RT-BarnesHut: Accelerating Barnes–Hut Using Ray-Tracing Hardware

Vani Nagarajan
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
nagara16@purdue.edu

Rohan Gangaraju
Department of Computer Science
University of Texas at Austin
Austin, TX, USA
rgangar@utexas.edu

Kirshanthan Sundararajah
Department of Computer Science
Virginia Tech
Blacksburg, VA, USA
kirshanthans@vt.edu

Artem Pelenitsyn
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
apelenit@purdue.edu

Milind Kulkarni
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
milind@purdue.edu

Abstract

The n-body problem involves calculating the effect of bodies on each other. n-body simulations are ubiquitous in the fields of physics and astronomy and notoriously computationally expensive. The naïve algorithm for n-body simulations has the prohibiting $O(n^2)$ time complexity. Reducing the time complexity to $O(n \cdot \lg(n))$, the tree-based Barnes–Hut algorithm approximates the effect of bodies beyond a certain threshold distance. Other than algorithmic improvements, extensive research has gone into accelerating n-body simulations on GPUs and multi-core systems. However, Barnes–Hut is a tree-traversal algorithm, which makes it a poor target for acceleration using traditional GPU shader cores. In contrast, recent work shows that, for tree-based computations, GPU ray-tracing (RT) cores dominate shader cores. In this work, we reformulate the Barnes–Hut algorithm as a ray-tracing problem and implement it with NVIDIA OptiX. Our evaluation shows that the resulting system, RT-BarnesHut, outperforms current state-of-the-art GPU-based implementations.

We thank the anonymous PPoPP reviewers and our shepherd for their valuable feedback. We thank Dr. Tom Quinn for answering our questions about ChaNGa. This work was funded by NSF grants CCF-1908504, CCF-1919197 and CCF-2216978.

PPoPP '25, March 1–5, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1443-6/25/03
https://doi.org/10.1145/3710848.3710885

CCS Concepts: • **Computing methodologies** → **Ray tracing**; **Massively parallel and high-performance simulations**; **Massively parallel algorithms**.

Keywords: GPU, RT-Core, Barnes-Hut, Tree Traversal, Ray Tracing, accelerators

1 Introduction

Barnes–Hut is an approximation technique used to compute forces in an n-body simulation [1]. The algorithm is widely used in the fields of computational astrophysics, computer graphics and astronomy [9, 22, 24]. Barnes–Hut approximates the force exerted on a body by other bodies at a longer range and reduces the complexity of the n-body problem from $O(n^2)$ to $O(n \cdot lg(n))$ without a significant decrease in accuracy. The algorithm's central idea employs a spatial tree (quadtree or octree) that groups spatially close bodies. An internal node of the tree stores the center of mass of the node's descendants and allows to estimate the effect of those. The estimation applies when the underlying bodies are sufficiently far away, as defined by a given threshold.

As n-body simulations are computationally intensive and typically process large volumes of data, researchers have proposed ways of accelerating Barnes–Hut on multi-core systems and GPUs [2, 10, 12]. As the force calculation for each query body is independent, it would seem that Barnes–Hut computation maps well onto GPUs as they are well-known for accelerating massively parallel workloads. However, Barnes–Hut computation is an *irregular* workload as it involves traversing the Barnes–Hut tree and the traversal order of nodes differs for each query body. This irregularity in control flow and memory accesses adversely affects the performance of Barnes–Hut computation on a GPU.

The introduction of Ray Tracing (RT) cores in recent GPUs has provided users with a new way to hardware-accelerate some irregular workloads on GPUs. The original purpose

of the RT cores is to accelerate ray tracing workloads. *Ray tracing* is the process of following the rays that were cast through the image plane and testing whether they intersect with objects in a scene to produce an image. Since this would require each ray to be tested against every object in the scene to record all the interactions, Bounding Volume Hierarchies (BVH) are used to reduce the *number* of ray-object intersection tests. The BVH is a spatial tree structure that recursively groups objects together using bounding volumes so that ray-object intersection tests now become ray-bounding-volume intersection tests. RT cores accelerate the traversal of BVH and ray-object intersection tests if the objects are triangles.

RT cores can be thought of as hardware that accelerates a *particular type* of tree traversals. Prior works have proposed reductions of several problems that use a spatial tree optimization to the ray tracing problem that can be accelerated by RT cores [3, 11, 19, 20, 23, 26, 27]. The reductions typically involve representing the input data as geometries (leaf nodes of the BVH), query points as rays, and using intersection tests to perform the underlying computations. One of the proposed reductions involves mapping Nearest Neighbor Search (NNS) to ray tracing such that the intersection of a ray and an object indicates finding a neighbor.

A limitation of the reductions from prior work is that they only work when the *leaf nodes* are the sole contributors to the computation [19, 20, 27]. In contrast, internal nodes of the Barnes–Hut simulation are used to approximate the force exerted by a group of distant bodies. We find that the existing reductions fall short at problems like Barnes–Hut.

In this work, we introduce RT-BarnesHut, the first RT-accelerated Barnes–Hut computation, and propose a novel reduction that maps the components of Barnes–Hut to the ray-tracing counterparts. First, we build a Barnes–Hut tree using the bodies in the input dataset. Then, we map Barnes–Hut tree nodes onto triangles in the RT scene and model the query bodies as rays. Finally, we use the triangle-ray intersection tests performed by RT cores to determine whether to perform exact or approximate force computations.

To summarize, our contributions are as follows:

- We propose RT-BarnesHut, the first RT-accelerated Barnes–Hut computation.
- We create a novel mapping of Barnes–Hut tree nodes and query bodies to objects and rays, respectively, to better utilize the RT cores.
- We show that our approach is from 2× to 40× faster than the current state-of-the-art GPU-accelerated Barnes–Hut implementation.

The rest of the paper is organized as follows. We discuss the necessary background in Section 2. In Section 3, we illustrate the design of RT-BarnesHut. Section 5 showcases our evaluation against the state-of-the-art and is followed by a discussion of limitations in Section 6. We pivot from the Barnes–Hut problem and discuss how to extend our ideas to similar n-body problems in Section 7. Finally, we review the related work in Section 8 and conclude in Section 9.

2 Background

In this section, we detail the Barnes–Hut algorithm (Section 2.1), explain Ray Tracing (RT) cores and their programming model, and show an existing reduction of a database range query to the ray-tracing problem (Section 2.2).

2.1 Barnes–Hut Algorithm For Force Computation

The Barnes–Hut algorithm was proposed to improve the time complexity of n-body problems by treating a *group* of bodies as a single body and calculating the force exerted by this body [1]. The bodies in 2D space (respectively, 3D space) are spatially grouped by recursively dividing the space into quadrants (octants) and storing them into nodes of a quadtree (octree). The internal nodes of the quadtree (octree) represent the resulting *groupings* and the leaf nodes contain the individual bodies.

Fig. 1 shows the construction of a Barnes–Hut tree. A, B, C, D, E, F, G, and H are the bodies, which form the leaves of the tree. The nodes N_1, N_2, N_3, N_4, and N_5 represent empty or null leaf nodes corresponding to the empty quadrants in Fig. 1. The nodes I_1, I_2, and I_3 represent the internal nodes that store the center of mass and total mass of the nodes in subtrees rooted at the node.

2.1.1 Building Barnes–Hut Tree.

The Barnes–Hut algorithm uses a quadtree or octree [15] to represent the bodies and their spatial relationships. In the following sections, we will focus on the construction of quadtrees, but the same ideas can be extended to octrees.

The rationale for using a quadtree is to create *spatial groupings* of bodies. These groupings allow us to evaluate the force exerted by a *group* of bodies rather than every individual body, reducing the complexity of the algorithm from $O(n^2)$ to $O(n \cdot \lg(n))$. We create the spatial groupings by recursively subdividing the space containing the bodies such that each subdivision has *at most* one body.

