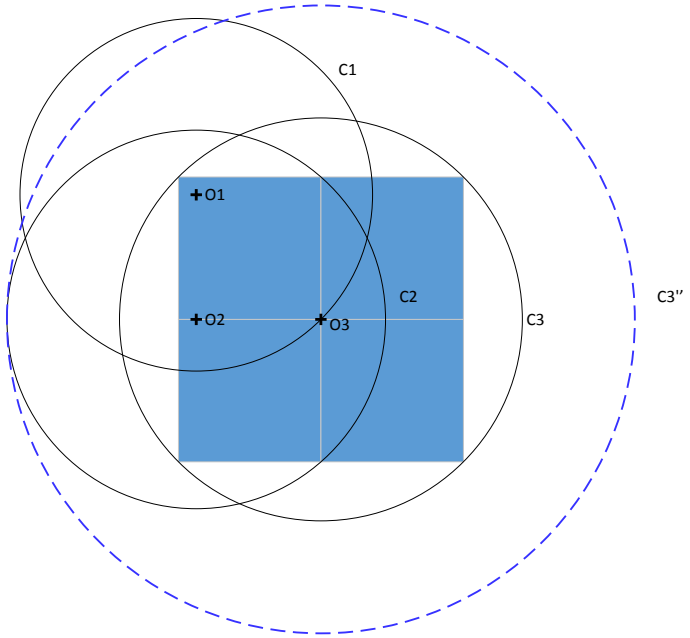


Fig. 5.10.: CornerCase

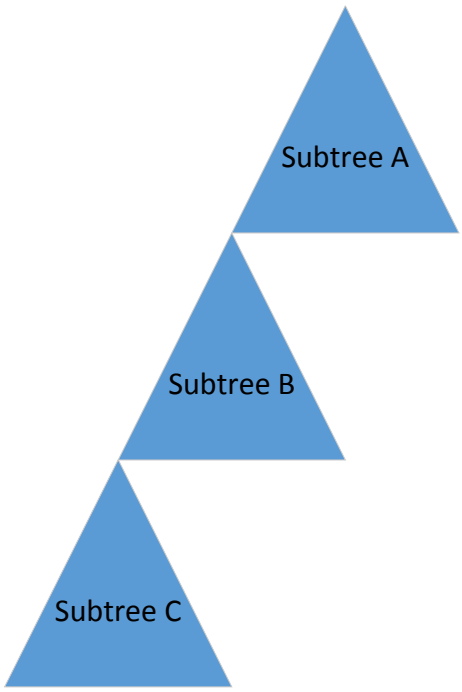


Fig. 5.11.: Dependency among subtrees

A straightforward idea is to store as a boolean matrix. Each row represents a bucket node in the local tree piece and each column corresponds to a remote tree node. The value indicates whether the bucket node needs to visit that remote node or not. Let's take the ***dwf1.6144*** (containing 80M particles) as example. Suppose we split the whole global tree into 8 pieces and distribute them among 8 machines, then each tree piece has 10M particles. And assume the bucket size is 32, so every tree piece has 312.5K buckets (the number of rows). If we use one bit to represent one bucket, the number of columns is around 312.5K / 8 ≈ 39K. For the worst case, each tree piece wants to visit the whole global tree and we need to pre-allocate 39K * 7 * 312.5K ≈ 85GB space (rough estimation). Given that GPU usually has limited memory (typically <=4GB), it is impractical to directly store such a matrix on GPU.

If we store such information in the CPU, the data transfer between CPU and GPU is inevitable. Given a cache subtree, we need to send a vector indicating which buckets need to visit it and copy back a matrix containing which buckets need to visit which descendant subtrees. This process causes dependency and latency issues.

