Fast Database Join on Ray-tracing Core Equipped GPU

by

Yijie Wu
B.Sc., University of Victoria, 2023

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We acknowledge and respect the Lək̓ʷəŋən (Songhees and Xʷsepsəm/ Esquimalt) Peoples on whose territory the university stands, and the Lək̓ʷəŋən and W̱SÁNEĆ Peoples whose historical relationships with the land continue to this day.

Fast Database Join on Ray-tracing Core Equipped GPU

by

Yijie Wu
B.Sc., University of Victoria, 2023

Supervisory Committee

Dr. Sean Chester, Supervisor
(Department of Computer Science)

Dr. Brandon Haworth, Departmental Member
(Department of Computer Science)

Abstract

With the increase in GPU memory and computing power, GPU databases have become more popular, driving extensive research on GPU-based indexing. One study introduced a novel approach called RTX(Ray-tracing Index), which utilizes ray-tracing cores(RT cores) to accelerate GPU indexing. However, RTX suffers from a large build size and slow range queries. A follow-up work called cgRX(Coarse-granular Indexing), optimized the construction and range query algorithms, improving throughput by 1.5x–3x in relation to memory footprint, the range query time by 2x, and 5.5x faster updatability compared to RTX. However, the experimental results of cgRX may be inaccurate because RTX was not properly optimized as a baseline in cgRX, at least for the range query.

To optimize the RTX, this thesis explores multiple OptiX(Nvidia's Ray-tracing Software API) optimization strategies for RTX, including a revised range query algorithm, BVH partitioning, reverse mapping, and spatially closed query mapping. Additionally, the best configurations are applied to other baselines, including cgRX. All these improvements together are used to reproduce the experiments in cgRX.

The evaluation is first based on the impact of each optimization technique on RTX. These optimizations reduce RTX's memory usage during construction and improve range query performance. Then, cgRX, optimized RTX, and other baselines are compared using the same experimental setup as cgRX, all using their best configurations. The re-evaluated results differ significantly from those in cgRX.

In summary, this thesis contributes to RTX optimization by exploring the effects of multiple optimization techniques. The optimized RTX and baselines configured with optimized settings collectively aim to develop a high-performance GPU database index.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Acknowledgments

I am grateful to my advisor, Dr. Sean Chester. Over the past two years of his supervision, I have learned not only academic skills but also valuable life lessons.

Yijie Wu,
Victoria, Monday 14th April, 2025.

Chapter 1

Introduction

Join clauses are a crucial component in SQL-based relational databases, used to combine rows from two or more tables based on related columns. Efficient join execution is critical to query performance, especially for large-scale analytical workloads. Traditional join algorithms, such as hash join and sort-merge join, rely heavily on indexing techniques like B-trees [BM72], hash tables [Cor+09], and R-trees [Gut84] to accelerate join key matching by reducing the search space. These methods are widely adopted in CPU-based systems.

With the growing demand for high-throughput data processing, GPU-based join execution has gained significant attention due to the massively parallel nature and high memory bandwidth of modern GPUs [He+09]. Many traditional join strategies, such as hash joins on pre-partitioned data and sort-merge joins on sorted arrays, have been adapted for GPU architectures to exploit parallel execution and memory coalescing [Kal+12]. Additionally, GPU-specific optimizations—such as cooperative thread processing and warp-level synchronization—have enabled new designs that go beyond simple adaptations of CPU methods. These GPU-accelerated join implementations not only improve performance but also enable novel approaches to data access and indexing that better align with the GPU execution model.

Ray tracing is a fundamental technique in computer graphics, widely used to generate realistic images by simulating how light interacts with objects [Whi80]. Traditionally, ray tracing has been associated with rendering applications with the acceleration structure, such as bounding volume hierarchies (BVH) [Gla89], which optimize ray traversal through all objects in the 3D scene. With the introduction of ray-tacing cores(RT Cores) in NVIDIA GPUs, ray traversal and intersection testing have become significantly more efficient [Par+10], enabling real-time ray tracing for various applications beyond traditional rendering.

A recent study [HS23] explored a novel use case—leveraging ray-tracing for database indexing—by proposing **RT-Index (RTX)**, a ray-tracing-based index-

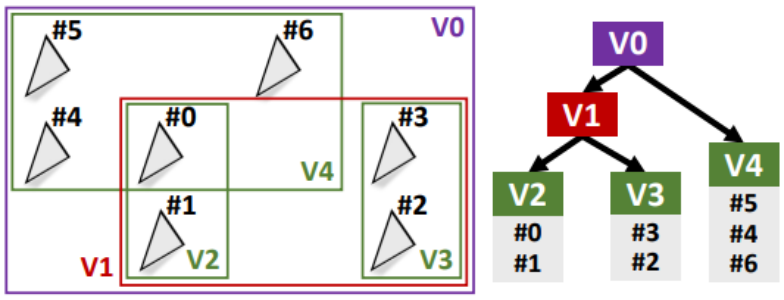

Figure 1.1: Exemplary bounding volume hierarchy for a (two-dimensional) triangle arrangement. Image by Justus Henneberg and Felix Schuhknecht (2023), source: `https://www.vldb.org/pvldb/vol16/p4268-schuhknecht.pdf`, licensed under CC BY-NC-ND 4.0 (`https://creativecommons.org/licenses/by-nc-nd/4.0/deed.en`).

RowId	Country	Code
0	Canada	10(#0)
1	USA	3(#1)
2	Mexico	2(#2)
3	France	5(#3)
4	Germany	14(#4)
5	UK	1(#5)
6	Germany	6(#6)

RowId	Capital City	Code
0	Ottawa	10
1	Washington DC	3
2	Mexico City	2

QueryId	Lower	Upper
0	1	3
1	2	5
2	3	8
.....	.....	.....
N-1	100	120
N	0	150

Figure 1.2: Left: The Table Corresponding to Figure 1.1. Middle: A Table to Illustrate Inner Join. Right: An Example of a Large Set of Range Queries

Results for Inner Join

RowId	Capital City	Country
0	Ottawa	Canada
1	Washington DC	USA
2	Mexico City	Mexico

Results for Range Queries

QueryId	Results(Code)
0	1,2,3
1	2,3,5
2	3,5,6
.....	.....
N-1	Null
N	1,2,3,5,6,10,14

Figure 1.3: Left: The Results to Inner Join in Figure 1.2. Right: The Results to The Range Queries in Figure 1.2

ing approach built on **NVIDIA's OptiX** package to structure data efficiently and minimize search space using ray traversal techniques.

What is database indexing? Database indexing is a technique that improves the query performance by organizing data into a structure like B+ trees or hash indexes, then, when performing a query, it allows the database to locate relevant data without scanning the entire table, therefore speeding up lookup operations. Figure 1.1 is an example of a BVH structure, which is also the database indexing used in the RTX paper. On the left: A spatial partitioning of triangles using bounding volumes. V0 is the root node, subdividing into V1 and V4. V1 further splits into V2 and V3, each containing specific triangles. On the right: A tree representation of the BVH structure.

Database indexing plays a crucial role in inner joins and a large set of range queries. An inner join combines rows from two tables based on a matching column, returning only matching records. A range query is a type of query that retrieves all records within a specified range of values. An example is shown in Figure 1.2. The *Code* column in the table on the left corresponds to the data in Figure 1.1. The middle table represents another relation in the database. When performing an inner join between the left table and the middle table on the *Code* column, the *Code* column in the middle table becomes a set of point queries that search the structure in Figure 1.1, returning the matched rows. The right table shows an example of a large set of range queries. Every range query has an upper bound and lower bound for the query in the *Code* column. The Figure 1.3 shows the results of the inner join and range queries.

The RTX study consists of three key contributions. First, a method was introduced to build a BVH with indexes. This process begins by mapping database indexes into 3D points; for example, key 12 will be converted to (12,0,0). These points are then used as the centroids of triangles, which serve as fundamental geometric primitives for constructing the BVH. The keys here are *unsigned int*; other data types will first be converted to *unsigned int*. Second, it proposes a technique to cast rays for point queries. The ray origin is determined by converting the query into a 3D point, following the same convention as above, but with a small numerical offset subtracted to ensure proper ray-object intersection. Similarly, the destination is computed by adding a numerical offset. The ray is then cast in the (0, 0, 1) direction along the Z-axis, for example, we want to cast a ray for point query 12, suppose the offset value is 0.5, Then, The ray origin will be (12-0.5, 0, 0) and the ray destination will be (12+0.5, 0, 0). The triangle/triangles(if the keys are duplicated) that are hit will be returned as results. Third, it extends this approach to cast rays for range queries. For range queries, the ray origin is set to the point of lower bound in the queried range, while the destination is set to the point of upper bound. All the triangles that are hit will be returned as the results.

This method presents an intriguing alternative to traditional database index structures and provides an opportunity to invent a better database index structure on GPU. However, the experimental results indicate that the RTX does not perform particularly well in practical scenarios. One major drawback is its large build size, which results in excessive memory consumption compared to traditional indexing methods. In addition, the build time is significantly longer. Finally, the range query performance is considerably slower. The last problem also happens in the case of the point query on duplicate keys as the ray also requires intersecting multiple triangles. Therefore, these two cases are considered to have the same performance issue. These results show that ray-tracing techniques face fundamental challenges in database indexing.

Various studies have demonstrated the effectiveness of ray tracing in accelerating queries, such as radius search, where sample points are mapped as spheres into the BVH, and rays are cast from each query point to detect intersecting spheres [Eva+21]. Another example is spatial join, where one set of geometric objects is mapped into a BVH, and each element of another set casts rays to determine intersections [GLZ24]. These ray-tracing-based acceleration techniques have shown significant performance improvements in query processing.

A follow-up study [Hen+24] called the **coarse granular ray tracing index (cgRX)** solved these problems by introducing several optimizations to the RTX approach. First, it significantly reduced the build size and build time by sorting the key set and then applying bucket compression on the keys. This technique works by evenly partitioning the key set into fixed-size buckets and mapping only the maximum key from each bucket to the BVH structure. Second, since the key set is already sorted, a range query only needs to intersect the first relevant triangle and return its position. From there, the search can proceed with a linear scan of the key set starting at that position rather than intersecting all triangles. This significantly reduces unnecessary computations due to the high cost of triangle intersection. According to the results presented in the study, this optimization led to a 1.5× speedup in range query performance, 2-3x smaller in build size compared to RTX. On the other hand, the study results show that RTX achieves better point lookup time, as cgRX requires an additional search within the bucket to locate the exact key. This extra step introduces overhead, making point lookups slightly slower compared to RTX.

The results for build size and point query lookup time presented above are expected, but is the fact that cgRX has a **better range query lookup time also expected**? In the study of cgRX, the RTX baseline uses the same implementation as in the study of RTX. Let us have one more look at the range query algorithms for RTX and cgRX. The cgRX is affected by the sorting of the key set, allowing it to be optimized so that it only intersects a single triangle before performing a linear scan over all qualified keys. Similarly, RTX can also

adopt this optimization to achieve better lookup time. To accurately analyze the performance difference between RTX and cgRX in range query lookups, the **same improvement should be applied to RTX before evaluation**.

This thesis **aims to develop a high-performance GPU database index**. We first revise the RTX range query algorithm to match the approach used in cgRX and then examine the performance of RTX and cgRX under the same conditions. Additionally, further **Optix optimization techniques** used in other ray-tracing-accelerated algorithms will be applied to RTX to examine their impact on build size, build time, and query lookup time.

How do we revise the range query algorithm for RTX? First, we need to ensure the key set to be sorted. Then, we revise the implementations of Optix Shader Programs. Currently, the ray-gen program defines the ray to use the any-hit shader program, meaning that every primitive intersected by the ray will trigger the shader program and return a result. To make the ray only hit the triangle with the smallest value within the range, we define the ray to apply for the closest-hit shader program, resulting in only the closest triangle to the ray origin, triggering the shader program and returning a result.