Fig. 1 shows the construction of the Barnes–Hut tree for the running example. On the left, we see the spatial subdivision of the bodies A through H. Whenever multiple bodies share a quadrant, the quadrant is subdivided into four more quadrants. The process repeats until each body has its own quadrant. For example, the quadrant containing bodies E, F, G, and H is divided into 4 quadrants, out of which the quadrant housing F, G, and H is divided again so that each body has its own quadrant.

The resulting quadtree (on the right in Fig. 1) captures the hierarchical relationship between the bodies and quadrants. First, the internal node I_2 represents the quadrant containing E, F, G, and H and I_3. Next, I_3, a child of I_2, represents the quadrant containing F, G, and H. Finally, the leaves of the Barnes–Hut tree contain the bodies and nodes N_1 through

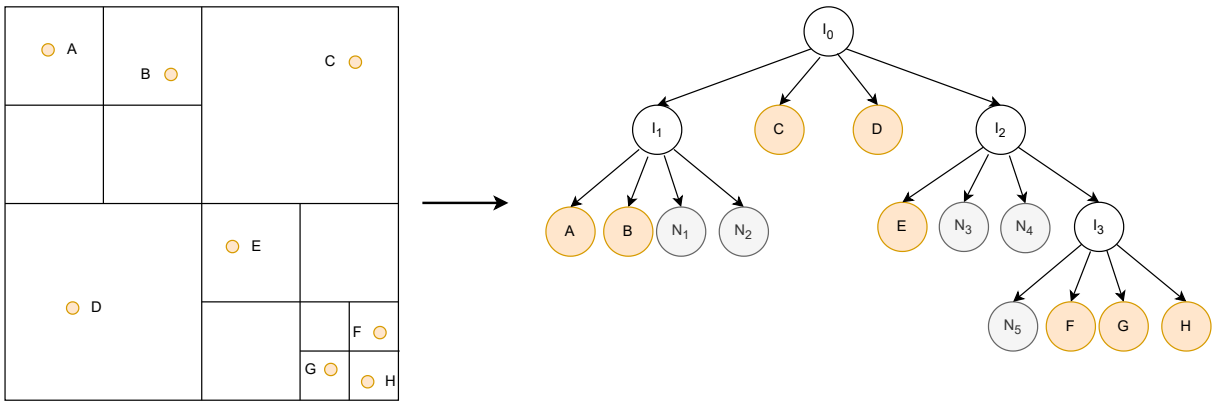

Figure 1. Barnes–Hut Tree Construction

N_5, which represent empty nodes that are artifacts of spatial subdivision. Moreover, the internal nodes (I_1, I_2, and I_3) of the tree store information about the center of mass and total mass of all bodies contained in the corresponding subtree.

2.1.2 Force Computation. This section walks through the Barnes–Hut force computation process to show how the algorithm reduces the time complexity. The crux of the optimization is as follows: if a group of bodies is sufficiently far away from the query body, we can approximate the force exerted by individual bodies in the group. The approximation calculates the force exerted by a virtual body with the mass equal to total mass of the group and placed at the center of mass of the group. This way, the algorithm avoids doing $O(n^2)$ computations.

During its depth-first traversal of the tree, Barnes–Hut uses a threshold condition to determine whether to truncate or continue traversal based on the distance from the query body. The threshold distance beyond which the force exerted by the bodies is approximated is defined by the formula:

$$s/d < \theta, \tag{1}$$

where s is the width of the quadrant corresponding to the current internal node; d represents the distance between the query body and the internal node's center of mass (center of mass of nodes contained within the internal node); θ is a user-defined constant that controls the accuracy of the computation. Larger θ value results in a more accurate force computation. It is typically set to 0.5.

In Fig. 1, assume that the width of the entire grid is 100. Then, the width of I_1 and I_2 is 50, and the width of I_3 is 25. If Condition (1) holds, the query body is far enough away that we can approximate the force. If the condition does not hold, we need to traverse the subtree rooted at the internal node and repeat this process. For example, if I_1 does not satisfy the condition, we continue traversing its children and compute the force exerted by leaf nodes A and B. If I_1 satisfies the condition, we would compute the force exerted by I_1 and jump to leaf node C.

Condition (1) only applies when we try to compute the force between the query body and an *internal* node. If the node is a leaf node, we directly perform the force computation between the two bodies. In the running example, since C and D are leaf nodes, we directly compute their forces.

Overall, traversing the Barnes–Hut tree involves a certain amount of jumping between nodes either in the DFS order (when Condition (1) holds) or in some special order (when Condition (1) fails). For the latter case, it is possible to statically determine where to jump on truncation. Hence, we install autoropes [6] to map each node to its jump location if traversal is truncated. More details in Section 3.3.

2.2 RT Cores

Ray tracing is a popular image rendering algorithm [25]. The process involves casting a ray from its source through pixels in the image plane and tracing its interactions with objects in the scene. Depending on the material of the objects in the scene, the rays can be reflected and/or refracted to produce secondary rays that are traced further to determine the final color of the pixel in the image plane.

Popularity of the ray-tracing problem prompted development of specialized hardware that accelerates parts of the ray-tracing pipeline [21]. In Section 2.2.1, we discuss the ray-tracing algorithm and follow it with a discussion of the RT cores and the OptiX programming model in Section 2.2.2. Finally, we look at prior work on accelerating general-purpose computations using RT cores in Section 2.2.3.

2.2.1 Ray Tracing. The heart of the ray tracing algorithm is the ray-object intersection test that determines whether the object will influence the color of the pixel corresponding to the ray. It is also the most time-consuming component in the ray tracing pipeline [4]. For m rays and n objects, the naive ray tracing algorithm requires $O(mn)$ intersection

tests, with more being needed if the original rays spawn reflected/refracted rays.

Similar to n-body problems that use spatial trees such as quadtree to represent the bodies, ray tracing relies on Bounding Volume Hierarchies (BVH). The BVH is a tree structure that represents an object-based subdivision of a scene. Similar to Fig. 1, the BVH is built by recursively combining nearby objects into *bounding volumes* until we are left with a bounding volume that encompasses the whole scene. The most popular choice of bounding volume is an Axis-Aligned Bounding Box (AABB).

BVH helps reduce the number of ray-object intersection tests by performing ray-bounding volume intersection tests instead. As bounding volumes enclose one or more objects or other bounding volumes, if the intersection test with the bounding volume fails, it is guaranteed that the ray cannot intersect anything enclosed by that bounding volume. This helps prune large parts of the BVH tree, reducing the complexity to $O(m \cdot lg(n))$ for m rays and n objects.

To summarize, the main components of the ray-tracing pipeline are as follows.

1. **Scene Construction:** set up the scene with objects
2. **Ray Definition:** create rays with a specific origin, direction, and length
3. **Intersection Test:** test for ray-object intersections
4. **Computation**: perform computations based on results of the intersection test

2.2.2 OptiX Programming Model.

The Ray Tracing cores in modern GPUs were designed to accelerate the BVH traversal and intersection tests. These RT cores share the same device memory as shader and tensor cores in NVIDIA GPUs, which allows us to offload RT-specific computations to the RT cores while the shader cores are free to perform other tasks. They also accelerate BVH traversal and ray-*triangle* intersection tests in hardware.

The OptiX API provides an interface to RT cores inside the larger CUDA framework. The API allows users to modify the following components of the ray tracing pipeline:

RayGen It is used to generate rays by specifying the origin, direction, and interval of the ray. The ray interval (T_{min}, T_{max}) decides the interval within which the ray is valid.
Intersection It is defined when the geometries in the scene are not triangles. The user is required to specify a custom ray-*object* intersection test program to be executed when the hardware reports a ray-*bounding volume* intersection.
AnyHit It is used to register an intersection (hit) and allows the user to decide what to do with the information. The user additionally has the option to terminate traversal on a hit.
ClosestHit It is invoked after the BVH traversal ends to identify the hit closest to the ray origin.
Miss It is invoked after the BVH traversal ends if the ray did not intersect *any* object.