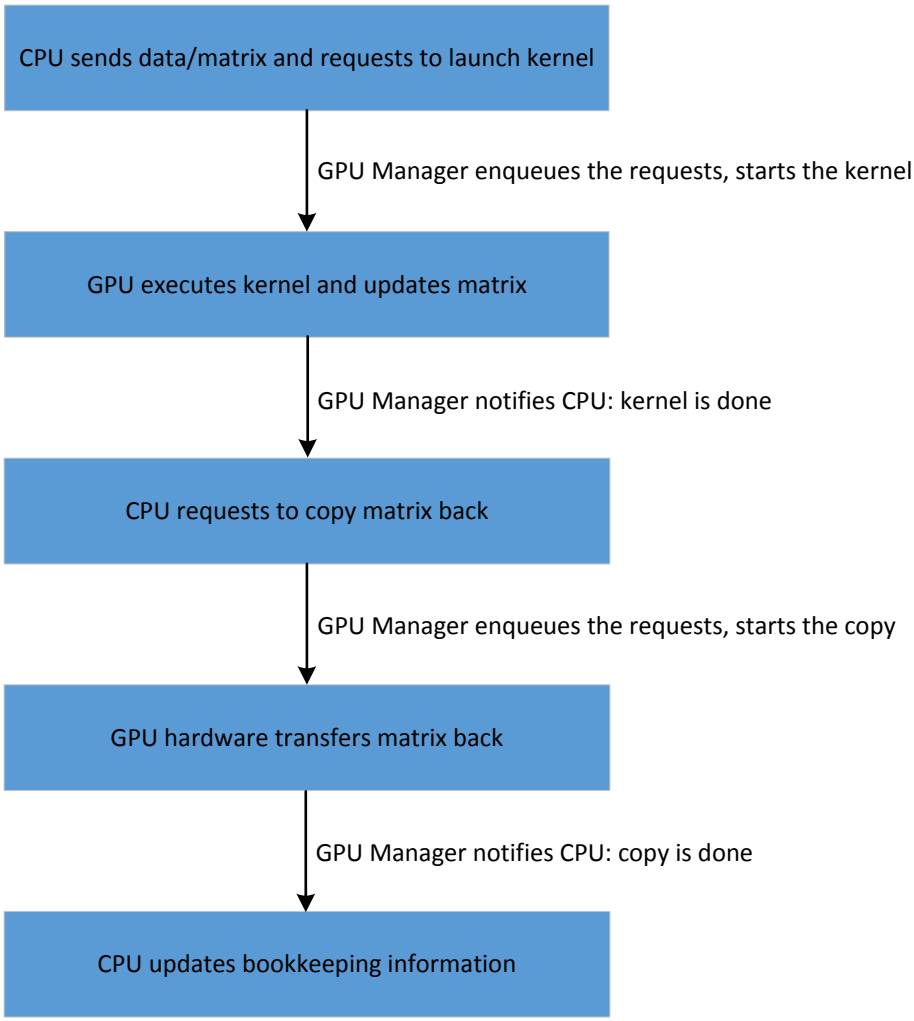


Fig. 5.12.: CPU/GPU asynchronous communication workflow

Figure 5.11 shows three cache subtrees: subtree B is a descendant of subtree A, and subtree C is a descendant of B. In original ChaNGa, these three subtrees are received in separate messages, but very likely to be packaged together and sent to GPU in one data transfer (together with interaction lists). In our GPU remote tree walk, we can only know which buckets want to visit subtree B after GPU finish the traversal of A – the subtree B should be sent after subtree A has been processed. These three subtrees have to be sent in THREE separate data transfer in sequence. For one thing, we are not able to overlap the GPU computation time with the data transfer; for the other thing, there is a latency issue among subtrees due to ChaNGa's asynchronous communication.

Figure 5.12 shows the whole procedures for one cache subtree. After the CPU receives a cache subtree from remote machines, it sends the data to the GPU manager and requests to launch the GPU kernel. The data includes the serialized cache subtree, and a vector indicating which buckets need to visit this subtree to the GPU manager. The GPU manager enqueues the data/request (FIFO) and starts the kernel when the GPU hardware is available. As the kernel is done, the GPU manager notifies the CPU through a callback function. Then the CPU sends another request to get the new bookkeeping matrix back (which buckets need to visit which descendant subtrees). The GPU manager again enqueues the request and eventually finish the data transfer. At the last step, the CPU updates its bookkeeping matrix. Due to asynchronous communication, the whole process could be interrupted by other tasks and seriously delayed. This problem could be alleviated by larger *nCacheDepth*, but *nCacheDepth* is a key parameter for the whole system. Large *nCacheDepth* value may cause load balancing problem.

Let's look back to the original ChaNGa. The CPU performs tree walks and generates the interaction lists. Once the lists are built, the CPU offloads them to the GPU manager and switches to other tasks. The CPU is not blocked while the GPU is working. There is neither dependency nor latency issue. The GPU doesn't need to copy the result back until the end of the whole computation. And we overlap the GPU computation time with the CPU/GPU data transfer, and the CPU tree walks with the remote communication. Based on these, we're afraid the bookkeeping method may not be able to deliver enough speed up, at least not so much as GPU local tree walk.

We don't discuss other data structure like hashmap on CPU side because 1) for the worst case, hashmap takes even more space than matrix, 2) we have to translate the hashtable into arrays during CPU/GPU communication, which will take extra time.