What else can we try to optimize the RTX? We do this by first trying a **BVH partitioning technique**. We partition the BVH into a number of equal-size sub-BVHs and try different numbers for sub-BVHs to find the optimal one. The key to our partitioning is to reduce the depth of the BVH, thereby improving the efficiency of ray traversal. We also try an **inverse mapping technique**. We compare the size of the build keys and probe set before building the index, then map the smaller side as indexes and use the other side as queries. The key is to first take advantage of the fact that firing rays in Optix can be exposed to more parallelism, and second, the BVH can always build with the smaller side. In addition, before building up the BVH, we **compress the duplicate elements** into a bucket for every unique key and map only 1 into 3D scene; this is because mapping multiple triangles at the same location in a 3D scene significantly increases the BVH complexity. By compression, we can ensure there is a maximum of one triangle at each location. We are also interested in how performance is affected with respect to **spatially-ordered query scheduling** such that adjacent rays follow similar BVH traversal paths. We do this by comparing the performance of the query set being sorted versus the query set being unsorted.

We will conduct a large number of targeted experiments. Our experiments are divided into **two stages**. In the first stage, we will individually test each optimization method to analyze its impact on RTX and compare the results with the original RTX. In the second stage, based on the findings from the first stage, we will apply all the effective optimization methods to RTX and conduct systematic experiments using the same experimental setup as the cgRX paper to obtain the final results. Additionally, we will also use the optimization settings

for all the other baselines.

Our experiments are conducted on a state-of-the-art GPU platform equipped with an **NVIDIA RTX 4060 Laptop GPU**, which offers strong computational performance and advanced ray tracing capabilities. If necessary, we will benchmark the most critical experimental results on an RTX 4090 for comparison.

Our experiments show a significantly different result compared to that in the cgRX study. The gap in build size for RTX has narrowed to only twice that of cgRX. Almost all methods show performance improvements for point queries and range queries under the optimized setting. The sorted array becomes the fastest method, and the optimized RTX outperforms cgRX in range queries and point queries with duplicated keys.

1.1 Contributions

This thesis's main focus is to explore a higher performance GPU database indexing technique and evaluate it using the baselines with the best settings. It achieves the following:

- A range query algorithm for RTX that intersects only a triangle with the smallest key value in the range.
- A BVH partition algorithm for RTX that partitions the BVH into some equal-size sub-BVHs to reduce the depth of the BVH.
- An optimal number for sub-BVHs will be determined by experimenting with different partition sizes and evaluating their impact on performance.
- Experiments demonstrating the impact of compressing identical keys will be conducted to analyze its effect on performance.
- Experiments demonstrating the impact of sorting the probe set will be conducted to evaluate its effect on query performance and traversal efficiency.
- Experiments demonstrating the impact of inverse mapping will be conducted to evaluate its effect on query performance.
- A comprehensive experiment will be conducted by applying all effective optimizations to RTX to evaluate its performance improvement.

1.2 Overview of following chapters

Chapter 2 provides a definition of index and two primary applications of the indexing in database query. Next, the difference between clustered index versus non-clustered index is provided. Then, an introduction to Nvidia's RT cores and OptiX ray-tracing package with its programming model and execution model are provided. Additionally, the baselines used in the cgRX paper will be briefly introduced.

In Chapter 3, a comprehensive discussion of the RTX and cgRX methods will cover both their construction and querying aspects. Additionally, several papers that have applied the OptiX optimization techniques mentioned in the introduction will be reviewed in terms of their effectiveness.

In Chapter 4, the revised query algorithm will be presented, along with a discussion on how each optimization technique is applied to RTX. In particular, the performance challenges encountered in inverse mapping and their corresponding solutions will be presented.

Chapter 5 covers the design of two experiments and the analysis of their results. The first experiment is an ablation study aimed at understanding the effect of each optimization technique on RTX performance. The second experiment treats the optimized RTX as a baseline and repeats the cgRX experiments to examine any differences in results.

To conclude the thesis, Chapter 6 provides a comparison between the results of cgRX and RTX, focusing on both similarities and differences along with an analysis of the underlying reasons. Additionally, it discusses RTX techniques that have shown promising results and could potentially be applied to cgRX in the future.

Chapter 2

Background

In this chapter, I will first explain what is an index and the application of indexing in database queries. Next, I will introduce clustered index and non-clustered index. Then, I will provide a detailed introduction to NVIDIA's OptiX Ray-Tracing Package, as this thesis's primary subject of analysis is based on this library. I will present OptiX from both a hardware and a software perspective.

- The hardware section focuses on CUDA cores and RT cores for accelerating ray-tracing algorithms, with a primary emphasis on RT cores.
- The software section covers the OptiX package, where I will explain its programming and execution models.

Following this, I will introduce the baselines compared in the cgRX work, including B+ tree, hash table, and sorted array.

2.1 Database Index and An Example

As mentioned in the introduction, a database index is a technique that improves query performance by organizing data into a structure like B+ trees or hash indexes. Indexes are typically created on columns that are frequently used in search conditions.

Here, the B+ tree will be used as an example to explain what an index looks like. A B+ Tree is a widely used data structure for database indexing. It is a self-balancing, multi-level tree. The concept is similar to the binary search tree but with the following differences: a B + tree allows each node to have multiple children, and its leaf nodes are linked. The visual structure is shown in Figure 2.1. Point queries traverse from the root to the corresponding leaf node, while range queries first locate the starting element and then directly traverse to the right through the linked leaf nodes.

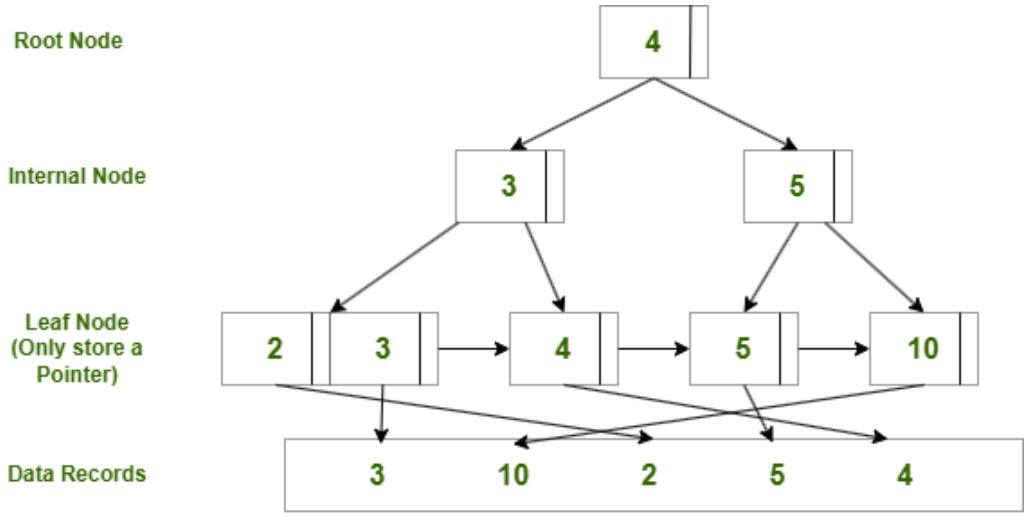

Figure 2.1: An Example of B+ Tree

2.2 Applications of Indexing for Database

In this section, I will discuss the application of indexing in database queries. Indexing plays a crucial role in optimizing query performance, primarily in two scenarios: join clause operations and the where clause of a query statement.

The **WHERE** clause in SQL is used to filter records based on specified conditions. The general syntax of a **WHERE** clause in a SQL statement is:

```
SELECT * FROM Customers WHERE age > 30;
```

Mathematically, this can be represented using relational algebra:

$$\sigma_{\text{age}>30}(\text{Customers})$$

The WHERE clause in SQL is closely related to index range queries, as indexed attributes help minimize the number of scanned records. When a range condition (e.g., column BETWEEN a AND b) is applied to an indexed column, the database engine can efficiently retrieve matching records using B-trees or other index structures, therefore improving query performance [TZB25]. An illustration of the above range query is shown in Figure 2.2, assume the table on the left is the one being queried by the range query, and the table on the right represents the returned results.

The **JOIN** clause in SQL is used to combine rows from two or more tables based on a column between them. The general syntax of a **JOIN** clause in a SQL statement is:

Table: Customers

customer_id	name	age
0	Bob	15
1	Lucy	45
2	David	33
3	Tom	31

Results

customer_id	name	age
1	Lucy	45
2	David	33
3	Tom	31

Figure 2.2: Illustration of Range Query

```
SELECT Customers.customer_id, Customers.name, Orders.order_id
FROM Customers
JOIN Orders ON Customers.customer_id = Orders.customer_id;
```

Mathematically, this can be represented using relational algebra:

$$Customers \bowtie_{Customers.customer\_id=Orders.customer\_id} Orders$$

The **JOIN** clause by default uses **INNER JOIN** clause, the result includes only the matching rows from both tables. Inner joins and point queries are closely related, as point queries are often utilized in indexed nested-loop joins to efficiently retrieve matching records from the joined table. The database engine can quickly locate specific rows using indexes instead of full table scans [Gra]. An illustration is shown in Figure 2.3, assume the above inner join query joins the left and middle tables, and then the table on the right represents the result.

Table: Customers

customer_id	name	age
0	Bob	15
1	Lucy	45
2	David	33
3	Tom	31

Table: Orders

customer_id	orderId	date
0	9905	2025-3-10
1	9906	2025-3-11
1	9907	2025-3-11
4	9908	2025-3-12

Results

customer_id	name	order_Id
0	Bob	9905
1	Lucy	9906
1	Lucy	9907

Figure 2.3: Illustration of Inner Join

2.3 Clustered Index vs Non-clustered Index

It is worth mentioning that there are **clustered index** and **non-clustered index**. A clustered index, which is shown in Figure 2.4, has the following characteristics:

First, it requires records to be sorted based on the indexed column; second, the leaf nodes also store the actual records. In a non-clustered index, which is shown in Figure 2.1, the records are unordered, and the leaf nodes store only pointers to the actual records. The main difference between the two is that a clustered index generally offers better performance. First, since a clustered index directly stores records, **retrieval is faster**. Second, because the records are sorted, range queries can **scan the record array** directly instead of scanning at the leaf node level, reducing the frequency of data retrieval, which is somewhat similar to the revised RTX range query method discussed later. Typically, a table will have a clustered index built on the primary key column, and the remaining columns will have non-clustered indexes if needed.

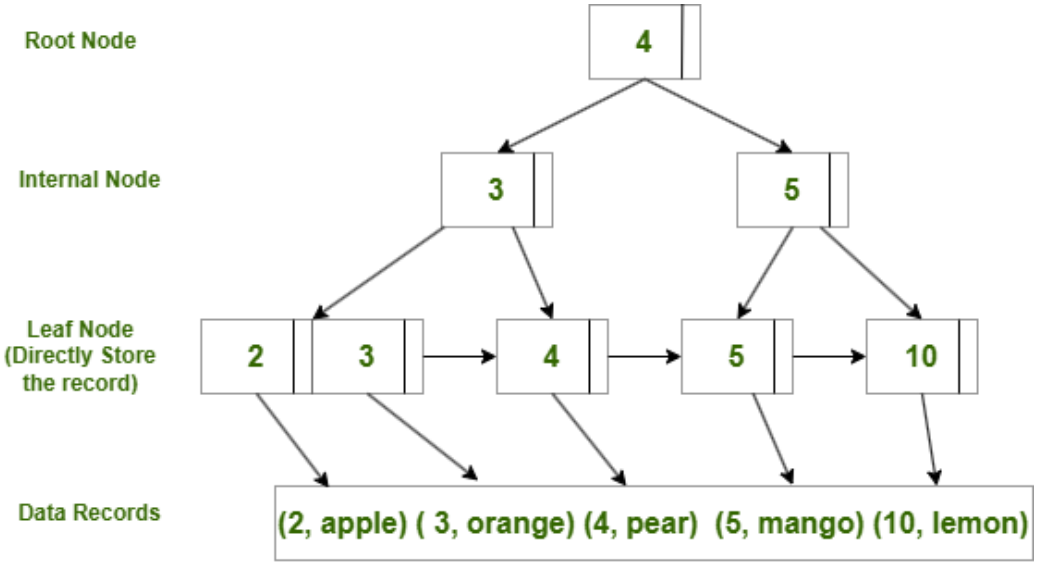

Figure 2.4: An Example of clustered B+ Tree

2.4 Nvidia's RT Cores and CUDA Cores

At the hardware level, RTX cards feature specialized Ray-tracing cores(RT cores) that have fixed functions [Wik25]. RT Cores on GeForce RTX GPUs provide dedicated hardware to accelerate BVH and ray/triangle intersection calculations, dramatically accelerating ray tracing [NVI19]. RT cores work as specialized accelerators alongside regular **CUDA Stream Multiprocessors (SMs)**. They use the same memory, so CUDA SMs handle general computing while RT cores handle ray tracing [Zhu22]. CUDA cores are parallel processing units in NVIDIA GPUs that execute thousands of threads simultaneously. The number of CUDA

cores varies by GPU model, ranging from a few hundred in entry-level GPUs to over 18,000 in high-end data center GPUs like the H100.