2.2.3 Accelerating Database Queries using RT cores.

Prior works accelerate problems that use a spatial-tree optimization by reducing the problem to the ray tracing problem and employing RT cores. These reductions typically involve representing the dataset as objects in the scene and queries as rays such that ray-object intersection tests are used to decide whether or not to perform computations.

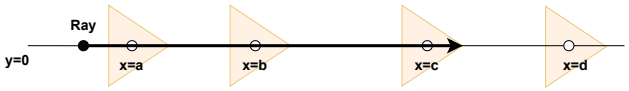

Figure 2. RT-accelerated database range query

RTIndex is an RT-accelerated database query processing algorithm that maps the database entries to 3D objects and queries to rays to solve range queries [11]. They use the column values (v_i) to create triangles such that each triangle has coordinates $(v_i, -0.5, -0.5)$, $(v_i{+}0.5, -0.5, 0.5)$ and $(v_i{-}0.5, 0.5, 0.5)$. In Fig. 2, to find database entries in the range (a, c), they launch a ray with $a - \epsilon$ as the origin and $t_{max} = c + \epsilon$ to find the desired entries.

An advantage of this reduction compared to others is that it uses triangles to represent the data. As ray-triangle intersection tests are accelerated in hardware, their performance will be significantly better. In Section 3.1, we show how we take inspiration from this algorithm to carefully create and place triangles such that the ray-triangle intersection tests can be used to perform force computations.

3 Design

In this section, we show how each component of the Barnes–Hut algorithm maps to a component of the ray-tracing pipeline (Section 2.2.1) to create RT-BarnesHut, the first RT-accelerated Barnes–Hut computation.

The first step is to set up the scene against which rays are traced. From Section 2.1, we know that we need to access the internal nodes of the Barnes–Hut tree during the Barnes–Hut force computation. From Section 2.2, we know that in ray tracing problems, objects in the scene are leaves of the Bounding Volume Hierarchy (BVH) and OptiX returns intersected objects. However, in the current OptiX setup, the user cannot access or modify the internal nodes of the BVH necessary for Barnes–Hut computations. Hence, the only way to ensure access to the internal nodes in RT-accelerated Barnes–Hut is to build the Barnes–Hut tree separately and add its nodes as objects to the BVH. This way, both internal and leaf nodes of the Barnes–Hut tree will be leaves of the BVH that is built over the scene.

The BVH is built over triangles representing the Barnes–Hut tree nodes, taking inspiration from the setup in Section 2.2.3 that accelerates database queries using RT cores. We use Condition (1) to set up triangles s-width apart and create rays of length θd such that the result of the intersection

decides whether or not to approximate force computation. Recall that s is the width of the quadrant, d is the distance between the query body and tree node and θ is a user-defined constant. We provide more details on the scene setup and ray definition in Sections 3.1 and 3.2, respectively. Section 3.3 explains how the result of ray-triangle intersection tests is used to compute forces. Finally, in Section 3.4, we put it all together to create the RT-BarnesHut algorithm.

3.1 Setting Up The Barnes–Hut Scene

First, we construct the Barnes–Hut tree using the standard algorithm (Section 2.1). Next, we apply a performance optimization to reduce the size of the tree by relaxing the restriction of one body per leaf node in the Barnes–Hut tree build (*bucketizing*) [12]. Finally, we set up the triangles that represent the nodes of the Barnes–Hut tree in the scene.

Bucketizing the Barnes–Hut Tree. Having one body per leaf node in the Barnes–Hut tree could result in trees of larger depth. This approach leads to deep and skewed Barnes–Hut trees in highly dense datasets where the coordinates of the bodies differ very slightly. Besides losing the logarithmic speedup of a tree, storing skewed trees of high depth consumes excessive memory. To counter this issue, we adopt ChaNGa's bucketizing optimization and store multiple bodies in the leaf nodes of the Barnes–Hut tree. We first sort the bodies in *Morton z-order* to ensure that bodies that are spatially close are grouped into *buckets* [18]. These buckets now form the leaf nodes of the Barnes–Hut tree.

Bucketizing typically lowers memory consumption as it results in trees of smaller depth. We note that the depth depends on the number of bodies in a bucket. If we assign a larger number of bodies to each bucket, it will result in a Barnes–Hut tree of smaller depth and vice-versa. However, we note that during force computation, we are required to iterate through the list of bodies in each bucket, resulting in a longer force processing time as opposed to just one body if the bucket size was 1.

We refer to the tree built over buckets of bodies as the *bucketized Barnes–Hut tree*. As the Barnes–Hut algorithm involves a depth-first traversal of the tree, we store the bucketized tree nodes in depth-first order to better mimic the original algorithm. After building the bucketized Barnes–Hut tree, we turn its nodes into triangles that form the RT scene.

Triangles Placement. We traverse the bucketized tree in the depth-first order and create a triangle corresponding to each node. The goal is to define and place triangles in appropriate locations such that the intersection test between a ray and triangle evaluates the Barnes–Hut termination condition (Condition (1)).

Recall that Condition (1) determines whether to continue or truncate Barnes–Hut tree traversal, which in turn determines whether we compute exact or approximate force. To encode this condition into the result of the ray-triangle intersection test, we first rewrite the termination condition[1]:

$$s/d < \theta \implies \theta d > s \tag{2}$$

Next, we incorporate the s and θd parameters into our triangle placement and ray definition, respectively.

We iterate through the nodes of the bucketized Barnes–Hut tree and place triangles such that the x-coordinate of the center of each triangle is offset by a distance s from $x = 0$. The y-coordinate of the center of each triangle is offset by a distance ρ so that the triangles do not overlap. The triangle vertices are also slightly offset by ϵ. This placement strategy is a 3D generalization of RTIndex, which solved 1D range queries on RT cores (Section 2.2.3).

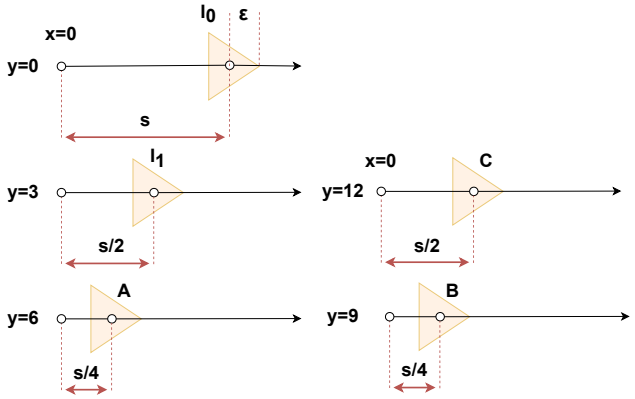

Figure 3. Barnes–Hut Triangles Setup

Fig. 3 shows triangles placement for the example from Fig. 1 assuming the depth-first path $I_0 \rightarrow I_1 \rightarrow A \rightarrow B \rightarrow C$. The vertices of the triangle representing node I_0 are $(s, 0, 1.0)$, $(s, -1.0, 1.0)$ and $(s, 1.0, 1.0)$, assuming $\epsilon = 1$. The next triangle, representing node I_1, will have vertices $(s/2, \rho, 1.0)$, $(s/2, -1.0 + \rho, 1.0)$ and $(s/2, 1.0 + \rho, 1.0)$. The vertices of the triangles representing nodes A and B will have the same x value of $s/4$ however their y values will be offset by $2 * \rho$ and $3 * \rho$ respectively. Subsequent triangles in the depth-first path increment the *y-offset* by ρ while the x value is set as the node's s value. Triangles representing nodes at the same level share the same x-coordinate but all have unique y-coordinates.