5.5 Conclusions

As GPU's performance is getting stronger and stronger, it has become a general trend to use more and more powerful GPU in supercomputers. Thus offloading more computation

tasks to GPU is necessary and urgent. We have shown that GPU is able to execute local tree walk very efficiently. However, due to ChaNGa's specific system design, we are not able to migrate remote tree walk to GPU in high efficiency. We have tried several methods and analyzed why they do not work. The lack of satisfaction to the inclusion condition makes the remote tree walk offloading extremely hard. We strongly recommend ChaNGa team to change their implementation to meet the inclusion condition and improve the CPU/GPU asynchronous communication efficiency. We emphasize that our design should work well with other applications like Point Correlation, k-Nearest Neighbor search, and Smooth Particle Hydrodynamics, etc.

6. CONCLUSIONS

This dissertation is tackling optimizations for n-body problems on heterogeneous systems. We investigated performance improvement on both single node and distributed heterogeneous systems.

6.1 Single node heterogenous system

We first formulated the general scheduling algorithm and showed that the scheduling for the n-body problem on GPU is NP-hard. Then we came up with a hybrid, inspector-executor based, dynamic scheduling framework. Our framework is able to perform dynamic scheduling in a general, application-agnostic and input-independent manner. We also developed different optimized profiling and scheduling algorithms that exploit the structural properties of traversal algorithms. Last but not least, we developed a new skeleton for writing GPU kernel portion of the n-body problem that minimize unnecessary memory accesses.

On a single node heterogenous system, we demonstrated that for six benchmarks, our hybrid scheduling framework plus better GPU kernels can deliver 5.96$times$ (on average) speedup over original GPU kernel with unsorted inputs. More interestingly, our framework is 1.41$times$ faster than our GPU kernels with application-specific sorted inputs.

6.2 Distributed heterogenous system

On a state-of-the-art distributed heterogenous n-body simulation platform, we spent effort on both local walk, in which bodies on a single node only visit other bodies within the same node, and remote walk, where bodies interact with others on different nodes.

On local walk part, we observed that the asymptotic advantage of the dual-tree walk was effective for CPU computation, but was counterbalanced by the requirements of parallelism and regular control flow for GPU computation. By using an efficient single-tree algorithm, we moved both tree walk and force calculation to the GPU. We took advantage of GPU's powerful parallelism to eliminate the CPU and communication bottleneck. We showed that our approach was able to overcome the higher asymptotic complexity of the single tree algorithm. Across several benchmarks, our implementation is 8.25*times* faster on average than original ChaNGa on a single node configuration, and 1.8*times* on a multiple nodes configuration.

On remote walk part, we demonstrated that the break of inclusion condition prevented straightforward GPU remote tree walk. We explained how the problem happens and why several approaches didn't work well. We pointed out that it was necessary to change the calculation of the bounding box center to satisfy the inclusion condition. Our design should work well with other applications like Point Correlation, k-Nearest Neighbor search, and Smooth Particle Hydrodynamics.

REFERENCES

REFERENCES

[1] J. Barnes and P. Hut, "A hierarchical o(n log n) force-calculation algorithm," *nature*, vol. 324, p. 4, 1986.

[2] P. J. E. Peebles, *The large-scale structure of the universe*, 1980.

[3] N. Kumar, L. Zhang, and S. Nayar, "What is a good nearest neighbors algorithm for finding similar patches in images?" in *European conference on computer vision*. Springer, 2008, pp. 364–378.

[4] M. Burtscher and K. Pingali, "An efficient CUDA implementation of the tree-based barnes hut n-body algorithm," in *GPU Computing Gems Emerald Edition*. Elsevier Inc., 2011, pp. 75–92.

[5] T. Foley and J. Sugerman, "Kd-tree acceleration structures for a gpu raytracer," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ser. HWWS '05, 2005, pp. 15–22. [Online]. Available: http://doi.acm.org/10.1145/1071866.1071869

[6] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek, "Realtime ray tracing on gpu with bvh-based packet traversal," in *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, ser. RT '07, 2007, pp. 113–118. [Online]. Available: http://dx.doi.org/10.1109/RT.2007.4342598

[7] M. Hapala, T. Davidovic, I. Wald, V. Havran, and P. Slusallek, "Efficient Stack-less BVH Traversal for Ray Tracing," in *Proceedings 27th Spring Conference of Computer Graphics (SCCG) 2011*, 2011, pp. 29–34.

[8] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-d tree gpu raytracing," in *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ser. I3D '07, 2007, pp. 167–174.