2.5 Nvidia's OptiX Ray-tracing Package

The OptiX package is the CUDA-based Ray-tracing API developed by Nvidia, and it natively supports RT cores. CUDA is a parallel computing library developed by NVIDIA that enables software to utilize CUDA cores for accelerating computations on GPUs. I will present an overall programming model and execution model here, the details can be found in the Programming Guide [NVI24].

OptiX Programming Model

In OptiX programming, the first step is the initialization of the context. The context manages one GPU and all the resources in this GPU. It is created using the device's CUDA context. The next step is to build a BVH, this stage executes on SMs using OptiX's built-in function, which is non-programmable. The next step is to build a pipeline. A pipeline defines the ray tracing workflow. OptiX presents a fixed pipeline organization but exposes interface for user-defined programs. The organization of the OptiX pipeline is shown in Figure 2.5. All the rectangles in the figure are OptiX programs. Green represents fixed function, and gray represents the user-defined program. The rectangle with both gray and green represents parts that are not customizable and parts that are customizable. The workflow starts with **Ray Generation** shader, which initiates the ray tracing process. The definition of a ray is shown in

$$P(t) = O + t\mathbf{d}, \quad t \in [t_{\min}, t_{\max}] \tag{2.1}$$

where:

- $P(t)$ is the position of the ray at parameter t.
- O is the ray origin.
- $\mathbf{d}$ is the ray direction vector.
- t is the parameter that determines a point along the ray.
- $t_{\min}$ and $t_{\max}$ define the valid range of t.

The rays then **traverse the 3D scene**, interacting with objects in the scene. Depending on the intersection results with triangles and user-defined parameters, different **shader programs** are triggered. The description is shown in 2.1.

Finally, additional calculations such as shading and lighting are performed in **callable functions**. It is worth mentioning that there are two types of callable functions: the direct callable, which performs some calculations and directly returns, and the continuation callable, which recursively traces rays, such as refracted light.

Table 2.1: Description of shader types in ray tracing.

Shader Type	**Description**
Closest-hit (CH)	Executes when the closest intersection along a ray is found.
Any-hit (AH)	Invoked for each intersection along a ray.
Intersection	Defines custom intersection tests for user-defined geometry.
Miss	Triggered when no intersection is found along the ray's path.

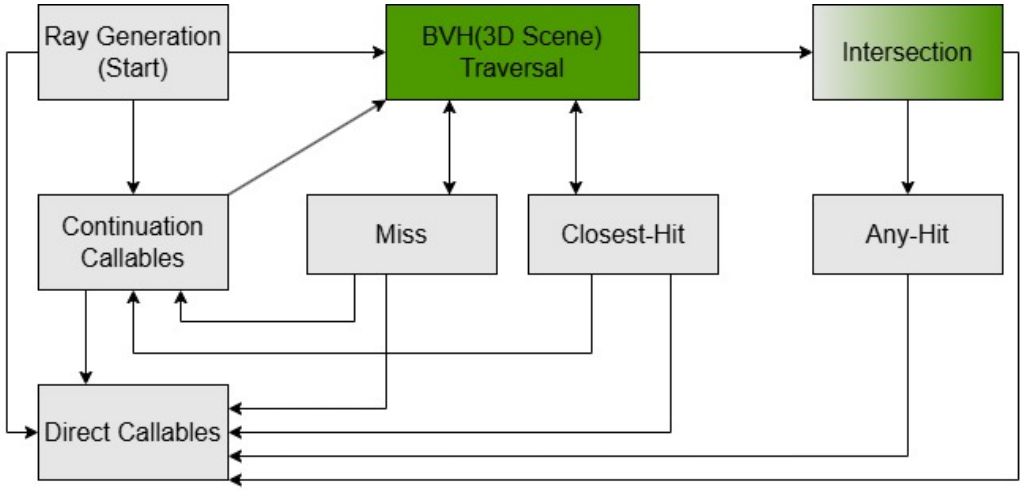

Figure 2.5: Optix Pipeline Organization

Optix Execution Model

OptiX utilizes the Single Instruction, Multiple Ray (SIMR) execution model, where a single instruction is applied to multiple rays in parallel. Every ray will be mapped into 1 CUDA thread and go through the pipeline organization in 2.5.

2.6 Other Baselines for Comparison

In this section, I will explain each of the baselines compared in the experiments.

2.6.1 GPU-based B+ Tree

The description of a B+ Tree is given in section 2.1. The cgRX references the GPU implementation of the B + tree from [APO23]. The implementation can be found in [Gro24]. In this GPU implementation, each key is inserted individually in a concurrent manner using warp-cooperative traversal and fine-grained locking. Similarly, queries also utilize warp-cooperative traversal, which enables parallel search. Warp-cooperative traversal is a technique used in GPU computing where multiple threads within a warp (typically 32 threads in NVIDIA GPUs) work together to traverse a data structure instead of each thread performing independent traversal. Fine-grained locking is a concurrency control mechanism that locks only small portions of a data structure instead of the entire structure.

2.6.2 GPU-based Sorted Array

A Sorted Array sorts the key set first and then uses binary search to optimize lookup speed. The cgRX paper uses **CUB's DeviceRadixSort** [NVI25] for sorting and performs a binary search for each query in parallel. CUB (CUDA UnBound) is a high-performance CUDA library providing GPU-optimized parallel primitives for sorting, scanning, reductions, and other operations.

2.6.3 GPU-based Hash Table

A hash table can use a hash function to map all keys into the hash table and then be used for database indexing. The cgRX paper references the GPU implementation of the hash table from [Jun+20]. This implementation uses batched insertion to optimize the index construction and employs warp-cooperative probing, where multiple threads within a warp work together to handle a single insert or query. It also uses efficient memory layouts and double hashing to optimize query performance.

Chapter 3

Related Work

This thesis explores better GPU database indexing through RTX optimization. It is crucial to revisit and understand the works of RTX and cgRX, which the thesis is based on. In the Introduction, we briefly mention the construction of RTX, the ideas behind point query and range query, and the optimization methods of cgRX. In this section, I will provide a detailed explanation of these methods. At the same time, all the OptiX optimization methods mentioned in the Introduction have also been explored in some related papers. In this section, I will briefly introduce how these optimization methods have been applied in the relevant papers and their effects.

3.1 Ray-tracing Index(RTX)

In both this work [HS23] and the cgRX work [Hen+24], the chosen data types are **32-bit** unsigned int or **64-bit** unsigned int, as most primitive data types can be converted into these two formats, but certain cases (e.g., floating-point numbers with offsets) may require special handling.

Overall, RTX introduces the concept of converting 32-bit and 64-bit keys into OptiX primitives that only support 32-bit floating-point values. The method involves splitting the 32-bit or 64-bit key into two or three smaller segments from the middle bits and mapping them to the x, y, and z coordinates of the point. On top of this, RTX develops algorithms for casting rays for point queries and range queries. The point query algorithm maps the query point into a 3D point in the same way, then subtracts an epsilon value to set the ray's start point and adds an epsilon value to set the ray's endpoint. For range queries, the lower and upper bounds of the query are transformed into 3D points, with the lower bound subtracting an epsilon value for the start point and the upper bound adding an epsilon value for the endpoint. All intersecting triangles are then returned as the

result.

During the build phase, as mentioned in the introduction, RTX will convert the data in the key set to 3D points and then use these 3D points as the centers to create triangles by adding an eps to the x-axis, y-axis, and z-axis of this point. The conversion from the data to 3D points is slightly different from directly casting and has to follow the conversion rules in the following equation(for 64-bit keys):

$$x = k_{22:0}, y = k_{45:23}, z = k_{63:46}$$

Where:

- x, y, z: coordinate of the 3D point
- k: key
- 22:0: Bits 0-22 of the key (22:0)
- 45:23: Bits 23-45 of the key (23:45)
- 63:46: Bits 46-63 of the key (46:63)

The equation shows how a 64-bit key is converted to the 3D point; for a 32-bit key, only x and y are needed for mapping 32-bit. Why can't we directly map to the x-coordinate? In OptiX, primitives such as triangles only support 32-bit floating point types, making it impossible to directly map a 64-bit key. In a 32-bit floating-point number, there is 1 sign bit, 8 exponent bits, and 23 mantissa bits, so the effective precision is 23 bits. Therefore, even if a 32-bit key can be converted into a 32-bit float type, the order of the data in the 3D scene cannot be guaranteed, which means the correctness of the range query results cannot be ensured. The conversion rules for the 32-bit key are in the following equation:

$$x = k_{22:0}, y = k_{32:23}, z = 0$$

Where:

- x, y, z: coordinate of the 3D point
- k: key
- 22:0: Bits 0-22 of the key (22:0)
- 32:23: Bits 23-32 of the key (23:45)

After conversion, the triangle buffer will be passed into OptiX's built-in BVH construction function, *optixAccelBuild*, followed by another call to *optixAccelBuild* to compress the BVH. To maintain the same order as the key array, the order of the vertex buffer is kept identical to that of the key array. After all the triangles are mapped to the BVH, each triangle is assigned a unique ID called primitiveID, which corresponds to the index of that key in both the key array and the vertex buffer. The build phase ends here.

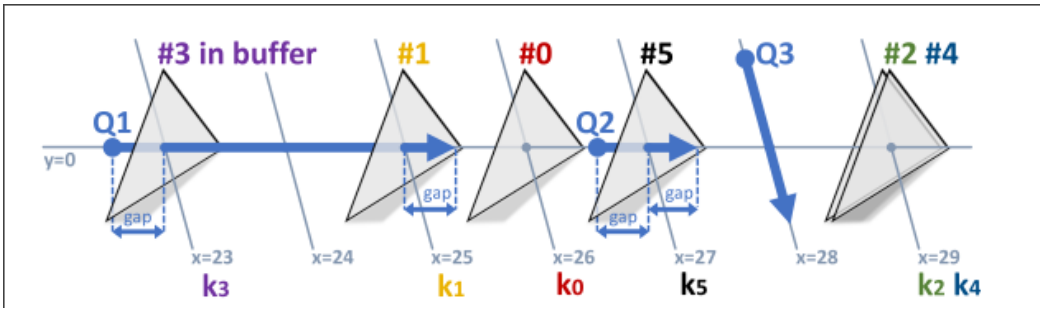

Figure 3.1: Key Array and RTX Query Explanation. Image by Justus Henneberg and Felix Schuhknecht (2023), source: `https://www.vldb.org/pvldb/vol16/p4268-schuhknecht.pdf`, licensed under CC BY-NC-ND 4.0 (`https://creativecommons.org/licenses/by-nc-nd/4.0/deed.en`).

The second phase is to perform queries by casting rays. OptiX uses the **Single Instruction Multiple Ray (SIMR)** execution model. Each query will be mapped to a single ray, therefore, multiple queries can be executed concurrently. The definition of each ray is shown in Figure 3.1. As illustrated in Chapter 2, to fire a ray, it requires the information of origin coordinate, direction, tmin, and tmax. Before firing a ray for each query, the query will be converted to a 3D point in the same convention as that in the build phase. The x-coordinate of this point will be reduced by a small epsilon value and then used as the origin of the ray for both the point query and range query. The destinations are slightly different. For a point query, the destination is the 3D point converted from the query value plus a small epsilon value. For a range query, which has a lower bound and an upper bound, the destination is the 3D point converted from the upper bound plus a small epsilon value. After obtaining this information, the ray is defined as follows: the origin is used as the ray's origin and tmin, the destination is used as tmax, the direction is (0, 0, 1), and all intersecting triangles in between have their indices in the buffer returned, which also correspond to the indices in the key array. The range query may span more than one x-row. In this case, an additional ray will be cast for each subsequent row. If the range spans X rows, this range query requires X rays. The Figure 3.1 provides a visual explanation. Q1 is an example of a range query. Q2 and Q3 are both examples of point query. Q3 is

an alternative way to emit a point query ray, where a vertical ray is cast from the top. The pseudocode for point query and range query is shown in Algorithm 1 and Algorithm 2. The difference is that there is no *queryUpperBounds* in case of the point query.