Recall that all internal nodes at a particular level share the same s value. For example, in Fig. 1, if we set the size of the whole grid (s) to be 100, the s value of nodes I_1, A, C, and I_2 will be 50 while their children will have s values of 25. This subdivision of the s value continues at every level of the Barnes–Hut tree and is stored in each Barnes–Hut node.

[1]As d (the distance between a body and a tree node's center-of-mass) is always positive, this rewrite is permissible.

3.2 Ray Generation

The second major component of the ray tracing pipeline (after the scene setup) is ray generation. In RT-BarnesHut, rays represent query bodies.

The s parameter from Condition (2) was used to space the triangles s distance away from $x = 0$. Now, we incorporate the θd parameter. Rays are defined by an origin, direction, and ray interval (Section 2.2.2). We initialize the origin of *all* rays representing query bodies to be $(0, 0, 0)$ and their direction as $(1, 0, 0)$. The ray interval $(t_{\min}, t_{\max}) = (0, \theta d)$, where d is the distance between the query body and root node of the Barnes–Hut tree.

In Fig. 3, if a ray launched from $(0, 0, 0)$ of length θd and direction $(1, 0, 0)$ intersects triangle I_0 placed s distance away, then Condition (2) holds and the traversal is truncated. Otherwise, the traversal goes on.

3.3 Intersection and Force Computation

The intersection test dictates the type of computation that needs to be performed and where traversal resumes:

- If a launched ray does not intersect a triangle, Condition (2) does not hold and the traversal goes on. This involves identifying the Barnes–Hut tree node associated with the triangle and examining its children. Since the bodies are stored in the depth-first order in the GPU memory, proceeding with the traversal means accessing the node next to the current one.
- If the ray intersects the triangle, the traversal skips the whole subtree and resumes at the location pointed to by *autoropes* [6].

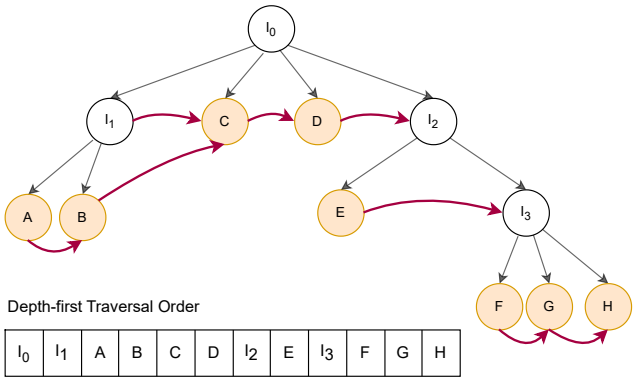

Figure 4. Barnes–Hut Autoropes Creation

Autoropes. We utilize autoropes [6] to map every node of the bucketized Barnes–Hut tree to the location from which traversal will resume should the traversal be truncated. The arrows in Fig. 4, which is a simplified version of the Barnes–Hut tree in Fig. 1 (after removing the empty nodes), represent the autoropes.

Since the Barnes–Hut tree traversal order (depth first) is statically determined, the same applies to the jump location (autorope) of each node:

$$node.autorope = node.index + SubTreeNodes(node) \quad (3)$$

The *SubTreeNodes* method takes a tree node as the input and returns the number of nodes present in the subtree rooted at the node, including the node itself. For example, in Fig. 4, let us assume that I_1 is sufficiently far away such that it can be approximated. To find the jump location, we first acquire the depth-first index of the current node, $node.index$, and then compute the number of nodes rooted at node's subtree, $SubTreeNodes(node)$. In this example, $node.index = 1$ and $SubTreeNodes(node) = 3$, so Equation (3) evaluates to $node.autorope = 1 + 3 = 4$. As shown in Fig. 4, this correctly truncates the traversal of the subtree rooted at node I_1 and jumps to node C located at index 4.

3.4 Putting It All Together: RT-BarnesHut

Using the components defined so far, Algorithm 1 formulates a ray-tracing query to calculate the force exerted on each body. In particular, tracing each ray mimics a depth-first traversal of the Barnes–Hut tree and uses ray-triangle intersection tests to determine when to truncate the traversal. Since the force computed by each body is independent of others, Algorithm 1 can run in parallel for several rays (input bodies) leveraging the GPU ray-tracing hardware.

The algorithm begins by examining the root node in Line 1. The origin of rays, which represent bodies in the dataset, is initially set to $(0, 0, 0)$ in Line 2. Line 3 begins the ray generation loop, where the ray will be tested against nodes of the Barnes–Hut tree until it reaches either the last node in the depth-first order or hits the truncation condition such that there are no further nodes to explore. For example, in Fig. 4, if the ray has completed testing for intersection with node H, the ray enables the *terminated* flag as there are no further nodes to test and exits the loop. Similarly, if the ray-triangle intersection test results in truncation of traversal at Node I_2 or I_3, the ray enables the *terminated* flag as Equation 3 evaluates to an autorope index *greater* than the number of nodes in the Barnes–Hut tree.

In Line 4, we compute the distance between the body associated with the ray and the Barnes–Hut node of interest, *node*, and multiply by the threshold value, θ. We set this as the length of the ray interval, t_{max}. Note that t_{max} corresponds to the parameter d from Condition 2. The $Intersect(ray, node)$ function in Line 5 returns true if the launched ray intersects the triangle associated with the Barnes–Hut *node* and false otherwise. The $Intersect(ray, node)$ function checks whether $s/d < \theta$ holds. For example, in Fig. 3, a ray originating at $(0, 3, 0)$ launching toward the triangle representing I_1 will intersect *only* if $rayLength >= s/2$.

Algorithm 1: RT-BarnesHut

```
   Input  : Ray ray, BHNode root
1  node ← root
2  ray.origin ← (0, 0, 0)
3  while ray != terminated do
4      ray.t_max ← distance(ray, node) * θ
5      if Intersect(ray, node) then
6          if node.leaf then
7              for Body b ∈ node.bucket do
8                  ComputeForce(ray, b)
9              end
10         else
11             ComputeForce(ray, node)
12         end
13         ray.origin ← node.autorope
14     else
15         if node.leaf then
16             for Body b ∈ node.bucket do
17                 ComputeForce(ray, b)
18             end
19         end
20         ray.origin ← DFSuccesor(node.index)
21     end
22     node ← getNode(ray.origin)
23 end
```

On a ray-triangle intersection hit, we first check if the node of interest is a leaf in Line 6. The bucketing optimization described in Section 3.1 necessitates iterating through all bodies in the leaf and calculating the cumulative force exerted using the $ComputeForce(ray, b)$ function, as shown in Lines 7-9. If the node is an internal node, we approximate the force by calling $ComputeForce(ray, node)$[2] in Line 10. We define $ComputeForce(ray, node)$ as follows:

Definition 3.1. $ComputeForce(ray, node)$ Given a *ray*, corresponding to a query body and a tree *node*, compute the gravitational force of attraction:

$$F = G\frac{m_1 m_2}{r^2}$$

- $m_1 \leftarrow DFSnodes[ray.ID].mass$,
- $m_2 \leftarrow node.mass$,
- $r \leftarrow$ distance between the center of masses of the tree node and query body,
- $G \leftarrow$ gravitational constant,
- $F \leftarrow$ magnitude of the force of attraction.

At the end of processing a hit, we set the ray's origin to the autorope location (Line 13) and the target node as the autorope node (Line 22). For example, if the ray intersects the triangle representing node I_1 in Fig. 3, the next ray launch would be towards the triangle representing node C, since I_1 autoropes to C as shown in Fig. 4.