[9] M. Méndez-Lojo, M. Burtscher, and K. Pingali, "A gpu implementation of inclusion-based points-to analysis," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. ACM, 2012, pp. 107–116.

[10] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012, pp. 117–128.

[11] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek, "Stackless kd-tree traversal for high performance GPU ray tracing," *Computer Graphics Forum*, vol. 26, no. 3, pp. 415–424, Sep. 2007, (Proceedings of Eurographics).

[12] M. Goldfarb, Y. Jo, and M. Kulkarni, "General transformations for gpu execution of tree traversals," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 10:1–10:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503223

[13] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, pp. 509–517, September 1975. [Online]. Available: http://doi.acm.org/10.1145/361002.361007

[14] M. S. Warren and J. K. Salmon, "A parallel hashed oct-tree n-body algorithm," in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '93. New York, NY, USA: ACM, 1993, pp. 12–21. [Online]. Available: http://doi.acm.org/10.1145/169627.169640

[15] J. G. Stadel, "Cosmological N-body Simulations and their Analysis," Ph.D. dissertation, Department of Astronomy, University of Washington, March 2001.

[16] V. Springel, "The cosmological simulation code GADGET-2," *Monthly Notices of the Royal Astronomical Society*, vol. 364, pp. 1105–1134, Dec. 2005.

[17] H. Menon, L. Wesolowski, G. Zheng, P. Jetley, L. Kale, T. Quinn, and F. Governato, "Adaptive techniques for clustered n-body cosmological simulations," *Computational Astrophysics and Cosmology*, vol. 2, no. 1, p. 1, 2015.

[18] L. V. Kale and S. Krishnan, "Charm++: Parallel Programming with Message-Driven Objects," in *Parallel Programming using C++*, G. V. Wilson and P. Lu, Eds. MIT Press, 1996, pp. 175–213.

[19] W. Dehnen, "A Hierarchical *O*(N) Force Calculation Algorithm," *Journal of Computational Physics*, vol. 179, pp. 27–42, Jun. 2002.

[20] D. Potter, J. Stadel, and R. Teyssier, "PKDGRAV3: beyond trillion particle cosmological simulations for the next era of galaxy surveys," *Computational Astrophysics and Cosmology*, vol. 4, p. 2, May 2017.

[21] X. Zhang and A. A. Chien, "Dynamic pointer alignment: Tiling and communication optimizations for parallel pointer-based computations," in *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '97. New York, NY, USA: ACM, 1997, pp. 37–47. [Online]. Available: http://doi.acm.org/10.1145/263764.263771

[22] V. K. Pingali, S. A. McKee, W. C. Hseih, and J. B. Carter, "Computation regrouping: Restructuring programs for temporal data cache locality," in *Proceedings of the 16th International Conference on Supercomputing*, ser. ICS '02. New York, NY, USA: ACM, 2002, pp. 252–261. [Online]. Available: http://doi.acm.org/10.1145/514191.514227

[23] Y. Jo and M. Kulkarni, "Automatically enhancing locality for tree traversals with traversal splicing," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '12. New York, NY, USA: ACM, 2012, pp. 355–374. [Online]. Available: http://doi.acm.org/10.1145/2384616.2384643

[24] Y. Jo, M. Goldfarb, and M. Kulkarni, "Automatic vectorization of tree traversals," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, ser. PACT'13. IEEE, 2013, pp. 363–374.

[25] J. Liu, N. Hegde, and M. Kulkarni, "Hybrid cpu-gpu scheduling and execution of tree traversals," in *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2016, p. 2.

[26] M. Amor, F. Argüello, J. López, O. G. Plata, and E. L. Zapata, "A data parallel formulation of the barnes-hut method for n -body simulations," in *Proceedings of the 5th International Workshop on Applied Parallel Computing, New Paradigms for HPC in Industry and Academia*, 2001, pp. 342–349. [Online]. Available: http://portal.acm.org/citation.cfm?id=645782.666840

[27] H. Han and C.-W. Tseng, "Exploiting locality for irregular scientific codes," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, pp. 606–618, July 2006. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2006.88

[28] J. K. Salmon, "Parallel hierarchical n-body methods," Ph.D. dissertation, California Institute of Technology, 1991.

[29] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy, "Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity," *J. Parallel Distrib. Comput.*, vol. 27, no. 2, pp. 118–141, 1995.