The results [HS23] show that RTX significantly exceeds other baselines in both build size and build time, at least 4× the size and construction time of the baselines. Point query performance is decent, only slightly slower than that of the hash table and B+ tree. However, range query performance is suboptimal, with query time twice that of the B+ tree and 1.5× that of the sorted array.

Algorithm 1 Original RTX Point Query Algorithm

Input: keySet, keySize, BVH, queryLowerBounds, querySize, resultBuffer (length=querySize, initialized to 0) **Output:** resultBuffer

```
1: for each query q at index i do in parallel
2:     traceRay(queryLowerBounds[i], queryLowerBounds[i], BVH)
3:     if intersection with a triangle then
4:         trianglePrimitiveId ← AnyhitShader()
5:         resultBuffer[i] ← trianglePrimitiveId + resultBuffer[i]
6:     else
7:         resultBuffer[i] ← not_found
8:     end if
9: end for
```

Algorithm 2 Original RTX Range Query Algorithm

Input: keySet, keySize, BVH, queryLowerBounds, queryUpperBounds, querySize, resultBuffer (length=querySize, initialized to 0) **Output:** resultBuffer

```
1: for each query q at index i do in parallel
2:     traceRay(queryLowerBounds[i], queryUpperBounds[i], BVH)
3:     if intersection with a triangle then
4:         trianglePrimitiveId ← AnyhitShader()
5:         resultBuffer[i] ← trianglePrimitiveId + resultBuffer[i]
6:     else
7:         resultBuffer[i] ← not_found
8:     end if
9: end for
```

3.2 Coarse-granular Index(cgRX)

The Coarse-guanular Index [Hen+24] is a follow-up work to improve the build size, build time, and range query performance.

Overall, cgRX applies the concept of bucket compression. Before constructing the BVH, the data is divided into equally sized buckets, and only the last key of each bucket is mapped to the BVH. This significantly reduces the size of the BVH and the construction time. However, the tradeoff is the additional work required to find the corresponding triangles within each bucket, and the key array must be sorted. Leveraging the sorted key array, cgRX modifies the range query algorithm to only intersect the first triangle in the range, then perform a linear scan in the key array. In contrast, RTX requires intersecting all the triangles in the range.

The build phase of the coarse-granular index requires the key array to be sorted. During the build phase, the key array will be divided into buckets of equal size, and only the last element in each bucket, which is also the largest key in the bucket will be mapped into BVH. An extra buffer is used to keep the key array and the position of the first element in each bucket. By this bucket representation, the number of triangles that need to be mapped is approximately only 1/bucket size of the original. When there are bucket representatives from more than one bucket that are duplicated, only the first one will be mapped. This significantly reduces the BVH size and build time. The triangle conversion and specific construction method are exactly the same as in RTX.

After compression, cgRX needs to modify the previous point query and range query algorithms in RTX. The origin of the point query remains the same as in RTX, but the destination needs to use the maximum key value. This is because the point query needs to find the first bucket representation after the query point. In extreme cases, this bucket could be located at the maximum key value position or may not exist at all. The new method ensures the correctness of the results, but in extreme cases, where a ray needs to be cast for every x-row, performance may significantly degrade. To address this issue, two methods were introduced into the tracing algorithm.

The first method is to map two new types of triangles in 3D: one called the row marker and the other called the plane marker as shown in Figure 3.2. For every plane, a row marker is placed at every entry in the column where $x = -1$, provided that the corresponding row contains at least one triangle. A plane marker is placed at each entry along the z-axis when $x = -1$ and $y = -1$, provided that there is at least one triangle in that plane. The new point query algorithm works as follows: If no triangle is intersected in the row where the query point is located, a ray is cast along the row marker's column within the current plane to find the nearest row containing triangles. If no such row is

Figure 3.2: Row Marker and Plane Marker Illustration. Image by Justus Henneberg, Felix Schuhknecht, Rosina Kharal and Trevor Brown (2024), Copyright © 2025, IEEE

found, a ray is then cast along the plane marker's z-axis to locate the nearest plane with triangles. After identifying the plane, the algorithm proceeds to find the row and, finally, the nearest triangle. In this way, in the worst-case scenario, only five rays need to be cast, reducing performance loss.

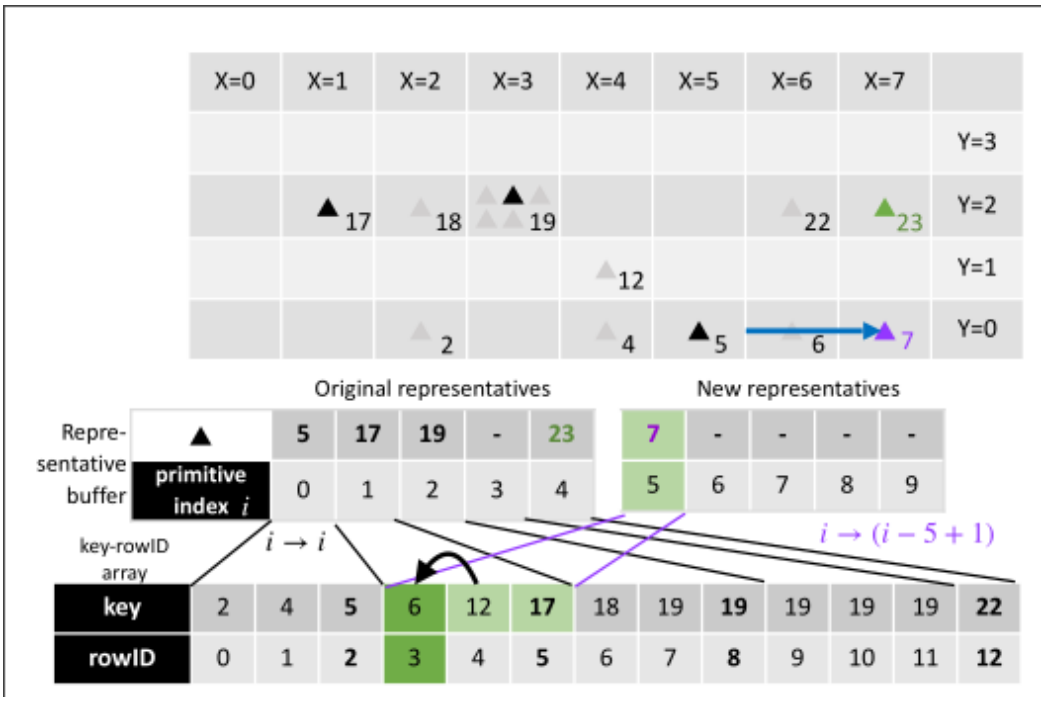

Figure 3.3: Extra Representative Illustration. Image by Justus Henneberg, Felix Schuhknecht, Rosina Kharal and Trevor Brown, Copyright © 2025, IEEE

The second method is to map extra representatives, as shown in Figure 3.3. An extra representative will be placed at the end of the row in two cases. The first case is that if the last triangle in this row is not a bucket representative, an extra representative corresponding to the next bucket will be mapped at the end of this row, for example, in the first row in the Figure 3.3, triangle 6 is not a bucket representative and is the last triangle in the row, so a new representative triangle 7 is mapped at the end of the row. The triangle 7 corresponds to the same bucket the triangle 17 represents. The second case is that if the last triangle in this plane is not a bucket representative, a new representative will be placed at the end of the plane to indicate the next plane's bucket representative for this triangle. For example, in the plane shown in the figure, the last triangle is 22, which is not a bucket representative. Therefore, triangle 23 is placed at the end of the plane, representing the bucket of 22, which is mapped in the next plane.

The advantage of this approach is that if we need to look up key 6, instead of firing three rays as before, now we only need to fire a single ray. Additionally, this reduces the need to map an extra plane marker.

With the new point query algorithm explained, the range query follows a different approach. Given that the key set is sorted during construction, the point query lookup algorithm helps locate the lower bound. From this position, a linear scan is performed until the first out-of-range element is found.

3.3 BVH Partitioning

[Zhu22] has tried the BVH Partitoning method in *Accelerating Neighbor Search Using Hardware Ray Tracing* which uses RT cores and ray-tracing to accelerate the nearest neighbor search. BVH partitioning is a technique that divides the data into smaller subsets based on the query's required search range and constructs smaller BVH structures for each subset instead of building 1 large BVH. This approach helps improve query efficiency by focusing computations only on relevant regions. They adjusted the BVH size with different parameters and found that the BVH size is directly proportional to probe time. They concluded from the experimental results that search time is strongly correlated with AABB size, as it determines the workload on both the SMs and RT cores. Reducing the AABB size decreases the search time.

3.4 Spatial-closed Query Mapping

[Zhu22] has tried Spatial-closed Query Mapping in the same work. Spatial-closed Query Mapping is effective because OptiX's execution model is closely related to CUDA's execution model as Optix is built upon CUDA. In CUDA, a warp (typically consisting of 32 threads) is the smallest execution unit for an SM. All threads within a warp execute instructions simultaneously and must complete execution before the warp can finish. Branch divergence is a typical performance issue in CUDA programming when threads within a warp take different execution paths due to conditional branching. In this context, conditional branching occurs during the ray traversal of the BVH. When the rays have different paths, the execution time of the warp is determined by the thread or ray with the longest path in the warp. Spatial-closed Query Mapping is a method that groups queries with the same path together for execution, aiming to minimize branch divergence as much as possible.

They tested ordered and unordered pair queries/rays separately and found that unordered pairs were five times slower than ordered pairs. From this ex-

periment, they concluded that search performance is sensitive to ray coherence, which can be improved by mapping queries to rays so that adjacent rays represent spatially close queries. They also mentioned that Spatial-Closed Query Mapping can improve performance by up to approximately 4× faster.

3.5 Inverse Mapping

[Eva+21] has tried Inverse Mapping in *Fast Radius Search Exploiting Ray-Tracing Frameworks* which is another early work that uses RT cores and Ray-tracing to accelerate the nearest neighbor search. The standard approach would be to map the sample set to the 3D scene and then cast rays from each query point to check whether each key falls within the radius. However, they used an inverse mapping approach, where the query set was mapped to the 3D scene, and rays were cast from each key point to determine whether the key belonged to any query. An illustration is shown in Figure 3.4, On the left is the original mapping, where samples are mapped to the BVH, and queries shoot rays to check if they fall within the range of the samples. In the middle, the query is mapped to the BVH, and sample points are used to shoot rays to check if they are within a given query. On the left is a rejection step, where, when the search radius is not constant, additional checks are needed to determine if certain sample points fall within the maximum radius but are not part of certain queries.

They mentioned that in their data, the number of keys is greater than the number of queries, and inverse mapping can enhance OptiX's parallelism. The SIMR execution model helps expose more parallelism by having a larger number of rays execute fewer instructions, achieved by traversing a shorter BVH. This method resulted in a related good performance improvement.

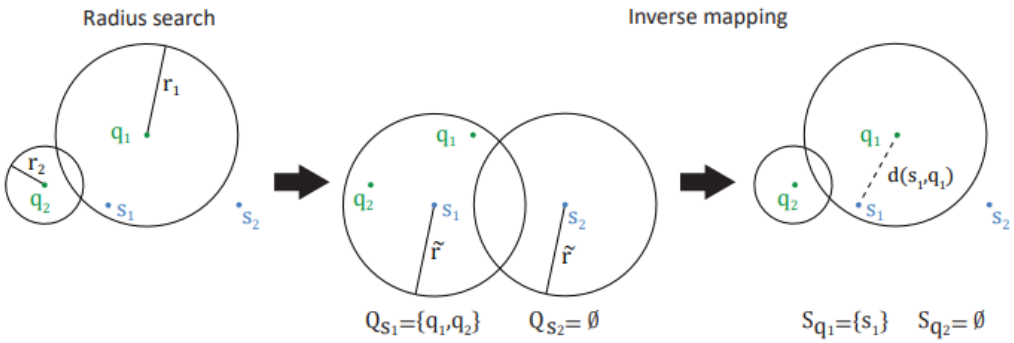

Figure 3.4: An illustration for Inverse Mapping. Image by I. Evangelou, G. Papaioannou, K. Vardis, and A. A. Vasilakis (2021), source: `https://jcgt.org/published/0010/01/02/`, licensed under CC BY-ND 3.0 (`https://creativecommons.org/licenses/by-nd/3.0/deed.en`).