On a ray-triangle intersection miss (Lines 15-18), if the node is a leaf, we perform the force computation on each of the bodies in the node's bucket as shown. Following this, we set the ray's origin to launch toward the next node in the depth-first traversal order in Line 20. For example, if the ray does not intersect I_1 in Fig. 3, the next ray launch would be towards the triangle representing node A, since it is the depth-first successor of I_1 as shown in Fig. 4. Finally, we set the next node to test as the Barnes–Hut node corresponding to the updated ray origin in Line 22. For I_1 in Fig. 3, *node* will be set to the Barnes–Hut node corresponding to either C or A, depending on the result of $Intersect(ray, node)$.

4 Implementation

RT-BarnesHut implementation builds on top of the Optix Wrapper Library (OWL)[3], which abstracts some aspects of Optix 8.0. Below we explain how different Optix programs (Section 2.2.2) map on the implementation of our algorithm.

Our RayGen program iteratively generates rays to mimic a depth-first traversal of the Barnes–Hut tree. Each ray corresponds to a unique OptiX launch index acquired using *optixGetLaunchIndex*. The rays are launched using the *optixTraverse* function call, setting $t_{\min}$ to 0, $t_{\max}$ to *rayLength*, and the direction to $(1, 0, 0)$. The *optixTraverse* function returns a hit or miss object, which is used by the *optixInvoke* function call to invoke the appropriate final program (Miss or ClosestHit).

To improve the performance of ray-triangle intersections, we disable the AnyHit program and terminate the ray on the first intersection hit. Furthermore, we try to limit local and global memory accesses and, hence, store commonly referenced values inside ray payload registers. These ray payloads allow us to pass data between different programs using a copy-in/copy-out format.

The ClosestHit program is invoked when a ray intersects the triangle, implying that the traversal truncation condition holds. We compute the force and store the result in a ray payload register. We then store the ray origin and index of the Barnes–Hut node corresponding to the autorope location in another set of ray payload registers. This information will be used by the RayGen program to launch the next ray.

The Miss program is invoked when there is no intersection, implying that we must traverse the Barnes–Hut tree further. However, if the node of interest is a leaf, we still compute the force. We then store the ray's next origin location and index of the next Barnes–Hut node in depth-first search order to be used by the subsequent ray launch.

[2]Note that both body *b* and node *node* contain mass and center of mass information and can use the same ComputeForce(ray, node) function for force computation

[3]https://owl-project.github.io/

5 Evaluation

In this section, we evaluate the performance of RT-BarnesHut against state-of-the-art GPU-accelerated Barnes–Hut baselines. We also break down the execution time of RT-BarnesHut and analyze the impact of preprocessing on performance. Lastly, we analyze the throughput of RT-BarnesHut in terms of memory and compute.

5.1 Experimental Setup

We perform the experiments on a system that has 48 24-core AMD Ryzen Threadripper PRO 5965WX CPUs and runs Ubuntu 22.04.4 LTS. The system has one NVIDIA GeForce RTX 4070 Ti GPU with 12 GB device memory that runs CUDA 12.2 and Optix 8.0.

5.1.1 Datasets. We use a mixture of synthetic and real-world datasets to evaluate RT-BarnesHut.

synthetic: Poisson distribution of bodies in a cubical volume. *25M* points.
lambb: A simulation of a 71M megaparsecs cubical volume of the universe. *80M* points.
dwarf: A simulation of a dwarf galaxy forming in a 28.5M megaparsecs cubical volume of the universe with 30% dark matter and 70% dark energy. The bodies are concentrated in the center. *4M* and *50M* points.

5.1.2 Baselines. We use ChaNGa, the state-of-the-art GPU-based Barnes–Hut implementation as the primary baseline. ChaNGa is an *n*-body simulator that uses Charm++ to efficiently parallelize and balance workload across multiple nodes and cores [12]. We use the GPU-accelerated variant that performs both the Barnes–Hut tree walk and force computations on the GPU [14]. Since RT cores (and, in turn, RT-BarnesHut) only support single precision computations, we evaluate against a single-precision configuration of ChaNGa.

Treelogy is a benchmark suite for GPU-accelerated, tree-based algorithms, including Barnes–Hut [10]. Treelogy uses shared-memory optimizations to accelerate Barnes–Hut. Since Treelogy does not bucketize the Barnes–Hut tree, it runs out of memory on the real-world datasets. Due to this failure, we only take a brief look at this baseline in Section 5.2.

5.1.3 Bucketizing. In Section 3.1, we discussed creating buckets with spatially close bodies as the leaves of a Barnes–Hut tree to save memory. This optimization results in a tree of a smaller depth, which consumes less memory and takes less time to traverse. However, the force computation now takes more time since we need to compute the force exerted by all bodies in a bucket individually. In other words, inside a bucket we apply the naïve algorithm.

ChaNGa authors did an extensive evaluation of the optimal bucket size for their setup [13] and came up with the bucket size that works best: 32. In our experiments, 32 particles per bucket also worked best for RT-BarnesHut. Hence, we set 32 as the bucket size throughout the evaluation.

5.1.4 Computational Pipeline. Our implementation contains the following stages:

1. Reading the dataset from a file
2. Building the Barnes–Hut tree out of the dataset using the given bucket size as a parameter (if supported)
3. Preprocessing the tree
4. Running the computation kernel of Barnes–Hut

Stages 1 and 2 are not specific to an implementation of the algorithm: both baselines and RT-BarnesHut have their own versions of these stages but they are the same in essence. Hence, we do not report the run times for these stages.

As the difference lies in stages 3 and 4, they are included in the experiment. Also, Section 5.2.1 evaluates stage 4 individually. In the case of RT-BarnesHut, pre-processing (stage 3) includes traversing the Barnes–Hut tree in depth-first order to determine the locations of the triangles, installing autoropes, and OptiX-specific operations such as building the scene and initializing GPU structures related to RT cores. The computational kernel of RT-BarnesHut (stage 4) closely follows Algorithm 1.

5.2 Performance Evaluation

Treelogy. Table 1 summarizes our evaluation of RT-BarnesHut vs. Treelogy on synthetic datasets. RT-BarnesHut is roughly 2x faster than Treelogy. Moreover, Treelogy runs out of memory for the synthetic-50M dataset as its tree is too big to fit in device memory.

Table 1. Execution time of Treelogy and RT-BarnesHut

Dataset	Treelogy	RT-BarnesHut	Speedup
synthetic-10M	5.7s	2.7s	2.1x
synthetic-25M	17.0s	7.7s	2.2x
synthetic-50M	-	12.04s	-

Table 2. Execution time of ChaNGa and RT-BarnesHut

Dataset	ChaNGa	RT-BarnesHut	Speedup
synthetic-25M	6.0s	1.0s	6x
dwarf-4M	4.0s	0.2s	20x
dwarf-50M	49.7s	1.2s	41.4x
lambb-80M	26.5s	4.7s	5.6x

ChaNGa. Table 2 shows the total runtime of ChaNGa and RT-BarnesHut for one time step when the bucket size is set to 32 for both implementations. We report results averaged over five runs. Our speedups range from 5.6x on the lambb dataset to 41x on the dwarf-50M dataset. We observe that the execution time is sensitive to both the size and distribution of the input dataset. The RT cores seem more adept at handling

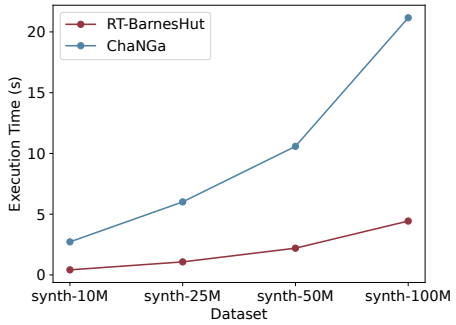

Figure 5. Scalabilty of RT-BarnesHut compared to ChaNGa

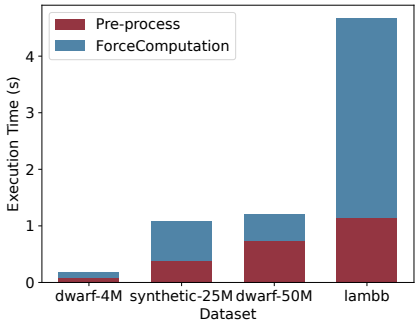

Figure 6. Breakdown of RT-BarnesHut execution time

datasets such as dwarf, where points are concentrated around the center.