[30] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan, "Rendering complex scenes with memory-coherent ray tracing," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, 1997, pp. 101–108. [Online]. Available: http://dx.doi.org/10.1145/258734.258791

[31] P. A. Navratil, D. S. Fussell, C. Lin, and W. R. Mark, "Dynamic ray scheduling to improve ray coherence and bandwidth utilization," in *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, ser. RT '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 95–104. [Online]. Available: http://dx.doi.org/10.1109/RT.2007.4342596

[32] P. A. Navratil, "Memory-efficient, scalable ray tracing," Ph.D. dissertation, 2010.

[33] T. Aila and T. Karras, "Architecture considerations for tracing incoherent rays," in *Proceedings of the Conference on High Performance Graphics*, ser. HPG '10. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2010, pp. 113–122. [Online]. Available: http://dl.acm.org/citation.cfm?id=1921479.1921497

[34] B. Moon, Y. Byun, T.-J. Kim, P. Claudio, H.-S. Kim, Y.-J. Ban, S. W. Nam, and S.-E. Yoon, "Cache-oblivious ray reordering," *ACM Trans. Graph.*, vol. 29, no. 3, pp. 28:1–28:10, Jul. 2010. [Online]. Available: http://doi.acm.org/10.1145/1805964.1805972

[35] J. H. Saltz and R. Mirchandaney, "Run-time parallelization and scheduling of loops," in *IEEE Transactions on Computers*, vol. 40. IEEE, May 1991, pp. 603–612.

[36] R. Wu, B. Zhang, and M. Hsu, "Clustering billions of data points using gpus," in *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*. ACM, May 2009, pp. 1–6.

[37] V. T. Ravi, W. Ma, , and G. Agrawal, "Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations," in *Proceeding of the 24th ACM International Conference on Supercomputing*. ACM, June 2010, pp. 137–146.

[38] V. T. Ravi and G. Agrawal, "A dynamic scheduling framework for emerging heterogeneous systems," in *Proceedings of the 2011 18th International Conference on High Performance Computing*. IEEE Computer Society Washington, DC, USA, 2011, pp. 1–10.

[39] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for gpu computing," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. ACM New York, NY, USA ©2011, March 2011, pp. 369–380.

[40] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, December 2009, pp. 45–55.

[41] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali, "Adaptive heterogeneous scheduling for integrated gpus," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM New York, NY, USA, August 2014, pp. 151–162.

[42] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn, "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*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: https://doi.org/10.1109/SC.2010.49

[43] L. V. Kale and A. Bhatele, *Parallel science and engineering applications: The Charm++ approach*. CRC Press, 2016.

[44] D. Reed, J. Gardner, T. Quinn, J. Stadel, M. Fardal, G. Lake, and F. Governato, "Evolution of the mass function of dark matter haloes," *MNRAS*, vol. 346, pp. 565–572, Dec. 2003.

[45] C. Power, J. Navarro, A. Jenkins, C. Frenk, S. D. White, V. Springel, J. Stadel, and T. Quinn, "The inner structure of λcdm haloesi. a numerical convergence study," *Monthly Notices of the Royal Astronomical Society*, vol. 338, no. 1, pp. 14–34, 2003.

[46] D. Reed, J. Gardner, T. Quinn, J. Stadel, M. Fardal, G. Lake, and F. Governato, "Evolution of the mass function of dark matter haloes," *Monthly Notices of the Royal Astronomical Society*, vol. 346, no. 2, pp. 565–572, 2003.

[47] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel, "Buffer kd trees: processing massive nearest neighbor queries on gpus," in *International Conference on Machine Learning*, 2014, pp. 172–180.

[48] N. Hegde, J. Liu, and M. Kulkarni, "Spirit: a framework for creating distributed recursive tree applications," in *Proceedings of the International Conference on Supercomputing*. ACM, 2017, p. 3.

[49] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[50] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[51] (2017). [Online]. Available: http://www.irisa.fr/alf/downloads/collange/cours/opt2018/opt2018_gpu_1.pdf

[52] J. G. Stadel, "Cosmological N-body simulations and their analysis," Ph.D. dissertation, UNIVERSITY OF WASHINGTON, 2001.

VITA

VITA

Jianqiao Liu was born in Huangchuan, China. He got his bachelor degree in Automatic Control from Harbin Engineering University, China, 2010. After that he joined Shanghai Jiao Tong University (SJTU), and got his master degree in Control Science and Engineering in 2013. At almost the same time, he also got a second master degree in Electrical and Computer Engineering from Georgia Institute of Technology (GIT) through the dual-master program between SJTU and GIT. In 2014 spring, he started to pursue his doctoral degree under the guidance of professor Milind Kulkarni in Purdue University. His current research focus is mainly about optimization for irregular applications on hetergeneous system and large scale system.