Chapter 4

Proposed Optimization Solutions

In this chapter of my thesis, I will present how I modified the RTX range query algorithm and how the optimization techniques were applied to RTX. Spatially-closed query mapping will not be explained here, as it only requires sorting the query set. The revised range query algorithm, BVH partitioning, and inverse mapping will be explained in this section.

4.1 A Performance-Driven RTX Range Query Algorithm

In this section, I present the revised RTX Range Query Algorithm. We assume that the key set is already sorted. The algorithm is divided into **two parts**: the first part involves modifications to the original RTX algorithm as shown in Algorithm 3. The major difference between the original RTX range query algorithm and the revised one is that the **Any-hit shader** is replaced by the **Closest-hit shader**, and a **linear scan** is added after the tracing. The linear scan is a CUDA kernel as shown in Algorithm 4, it uses a CUDA thread to process a single query. For each query, if a triangle is hit, we locate its position in the key set, scan for all keys that fall within the range, and compute the final result. By modifying it in this way, the range query will behave like cgRX, ensuring that at most one triangle is intersected. The visual illustration is shown in Figure 4.1 and Figure 4.2. In Figure 4.1, Originally, only two steps were needed: first, intersecting all triangles and computing the final result; second, writing the result after the ray terminates. In Figure 4.2, after the improvement, the ray stops after intersecting the nearest triangle in the first step, the second step directly scans the key array, and the third step stores the result after the scan is complete.

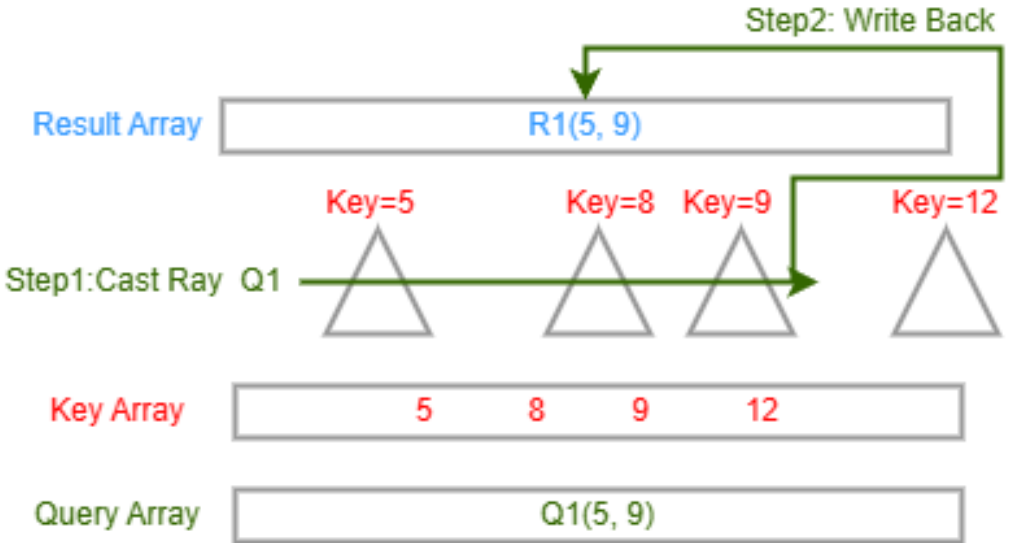

Figure 4.1: Original Range Query Algorithm Illustration

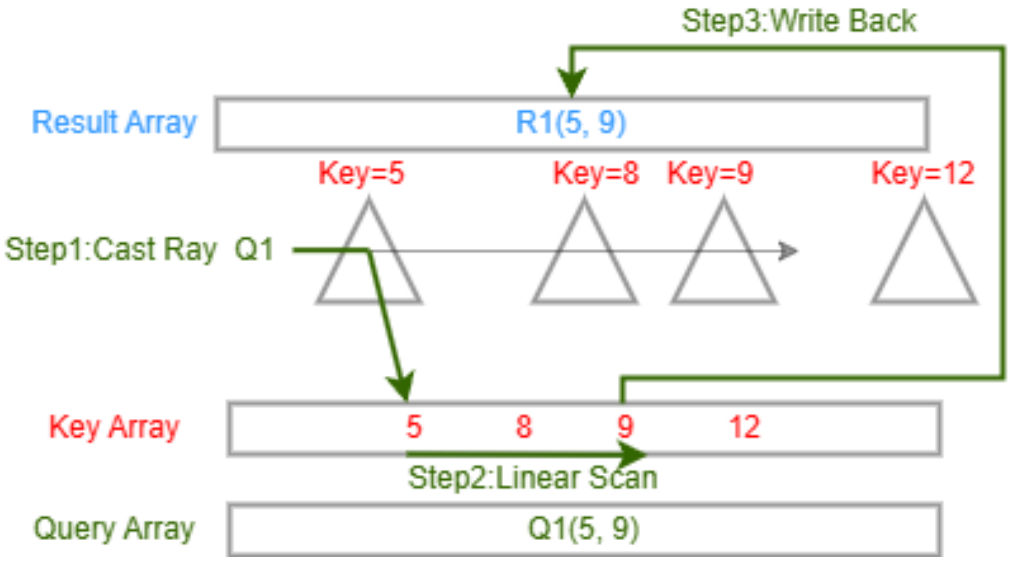

Figure 4.2: Revised Range Query Algorithm Illustration

Algorithm 3 Revised RTX Range Query Algorithm

Input: keySet, keySize, BVH, queryLowerBounds, queryUpperBounds, querySize, resultBuffer (length=querySize, initialized to 0) **Output:** resultBuffer

for each query q at index i **do in parallel**
    traceRay(queryLowerBounds[i], queryUpperBounds[i], BVH)
    if intersection with a triangle **then**
        trianglePrimitiveId $\leftarrow$ ClosesthitShader()
        resultBuffer[i] $\leftarrow$ trianglePrimitiveId
    else
        resultBuffer[i] $\leftarrow$ not_found
    end if
end for
resultBuffer $\leftarrow$ linearScan(keySet, keySize, queryLowerBounds, queryUpperBounds, querySize, resultBuffer)

Algorithm 4 CUDA Kernel Linear Scan

Input: keySet, keySize, queryLowerBounds, queryUpperBounds, querySize, resultBuffer **Output:** resultBuffer with final output **Overview:** This CUDA kernel runs in parallel across multiple threads to scan and accumulate indices of elements that fall within specified query bounds.

Step 1: Initialize and check thread index
Set the result to zero.
Compute the thread's global index.
If the index exceeds querySize, exit.
Step 2: Retrieve query range
Load the initial position from `resultBuffer`.
If the position is not found, store zero and exit.
Step 3: Load query bounds
Retrieve the lower and upper bounds for the query.
Step 4: Perform linear scan
Iterate through `keySet` starting from the initial position.
If the key is within bounds, accumulate its index.
If the key is out of range, stop the search early.
Step 5: Store the result
Write the accumulated result back to `resultBuffer`.
End

4.2 RTX BVH Partitioning Algorithm with Duplicated Key Compression

In this section, I will demonstrate how I specifically partition the BVH into **multiple sub-BVHs**. Before construction, we added a compression step: storing all unique values in a new buffer and using another buffer to record the first occurrence index of each unique value. This step takes extra time, so how do we do the compression effectively? We use a built-in function called *thrust::unique_by_key* in the *CUDA thrust API*. This function takes a *thrust::device_vector* and a thrust::sequence that stores all the indices of the vector and returns an array that stores the unique value and an array that stores the corresponding indices. This built-in function has already been optimized, which makes it very efficient. The case of point query on duplicate elements can also be solved by duplicated key compression technique. As mentioned in the introduction, the point query on duplicate element essentially has the same performance issue as the range query. As shown in Algorithm 1, it requires to intersection all the triangles that build from the same key. After the compression, the algorithm 1 intersects only one triangle. A subsequent linear scan in Algorithm 4 then completes the process.

During the build, originally, the data in the key set were converted into triangles and constructed into a BVH in a single step. I divided the data into **x equal parts** and then serially constructed a separate sub-BVH for each part. Once the sub-BVHs were built, they were stored in an **IAS (Instance Acceleration Structure)**, which is a BVH container that is used to hold multiple BVHs. The algorithm is shown in Algorithm 5. The visual illustration is shown in Figure 4.3, structurally different from the original BVH shown in Figure 1.1. In Figure 4.3, the maximum depth of the BVH is 3, while in Figure 4.3, the maximum depth is 2.

This method differs significantly from the construction approach in cgRX in Section 3.2. In cgRX, data is compressed into buckets before construction, which causes the query algorithm to require an additional step to search through the buckets after firing the rays. This is essentially a tradeoff, where the time spent searching the buckets is balanced against the query time. In contrast, the BVH partition here directly changes the construction method, so the query algorithm only needs to fire rays, without any extra lookup step.

4.3 Inverse Mapping Algorithm for Point Query

In this section, I will present the implementation of inverse mapping for point query. Inverse mapping assumes that the query set is sorted. It maps the query set to the BVH while using the key set for ray tracing. When a key intersects with

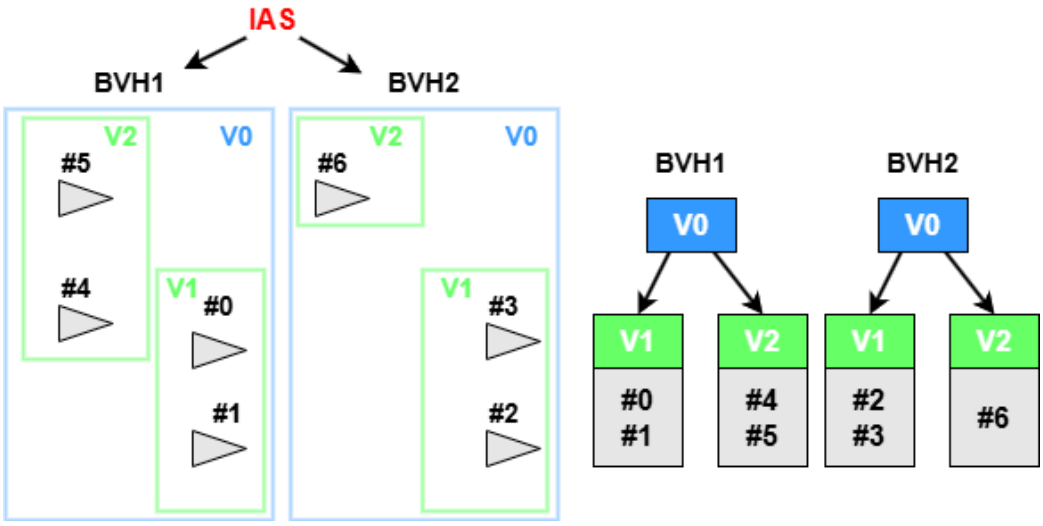

Figure 4.3: BVH Partitioning Illustration

Algorithm 5 Sub-BVH Construction and IAS Management

Input: compressedKeySet, compressedSize, numParts
Output: IAS handle

numParts ← 2^0 ▷ Set number of partitions
partSize ← ⌈compressedSize / numParts⌉
traversableHandles ← [ ]
for i ← 0 to numParts - 1 **do**
    offset ← $i\times$ partSize
    currentSize ← min(partSize, compressedSize - offset)
    primitiveBuffer ← ConvertKeysToPrimitives(compressedKeySet[offset], currentSize)
    traversableHandles[i] ← BuildTraversable(primitiveBuffer)
end for
instances ← CreateInstances(traversableHandles)
IASHandle ← BuildIAS(instances)
return IASHandle

a query, the result of the key is added to the corresponding query. Since queries can be **duplicated**, we apply a similar approach to Algorithm 3: after locating the intersected triangle, we perform a linear scan to collect all matching results. The two differences are that the query set and key set are **swapped** and the *queryUpperBounds* is not needed in ray definition for point query. Since the key set and query size are different, we need a **temporary array** called *resultReverse* of the same size as the key set to store the intersection results of each ray and triangle. The linear scan shown in Algorithm 6 is slightly different from Algorithm 4. In this kernel, we launch *keySize* threads, with each thread handling a value in *resultReverse*. This value represents the first corresponding query, we then start scanning from this position and add the key's result to all matching queries. An extreme case is that keys may have duplicates, causing multiple keys to simultaneously add values to the same query. In CUDA programming, this is called a **race condition**. To avoid it, atomic operations are needed when updating values. This also results in frequent **global memory access**, significantly reducing performance. The visual illustrations for before and after using inverse mapping are shown in Figure 4.4 and Figure 4.5. In Figure 4.4, originally, the keys were mapped to the 3D scene, and the queries were used to cast rays, which then intersected with triangles and returned the result. In Figure 4.5, the queries are mapped to 3D scene while the keys are used to cast rays. The intersected triangles are stored in *Result Array*, but the results in Result Array are the indices of the queries, so an additional step is needed to use the *Result Array*'s output to compute the final result in the *Key Array*.