Although ChaNGa uses shader cores in GPUs to accelerate tree traversal and force computations, we find the RT cores are better at accelerating Barnes–Hut due to their specialization to tree algorithms.

5.2.1 Computation Kernel. Table 3 reports only on the computation kernel (cf. Section 5.1.4) that is central to both implementations. Our RT-accelerated kernel is consistently faster on all datasets, with a speed up of 93x on the dwarf-50M dataset, 37x on the dwarf-4M dataset and 6x on the lambb dataset.

5.2.2 Scalability. To evaluate the ability of RT-BarnesHut and ChaNGa to scale to larger dataset sizes, we construct synthetic datasets containing 10M-100M Poisson-distributed bodies. Fig. 5 shows the growth of execution time vs. dataset size and we see that the growth rate is significantly slower for RT-BarnesHut compared to ChaNGa (*i.e.*, gap between execution times of ChaNGa and RT-BarnesHut increases with dataset size). As the RT cores are designed to handle large numbers of rays and objects, our ability to leverage them results in improved scalability for RT-BarnesHut.

5.2.3 Pre-processing Time. Fig. 6 splits the execution time of RT-BarnesHut into pre-processing and computation kernel to find out the dominant factor of the execution time. Overall, the ratio between the two phases is input-sensitive: pre-processing can take anywhere from 24% to 60% of the execution time. In particular, Fig. 6 shows that for dwarf-50M, more than 60% of execution time is spent on pre-processing. It takes up 46% on the dwarf-4M dataset, 24% on lambb and 35% on synthetic-25M.

5.2.4 Compute Throughput. We compute the throughput of RT-BarnesHut and ChaNGa as the number of rays processed per second and the number of bodies processed per second, respectively. When traversing the Barnes–Hut tree to perform force computation, we modify the ray origin and interval as required by the algorithm. Each modification generates a new ray that contributes to the total ray count used in the throughput computation. The number of rays is always much larger than the number of bodies, as each body is initially mapped to a ray, which, in turn, can spawn multiple rays.

Table 4 shows the throughput of both approaches. As the RT hardware is designed to handle large numbers of rays in parallel, RT-BarnesHut can process more than 9 billion rays per second. ChaNGa processes around 4 million bodies per second, showing that RT-BarnesHut can handle large datasets effectively.

5.2.5 Memory Throughput. We use the NSight Compute profiler to compute the memory throughput. We used the *dram__throughput.avg.pct_of_peak_sustained_elapsed* metric which represents the percent of the peak sustained rate achieved during elapsed cycles across all sub-units. The throughput is calculated as the percentage of utilization with respect to the theoretical maximum. We average the

Table 3. Computation kernel execution time of ChaNGa and RT-BarnesHut for the bucket size of 32.

Dataset	ChaNGa	RT-BarnesHut	Speedup
synthetic-25M	4.4s	0.7s	6.3x
dwarf-4M	3.7s	0.1s	37x
dwarf-50M	46.5s	0.5s	93x
lambb-80M	21.2s	3.5s	6.0x

Table 4. Computation throughput of ChaNGa and RT-BarnesHut for bucket size of 32

Dataset	ChaNGa (bodies/sec)	RT-BarnesHut (rays/sec)
synthetic-25M	4159733	3702222777
dwarf-4M	1266271	9117767957
lambb-80M	3045771	35864699

per-kernel throughput to present the average throughput for the whole program. Our memory throughput results are shown in Table 5.

6 Limitations

In this section, we discuss how certain limitations of the current RT cores interface, as implemented by NVIDIA, affect our work: RT cores only support single-precision floating-point arithmetic and are opaque to the profiler.

6.1 Single Precision

A general limitation of current RT-core technology is its support of single-precision floating-point numbers only. Yet, single precision is common for simulations where the domain allows it naturally. Besides, some optimization techniques (e.g. CPU-based SIMD acceleration) use single precision as well. In practice, every particular simulation task will have a specific demand for precision and can opt for other, slower implementations of the Barnes–Hut algorithm if single precision is not enough for its purposes.

In the RT-BarnesHut algorithm, force computation (Definition 3.1) is performed on the shader cores with double precision. However, the RT cores, which evaluate the condition to decide whether to truncate traversal or not, only supports single precision computations. As the traversal truncation condition controls the accuracy of the Barnes–Hut simulation, the lack of precision could lead to better or worse simulation accuracy. If we err on the side of truncating the traversal, we approximate more than the reference Barnes–Hut implementation. Otherwise, we approximate less and our force computation is more precise. In our experiments, we found that the single precision limitation had negligible effect on the force computation results compared to a reference Barnes–Hut implementation.

Ultimately, lesser precision plagued GPGPU programmers in the early days of traditional shader cores. The issue was eventually addressed by the manufacturer. Our work provides a compelling reason to do the same for RT cores by highlighting general applicability of the technology. If support for double precision is added to RT cores, RT-BarnesHut will require minimal modifications to adopt the change.

6.2 Opaque Utilization

NVIDIA's RT hardware internals is proprietary information and we cannot get numbers that would show how well we utilize this hardware. There is a positive movement from NVIDIA in this regard, though: RT cores profiling was added to the Nsight Graphics (Pro build) profiler recently. Our issue is that the profiler can only analyze rendering of images. We cannot use the profiler since RT-BarnesHut only computes and does not render anything. Besides, another CUDA profiler, Nsight Compute, still does not support RT cores.

7 Beyond RT-BarnesHut: Generalized *n*-body Problems and RT Cores

The Barnes–Hut algorithm is an example from a class of problems referred to as *generalized n-body problems* [8]. Such problems perform a computation concerning interactions between all pairs of n bodies (e.g. nearest neighbor search, collision detection in graphics, spatial joins [5], specific types of simulations in modelling). The naïve approach to solving generalized n-body problems loops through all pairs of bodies leading to an algorithm with the $O(n^2)$ time complexity. The quadratic complexity is not feasible for large values of n, hence, other approaches trade some accuracy for better complexity. An optimization usually involves recursively subdividing the space of bodies and grouping together bodies that are spatially close. A *spatial tree structure* represents the groupings, and computations now happen over these groups of data.

Generalized n-body problems differ in how they process the spatial tree:

1. Only leaf nodes of the tree contribute to the computation.
2. Both the leaf and internal nodes of the tree contribute to the computation.

For instance, nearest-neighbor problems come under the category (1) since the distance computation is only done between the bodies, and the bodies are leaf nodes. In contrast, Barnes–Hut requires the internal nodes contribute to the computation too since the internal nodes allow estimating the force exerted by the leaves in the subtree and to decide whether to proceed with the traversal of the subtree.

Spatial trees algorithms that process internal nodes (category 2 above) reduce the number of computations by approximating the effect of parts of the tree that lie beyond

Table 5. Memory throughput of ChaNGa and RT-BarnesHut for bucket size of 32

Dataset	ChaNGa (%)	RT-BarnesHut(%)
synthetic-25M	10.7	30.8
dwarf-4M	7.6	20.8
lambb-80M	6.7	46.6

Table 6. Characterization of Barnes–Hut Algorithm

Spatial Tree Characteristic	Barnes–Hut
Truncation condition	Is $s/d < \theta$?
Operation on truncation	Approximate force and autorope to next location
Operation at leaf	Compute force between query body and leaf

a threshold distance. Accordingly, such algorithms can be *characterized* by three features:

Truncation condition determines whether to truncate the traversal in the current subtree or not
Operation on truncation defines computation to be performed when the truncation conditions holds and the subsequent jump to the next tree node to process
Operation at leaf defines computation to be performed at the leaf node

Table 6 shows how Barnes–Hut fits this characterization.