To address this, I modified the approach: each thread first checks whether the previous thread processed the same key. If so, it simply returns. This ensures that only one thread handles each unique key. That thread then accumulates the results of all subsequent identical keys and sequentially adds them to the corresponding queries. This reduces both atomic operations and global memory access.

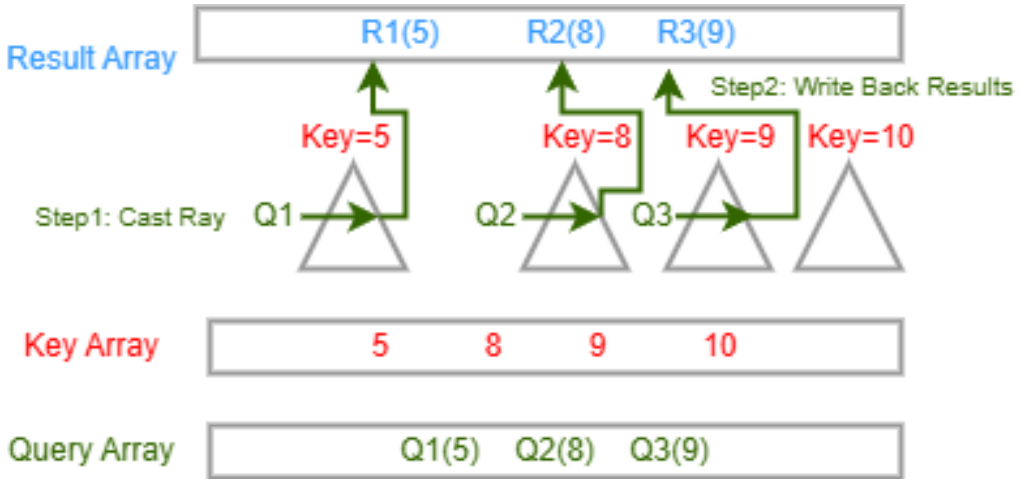

Figure 4.4: Illustration for Point Query before Inverse Mapping

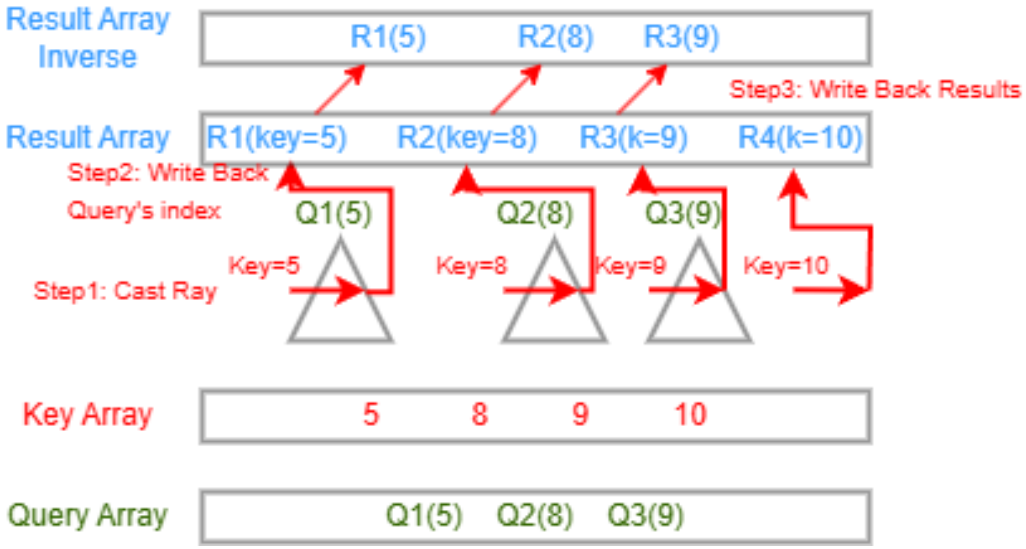

Figure 4.5: Illustration for Point Query after Inverse Mapping

Algorithm 6 High-Level Explanation of Collaborative Scan Kernel (Inverse Mapping)

Input: queryLower, queryUpper, keys, resultBuffer, resultReverseBuffer, buildSize

Output: resultReverseBuffer

Overview: This algorithm runs in parallel on a GPU to process query ranges and map them back to their original positions in a dataset. Each thread independently works on a portion of the input data.

1: **Step 1: Compute thread index**
2: Each thread calculates its global index ix.
3: If $ix \geq$ `buildSize`, exit.
4: **Step 2: Load key and precomputed bound**
5: Fetch the key from `queryLower[ix]`.
6: Retrieve the mapped index from `result[ix]`.
7: If the key is not found, exit.
8: **Step 3: Handle duplicate keys**
9: Initialize sum of indices.
10: If the previous key in `queryUpper` matches, exit.
11: **Step 4: Aggregate consecutive keys**
12: Scan forward while the next key remains the same.
13: Accumulate index values.
14: **Step 5: Write results to output**
15: Store the computed sum in `resultReverse`.
16: Increment bound index until all matches are processed.
17: **End**

Chapter 5

Empirical validation

5.1 Chapter Outline

In this chapter, I shall explain the generation methods of the key set and probe set. In each experimental section, I shall explain which **baseline** RTX is compared against and detail the parameter settings used. I shall explain how we verify the correctness of the results. For reproducing these experiments, if required, I shall also provide details on the experimental environment that was used for running these simulations. Additionally, I note that point queries often arise in join clauses, where a specific key must be matched against an index, while range queries are commonly associated with `WHERE` clauses that filter data within a specified interval.

5.2 Inputs

5.2.1 Input Parameters

The **input generation method** accepts user-defined parameters to generate the corresponding **data distribution**.

In a real database context, data distributions can exhibit various characteristics. For example, in a given dataset, the most common distribution might be a uniform distribution, where there are almost no duplicate keys, and the keys are evenly distributed. Alternatively, each key might have multiple repeated values, or only a subset of keys may have duplicates, which is referred to as a skewed distribution. In some cases, keys may be highly scattered, which can also be considered a uniformed distribution.

Similarly, queries can also exhibit different distributions. For example, most queries might hit many duplicate keys, or most queries might result in misses.

For range queries, the coverage of the range could be very large, or the number of matching keys within the query range could be high. The following parameters are used to generate different data distributions to evaluate the performance of the index under various data distributions.

Point Query Input Parameters

The point query input generation method supports the following parameters:

- build_key_uniformity_percent_options: Determines the uniformity of key distribution during index construction. An example of distribution with build_key_uniformity_percent_options = 100% is shown in Figure 5.1.
- misses_percent_options: Controls the percentage of queries that do not match any key. An example of distribution with misses_percent_options = 50% is shown in Figure 5.2.
- probe_zipf_coefficient_options: Specifies the Zipf coefficient for generating skewed probe distributions. An example of distribution with probe_zipf_coefficient_options = 1 is shown in Figure 5.3.
- log_key_multiplicity_options: Defines the logarithm of key multiplicity. Key multiplicity describes how often the same key is repeated in a table. This variable can be used to generate data distributions with many duplicate keys. An example of distribution with log_key_multiplicity_options = 10 is shown in Figure 5.4.

Range Query Input Parameters

The range query input generation method supports the following parameters:

- log_key_range_multiplier: Log key range multiplier adjusts the scale of the key range using a logarithmic factor, which can control the size of the range between the lower and upper bounds of the query.
- log_expected_hits: Log expected hit represents the logarithmic value of the expected number of matches for a range query. It influences the query's search behavior by indicating how many records match the specified range.