The features of the algorithm identified as above may be mapped to RT core primitives. For example, Table 7 summarizes the mapping of parts of Barnes–Hut to a ray-tracing problem implemented in RT-BarnesHut. In particular, the truncation condition is encoded in scene creation and ray generation components. The operation on truncation translates to intersection hit callback that approximates the force and resumes the traversal at the autorope location. The operation at leaf is handled by computing the exact force between the body in the leaf and the query body.

Based on our characterization, it should be possible to approach other spatial tree algorithms and map them to RT cores. The challenge is to go through the two steps: first, identify the parts of the algorithm as in Table 6, and, second, map the parts on the RT core primitives as in Table 7.

8 Related Work

Barnes–Hut. Barnes and Hut [1] proposed the Barnes–Hut optimization to reduce the complexity of n-body problems by approximating long-range forces. Grama et al. [7] recognized the parallel nature of Barnes–Hut computations and introduced parallel Barnes–Hut algorithms based on particle distributions. Burtscher and Pingali [2] implemented a GPU-based (CUDA) Barnes–Hut algorithm, which was over 70x faster than CPU implementations. Hegde et al. [10] proposed Treelogy, which characterized different tree traversal and implemented an optimized shared-memory-based Barnes–Hut algorithm. Jetley et al. [12] proposed ChaNGa, which uses Charm++ to optimize computation and communication costs for n-body simulations. Liu et al. [14] optimized ChaNGa for GPUs by minimizing the communication overhead between the CPU and GPU.

Table 7. Translation of Barnes–Hut to Ray Tracing

Ray Tracing Component	Barnes–Hut Component
Scene	Barnes–Hut Tree nodes (represented as triangles)
Rays	Bodies (coordinates from input dataset)
Intersection Condition	$s/d < \theta$
Intersection Hit	Estimate/compute force of internal/leaf node
Intersection Miss	Compute force if leaf or test next triangle

Accelerating non-Ray-Tracing Applications using RT Cores. Wald et al. [23] introduced the idea of using Rt cores to solve the point-in-tet problem, where the goal is to identify the tetrahedron to which a point a belongs. Other works have focused on accelerating other graphics problems using RT cores [16, 17]. Zellmann et al. [26] proposed a reduction of the fixed-radius nearest neighbor search (NNS) query to a ray tracing query by exploiting the geometric interpretation of neighbor searches in the Euclidean space. Other works proposed optimizations and variations to solving the NNS problem [3, 20, 27], including DBSCAN clustering, which uses NNS as a sub-routine [19]. Henneberg and Schuhknecht [11] proposed RTIndex to accelerate database range queries using RT cores by modeling the queries as rays and the database as triangles in the scene.

9 Conclusion

In this paper, we describe RT-BarnesHut, the first RT-accelerated Barnes–Hut algorithm. As the Barnes–Hut algorithm involves computations at *both* the leaf *and* internal nodes of the tree, we build the Barnes–Hut tree separately without directly relating it to the Bounding Volume Hierarchy of the Ray-Tracing architecture. By carefully mapping different components of the Barnes–Hut algorithm to their ray-tracing counterparts, we create a novel reduction that is between 2x to 40x faster than the state-of-the-art GPU-based baseline. Future work entails mapping other generalized tree traversals to RT cores.

References

[1] J. E. Barnes and P. Hut. 1986. A hierarchical O(n-log-n) force calculation algorithm. *Nature* 324 (1986), 446.

[2] Martin Burtscher and Keshav Pingali. 2011. Chapter 6 - An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm. In *GPU Computing Gems Emerald Edition*, Wen mei W. Hwu (Ed.). Morgan Kaufmann, Boston, 75–92. https://doi.org/10.1016/B978-0-12-384988-5.00006-1

[3] I. Evangelou, G. Papaioannou, K. Vardis, and A. A. Vasilakis. 2021. Fast Radius Search Exploiting Ray Tracing Frameworks. *Journal of Computer Graphics Techniques (JCGT)* 10 (5 February 2021).

[4] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. 1986. ARTS: Accelerated Ray-Tracing System. *IEEE Computer Graphics and Applications* 6, 4 (1986), 16–26. https://doi.org/10.1109/MCG.1986.276715

[5] Liang Geng, Rubao Lee, and Xiaodong Zhang. 2024. RayJoin: Fast and Precise Spatial Join. In *Proceedings of the 38th ACM International Conference on Supercomputing* (Kyoto, Japan) *(ICS '24)*. Association for Computing Machinery, New York, NY, USA, 124–136. https://doi.org/10.1145/3650200.3656610

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

[7] A.Y. Grama, V. Kumar, and A. Sameh. 1994. Scalable parallel formulations of the Barnes-Hut method for n-body simulations. In *Supercomputing '94:Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. 439–448. https://doi.org/10.1109/SUPERC.1994.344307

[8] Alexander G. Gray and Andrew W. Moore. 2000. 'N-body' problems in statistical learning. In *Proceedings of the 13th International Conference on Neural Information Processing Systems* (Denver, CO) *(NIPS'00, Vol. 13)*. MIT Press, Cambridge, MA, USA, 500–506.

[9] Tsuyoshi Hamada, Tetsu Narumi, Rio Yokota, Kenji Yasuoka, Keigo Nitadori, and Makoto Taiji. 2009. 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Portland, Oregon) *(SC '09)*. Association for Computing Machinery, New York, NY, USA, Article 62, 12 pages. https://doi.org/10.1145/1654059.1654123

[10] Nikhil Hegde, Jianqiao Liu, and Milind Kulkarni. 2016. Treelogy: a benchmark suite for tree traversal applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–2. https://doi.org/10.1109/IISWC.2016.7581286

[11] Justus Henneberg and Felix Schuhknecht. 2023. RTIndeX: Exploiting Hardware-Accelerated GPU Raytracing for Database Indexing. *Proc. VLDB Endow.* 16, 13 (sep 2023), 4268–4281. https://doi.org/10.14778/3625054.3625063

[12] Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas Quinn. 2008. Massively parallel cosmological simulations with ChaNGa. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–12. https://doi.org/10.1109/IPDPS.2008.4536319

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

[14] Jianqiao Liu, Michael Robson, Thomas Quinn, and Milind Kulkarni. 2019. Efficient GPU tree walks for effective distributed n-body simulations. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. Association for Computing Machinery, New York, NY, USA, 24–34. https://doi.org/10.1145/3330345.3330348

[15] Donald Meagher. 1980. Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer. (10 1980).

[16] Nate Morrical, Will Usher, Ingo Wald, and Valerio Pascucci. 2019. Efficient Space Skipping and Adaptive Sampling of Unstructured Volumes Using Hardware Accelerated Ray Tracing. *2019 IEEE Visualization Conference (VIS)* (2019), 256–260.

[17] Nate Morrical, Ingo Wald, Will Usher, and Valerio Pascucci. 2020. Accelerating Unstructured Mesh Point Location with RT Cores. *IEEE transactions on visualization and computer graphics* (2020).

[18] GM Morton. 1966. A Computer Oriented Geodetic Data Base; and a New Technique in File Sequencing.

[19] V. Nagarajan and M. Kulkarni. 2023. RT-DBSCAN: Accelerating DBSCAN using Ray Tracing Hardware. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, Los Alamitos, CA, USA, 963–973. https://doi.org/10.1109/IPDPS54959.2023.00100

[20] Vani Nagarajan, Durga Mandarapu, and Milind Kulkarni. 2023. RT-kNNS Unbound: Using RT Cores to Accelerate Unrestricted Neighbor Search. In *Proceedings of the 37th International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 289–300. https://doi.org/10.1145/3577193.3593738

[21] NVIDIA. 2021. NVIDIA Turing Architecture Whitepaper. https://gpltech.com/wp-content/uploads/2018/11/NVIDIA-Turing-Architecture-Whitepaper.pdf

[22] Laurens van der Maaten. 2013. Barnes-Hut-SNE. *CoRR* abs/1301.3342 (2013). https://api.semanticscholar.org/CorpusID:208915826

[23] Ingo Wald, Will Usher, Nathan Morrical, Laura Lediaev, and Valerio Pascucci. 2019. RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location. In *High-Performance Graphics - Short Papers*. The Eurographics Association.