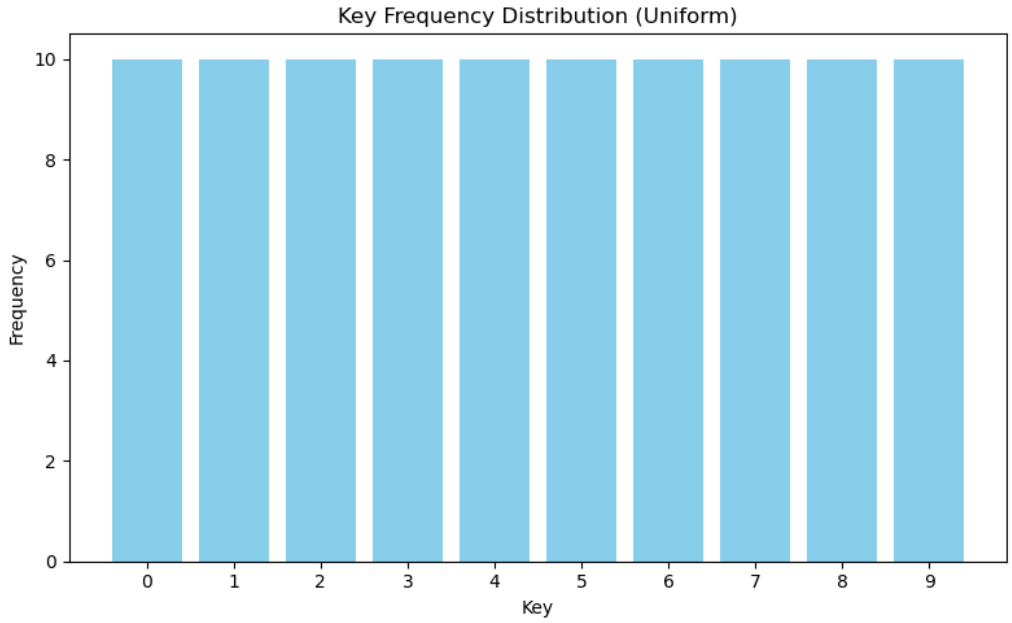

Figure 5.1: Uniform Distribution

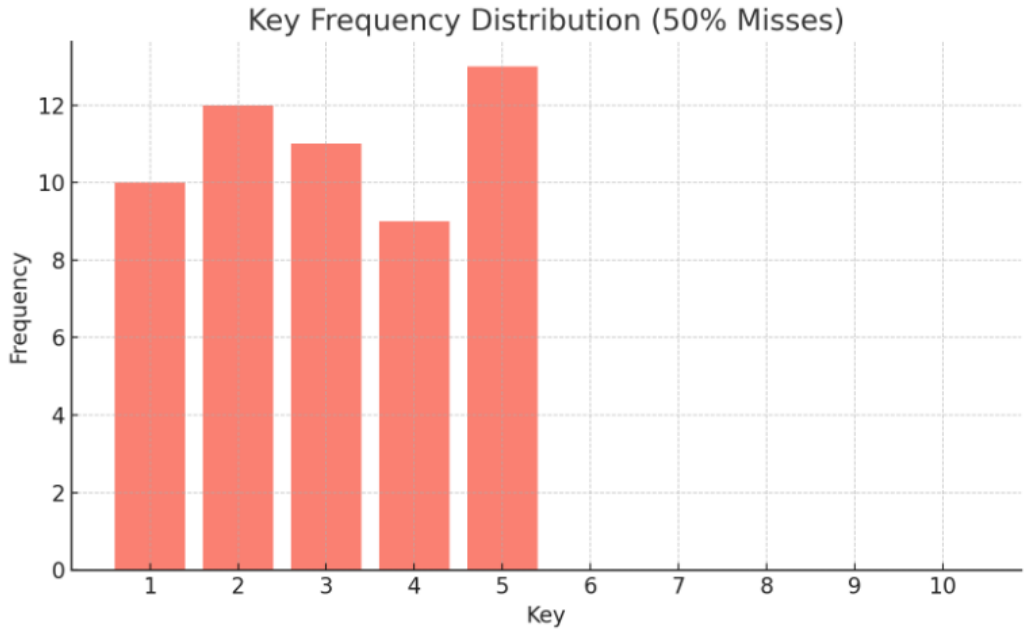

Figure 5.2: 50% Miss Distribution

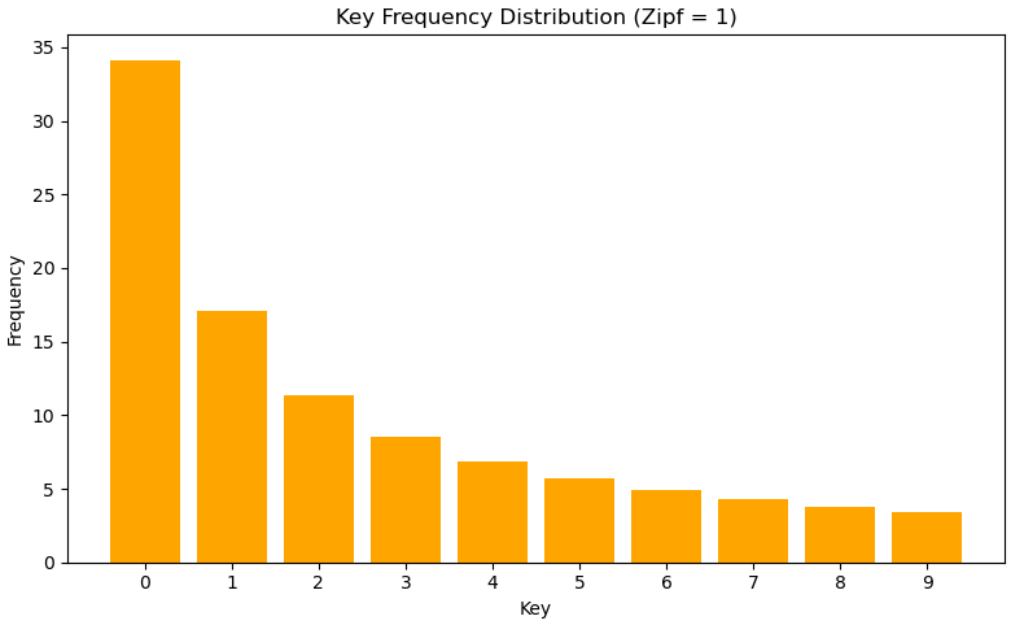

Figure 5.3: Zipf=1 Skewed Distribution

5.2.2 Input Generation Methods

This section covers the input generation methods on which the proposed methods are evaluated. This method is mainly divided into two types: one for point query input generation and the other for range query input generation. The numbers will be generated using the Random library in C++ with a random seed of 42 to reproduce the result. Also, the input supports 2 types of keys, 32-bit unsigned int and 64-bit unsigned int. The data generation method and seed count are exactly the same for both RTX and cgRX.

Point Query Input Generation Method

The point query input method first generates a user-defined number of key sets based on the input values of build_key_uniformity_percent_options and log_key_multiplicity_options. Then, based on the values of misses_percent_options, probe_zipf_coefficient_options, and key set, a user-defined number of query sets is generated.

Range Query Input Generation Method

The range query input method first generates a user-defined number of key sets based on the input values of log_key_range_multiplier and log_expected_hits.

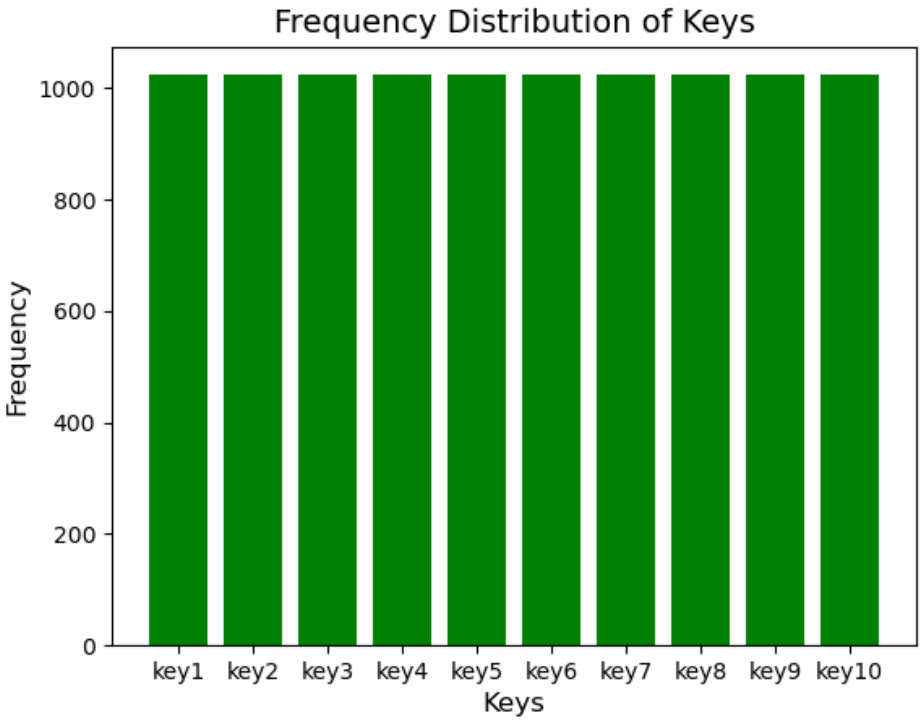

Figure 5.4: key_multiplicity=2^10 Distribution

Then, the generated key sets are used to create a user-defined number of query sets, influenced by the specified range parameters.

5.3 Default Value Selection for Parameters

5.3.1 General Parameters

The default values for general parameters are shown in Table 5.1.

Table 5.1: Configuration parameters and their default values.

Parameter	Default Value	Description
build_size	2^{23}	Size of the key set, chosen based on the memory size of the testing machine.
probe_size	2^{25}	Size of the probe set, chosen based on run time on the testing machine.
key_type	32-bit unsigned integer	32-bit keys are more commonly used in practical applications.
reverse_mapping_option	0	Default value is 0, set to 1 in some experiments.

5.3.2 Point Query Parameters

The default values for point query parameters are shown in Table 5.2.

Table 5.2: Configuration options and their default values.

Option	Default Value
misses_percent_options	0
build_key_uniformity_percent_options	100
probe_zipf_coefficient_options	0
log_key_multiplicity_options	0

5.3.3 Range Query Parameters

The default values for range query parameters are shown in Table 5.3.

5.4 Result Validation

In the phase of input generation, an expected value array is also generated based on the key set and query set. The logic for the expected value is that for each

Table 5.3: Configuration options and their default values.

Option	Default Value
log_key_range_multiplier	0
log_expected_hits	10

query, its expected value is equal to the sum of the keys' index in the key array for every qualified key. After the termination of querying, the algorithm output array will be compared with the expected value array to ensure the correctness of the algorithm.

5.5 Experiment Design for Revised RTX Range Query Algorithm

Sections 5.5 to 5.14 include the design and results of the first set of ablation experiments. In this set of experiments, we only compare against the original RTX and do not include other baselines, as the purpose of this set is solely to investigate the effect of each optimization method on RTX. There is no such set of experiments in cgRX.

In this experiment, our goal is to test how much the performance of RTX's range query has improved compared to the original RTX by revising the range query algorithm, as well as to evaluate the impact of point queries with duplicate keys.

We conduct **two** sets of experiments. The first set focuses on range query performance, where we vary the value of **log_expected_hits** across 2, 4, 6, 8, and 10. The second set examines point queries with duplicate keys, varying **log_key_multiplicity_options** across the same values: 2, 4, 6, 8, and 10. We choose to vary these parameters because they directly determine the number of intersections between rays and triangles, which has the most significant impact on performance.

5.6 Experiments for Revised RTX Range Query Algorithm

In this section, I will present the results of the experiment in section 5.5 which I conducted by varying the parameters declared in that section, and will derive insights from it.

5.6.1 Results for Range Query

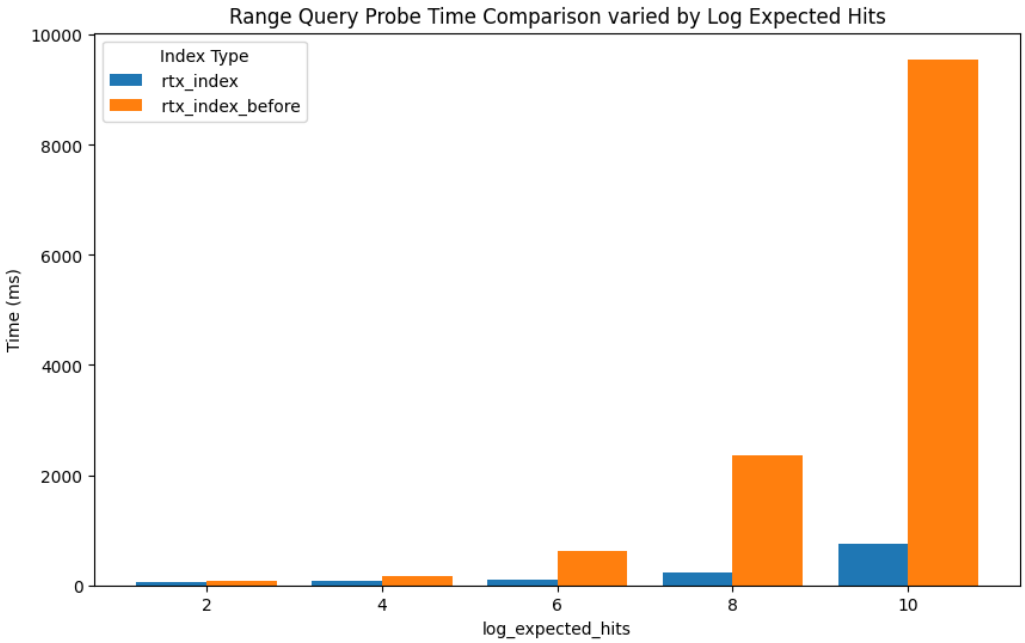

Figure 5.5: Range Query Probe Time Comparison Varied by Log Expected Hits

Understanding the effects of the new range query algorithm is the most important task. In Figure 5.5, the modified RTX achieves a significant improvement in range query performance, consistently outperforming the original RTX across all log_expected_hit settings. Its performance scales with the **increase** in expected hits, and when log_expected_hit is set to 10, the modified RTX is approximately **10**× faster than the original RTX. What could be the possible explanation for this?

In OptiX, triggering shader functions like Intersection (IS), Any-hit (AS), and Closest-hit (CS) are expensive. **Avoiding unnecessary intersections** can significantly improve query performance. When log_expected_hit = 2, each range query in the original RTX intersects with 4 triangles and triggers the shader 4 times. When log_expected_hit = 10, each range query intersects with 1024 triangles and triggers the shader 1024 times. In contrast, the modified RTX intersects with only one triangle per range query and triggers the shader once, with the remaining intersections handled by a linear scan. This explains why the performance of the modified RTX improves as log_expected_hit increases.