[24] M.S. Warren and J.K. Salmon. 1992. Astrophysical N-body simulations using hierarchical tree data structures. In *Supercomputing '92:Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*. 570–576. https://doi.org/10.1109/SUPERC.1992.236647

[25] Turner Whitted. 1980. An Improved Illumination Model for Shaded Display. *Commun. ACM* 23, 6 (jun 1980), 343–349. https://doi.org/10.1145/358876.358882

[26] Stefan Zellmann, Martin Weier, and Ingo Wald. 2020. Accelerating Force-Directed Graph Drawing with RT Cores. In *2020 IEEE Visualization Conference (VIS)*. 96–100. https://doi.org/10.1109/VIS47514.2020.00026

[27] Yuhao Zhu. 2022. RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 76–89. https://doi.org/10.1145/3503221.3508409

A Artifact Appendix

There are 3 artifacts associated with our paper:

RT-BarnesHut: Our contribution, an RT-accelerated Barnes–Hut algorithm.
ChaNGa: Our primary baseline, a tree-based, GPU-accelerated Barnes–Hut algorithm.
Treelogy: Another tree-based, GPU-accelerated Barnes–Hut algorithm.

A.1 Hardware Dependencies

We perform the experiments on a system that has 48 24-core AMD Ryzen Threadripper PRO 5965WX CPUs and runs Ubuntu 22.04.4 LTS. The system has one NVIDIA GeForce RTX 4070 Ti GPU with 12 GB device memory that runs CUDA 12.2 and Optix 8.0.

A.2 RT-BarnesHut Artifact

This artifact provides all the files required to run RT-BarnesHut. It is built using the Optix Wrapper Library (OWL). We provide the source code and scripts required to reproduce results reported in the paper.

A.2.1 Environment Setup.

1. Download Datasets:
   https://zenodo.org/records/14219911 and https://zenodo.org/records/14220233
2. Clone RT-BarnesHut
   ```
   git clone https://github.com/vani-nag/
       OWLRayTracing.git .
   git checkout BarnesHutRT
   ```
3. Build OWL
   - OptiX 8 SDK
   - CUDA version 12.2
   - C++11 capable compiler (regular gcc on CentOS, Ubuntu, or any other Linux should do; as should VS on Windows)
   - CMake for building
     ```
     sudo apt install cmake-curses-gui
     ```
   - if you want to build the graphical examples: glfw (sudo apt-get install libglfw3-dev), or all the libraries to build it from included source code (sudo apt-get install x11-xserver-utils libxrandr-dev libxinerama-dev libxcb-xkb-dev libxcursor-dev libxcb-xinput-dev libxi-dev)
4. Build RT-BarnesHut
   ```
   mkdir build
   cd build
   cmake ..
   make
   ```
5. Run Sanity Check
   To run a small 1M points dataset for 5 iterations to ensure installation went smoothly, use `./run_script.sh sanitycheck`. Observe console output for *Execution time* after completion of script to indicate successful setup. Ensure datasets are in root directory and names match those in `run_script.sh`.

A.2.2 Evaluation and Expected Results.

The `run_script` runs all the experiments we used in the paper to create Figures 5 and 6 and Tables 1, 2 and 3. This script has 2 input arguments: (i) the experiment to run and (ii) number of iterations. The number of iterations dictates the number of times each dataset is run.

In addition to printing results to the console output, the script also stores the console output to an output file in the `/outputs` folder for each iteration. For example, the output file `synthetic25M_output_3.txt` corresponds to the programs console output for the 3rd iteration run for the synthetic25M dataset.

Reproducing Table 1 Results: Run

```
./run_script.sh treelogy 5
```

to run RT-BarnesHut on treelogy_synthetic_10M, treelogy_synthetic_25M, and treelogy_synthetic_50M datasets for 5 iterations each. After each benchmark is completed, the results of RT-BarnesHut are averaged and print out to the console.

Reproducing Tables 2, 3 and Figure 6 Results: Run

```
./run_script.sh changa 5
```

to run RT-BarnesHut on synthetic-25M, dwarf-4M, dwarf-50M , and lambb-80M datasets for 5 iterations each. Similarly, after each dataset has completed execution, the results are averaged and print out to the console.

Reproducing Figure 5 Results: This experiment shows the scalability of RT-BarnesHut by steadily increasing the number of points from 10 million to 100 million and reporting the execution time. Run

```
./run_script.sh scalability 5
```

to run RT-BarnesHut on synthetic-10M, synthetic-25M, synthetic-50M, and synthetic-100M datasets for 5 iterations each.

A.3 ChaNGa Artifact

ChaNGa requires Charm++ for parallelization and load balancing. Please use the `run_tests_fig5.py` and `run_tests_table2.py` scripts to reproduce data from the paper. These scripts need to be executed from the `changa/teststep` directory. ChaNGa executes each benchmark in under 1 minute and we measure the average execution time over 5 runs.

A.3.1 Environment Setup.

1. Download Datasets https://zenodo.org/records/10976384
2. Download scripts https://zenodo.org/records/14220145
3. Set up ChaNGa
   - Download ChaNGa

```
git clone http://github.com/N-BodyShop/
    changa.git
git clone http://github.com/N-BodyShop/
    utility.git
```

- Download Charm++

```
git clone http://github.com/UIUC-PPL/
    charm.git
git checkout v7.0.0
export CHARM_DIR=charm/
```

- Build Charm++ with ChaNGa

```
cd charm/
./build \changa netlrts-linux-x86_64-
    cuda --with-production --force
```

- Configure ChaNGa with cuda

```
cd ../changa/
./configure --enable-gpu-local-tree-walk
     --with-cuda -- with-cuda-level=80
```

- Build ChaNGa

```
make clean && make
```

A.3.2 Evaluation and Expected Results.

Please copy the `run_tests_fig5.py` and `run_tests_table2.py` files from the zenodo repository and paste it in `changa/teststep`. Also, download the datasets and place them in a folder titled `changa/benchmarks`.

Reproducing Table 2 Results: It prints out the average execution time of five runs for the four reported datasets. Run

```
python3 run_tests_table2.py
```

Reproducing Figure 5 Results: It prints out the average execution time of five runs for the four synthetic datasets. Run

```
python3 run_tests_fig5.py
```

A.4 Treelogy Artifact

The artifact includes scripts required to produce results reported in the paper.

A.4.1 Environment Setup.

1. Download Datasets: https://zenodo.org/records/14220233
2. Set up Treelogy

```
git clone https://github.com/RohanG0407/
    RTBarnesHut-Treelogy.git  .
```

3. Build Treelogy

```
cd ./GPU/barneshut/octree/gpu_non_lockstep
make clean && make
```

Ensure datasets are in root directory and names match those in `run_script.sh`.

A.4.2 Evaluation and Expected Results.

We use the file `run_script.sh` to reproduce data from Table 1.

```
cd ./GPU/barneshut/octree/
    gpu_non_lockstep
run_script.sh 5
```

The command is used to run Treelogy on treelogy_synthetic_10M, treelogy_synthetic_25M, and treelogy_synthetic_-50M datasets for 5 iterations each. After each benchmark is completed, the results of Treelogy are averaged and print out to the console. Note, for the 50M dataset it is expected to run into a memory error as the tree build process for Treelogy does not scale well.