5.6.2 Results for point Query with Duplicated Keys

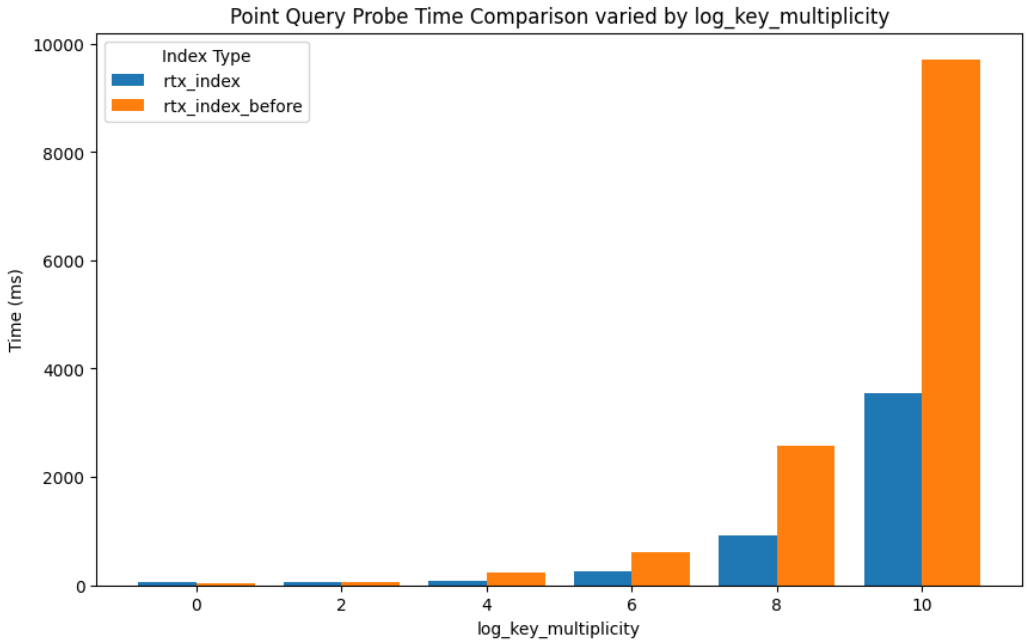

Figure 5.6: Point Query Probe Time Comparison Varied by Log Num Multiplicity

The result in Figure 5.6 shows a similar pattern to the result in Section 5.6.1 which is expected. However, the performance of the revised RTX is not as strong compared to Section 5.6.1. Notably, when log_num_multiplicity = 0, the revised RTX is even slower than the original RTX. This is due to the additional overhead introduced by the **linear scan kernel** in the revised RTX. When log_num_multiplicity = 10, the revised RTX is only about 2.5× faster.

For this performance issue, we observed that it is almost nonexistent when num_key_multiplicity = 0, suggesting that the slowdown might be caused by placing too many triangles at the same location. The method in Section 5.10 successfully addresses this issue, and detailed results and explanations will be provided.

5.7 Experiment Design for BVH Partitioning

The main objective of this experiment is to understand the impact of partitioning into smaller BVHs on the probe time of RTX. Additionally, since BVH partitioning also affects the BVH construction method to some extent, we also need to

investigate its impact on **build size** and **build time**. Another goal is to determine the **optimal number** of sub-BVHs based on the experimental results.

We will conduct a series of experiments by testing point queries on RTX. The number of sub-BVHs will be varied as 2^0, 2^2, 2^4, 2^6, and 2^8, while all other parameters will remain at their default values. The biggest number for sub-BVHs we chose here is 2^8 because building each sub-BVH produces additional overhead, and we believe this number is already sufficiently large.

5.8 Experiments for BVH Partioning

In this section, I will present the results of the experiment in Section 5.7, which I conducted by varying the parameters declared in that section, and I will derive insights from it.

5.8.1 Results for Partitioning Impact on Probe Time

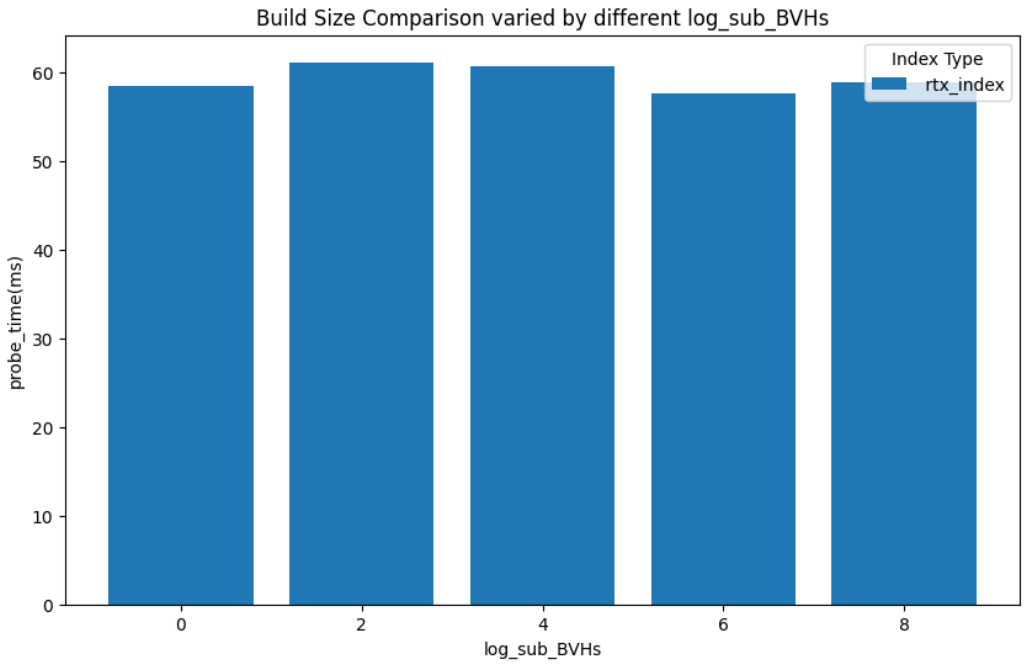

Figure 5.7: Build Time Comparison Varied by Log Sub BVHs

We were expecting the probe time to be improved due to the fact that the BVH depth is reduced by partitioning the BVH into smaller sub-BVHs. However, the result in Figure 5.7 contradicts this prediction. The probe_time only exhibits

some normal fluctuations and does not show any correlation with the number of sub-BVHs. This result indicates that the **depth** of the BVH has not changed significantly. A possible explanation is that the data is already well distributed, so even when divided into multiple sub-BVHs, the overall BVH depth remains largely unchanged.

5.8.2 Results for Partioning Impact on Build Size

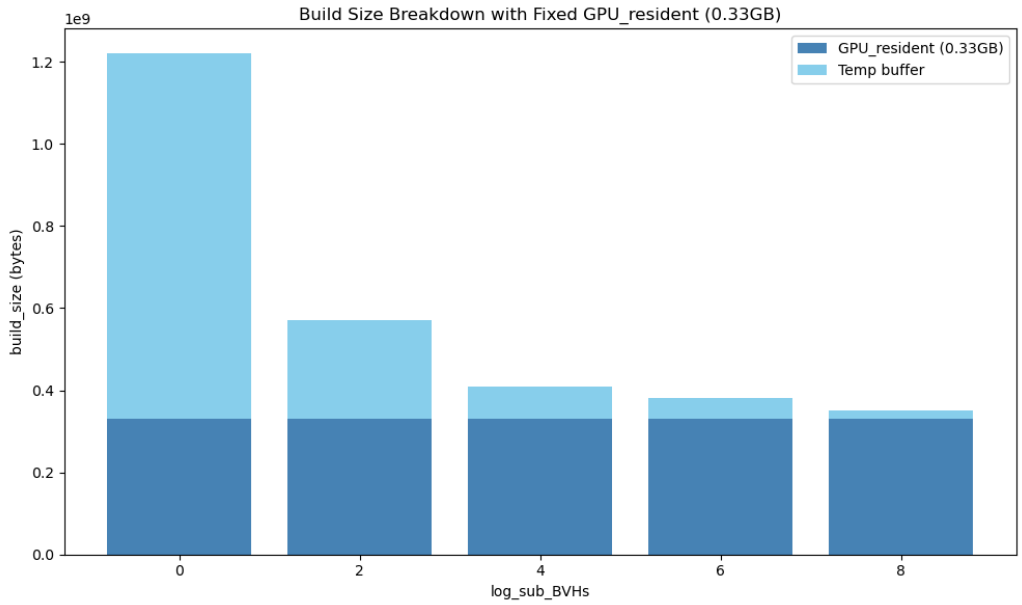

Figure 5.8: Build Size Comparison Varied by Log Sub BVHs

The results for build size were somewhat unexpected. The result in Figure 5.8 shows that as the number of sub-BVHs increases, the build_size decreases. When the partition size is 256, the build size is only one-third of that of the non-partitioned BVH. What could be the possible explanation for this?

In the cgRX paper, two different terms are used to represent build size: build_size and GPU_resident_byte. build_size refers to the amount of GPU memory required to construct the BVH, while GPU_resident_byte represents the final size of the BVH stored on the GPU. Under the same conditions, build_size can be approximately twice GPU_resident_byte, or even more. This is because, after constructing the BVH, OptiX provides a BVH compression function called **optixAccelCompact**, which reduces the BVH size. However, during the compaction process, both the uncompressed BVH data and an **empty memory buffer** for

the compressed BVH need to be stored simultaneously, the build size is the sum of these two buffers.

How does BVH partitioning reduce build size? When BVH partitioning is applied, each small BVH's data size is only 1/num_sub_BVH of the original BVH. As a result, during compression, the required temporary buffer size for each operation is also reduced to 1/num_sub_BVH of the original. This significantly reduces the GPU memory required during the construction process, leading to a substantial decrease in build size compared to the case without partitioning. BVH itself **consumes** a certain amount of memory, so as the partition count increases from 2^4 to 2^6 or even 2^8, the reduction in build_size is not as significant as before.

5.8.3 Results for Partitioning Impact on Build Time

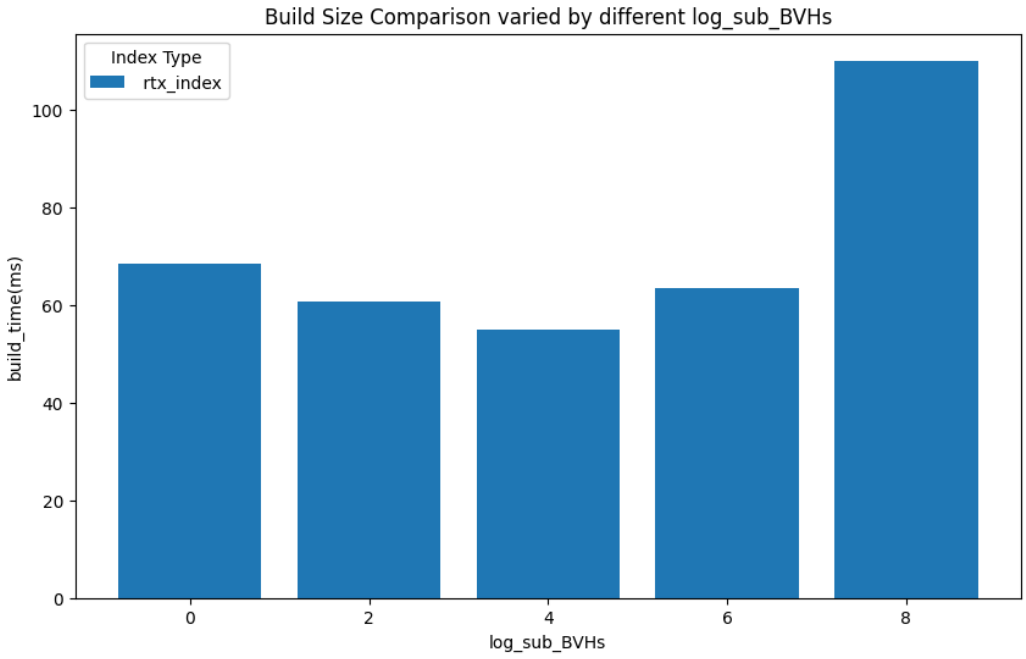

Figure 5.9: Probe Time Comparison Varied by Log Sub BVHs

The result for build time in Figure 5.9 indicates that when the number of sub-BVHs is below 2^4, there is a slight improvement in build time. However, when the number exceeds 2^4, the additional overhead of constructing multiple BVHs outweighs the benefits, leading to an increase in build time instead of a decrease.

5.8.4 Choice of the Optimal number for Sub-BVHs

Based on the result from the last three subsections, we choose 2^4 as the optimal number for sub-BVHs as it provides relatively good build size improvement and the best build time compared to other numbers.

5.9 Experiment Design for Compression on Duplicate Elements

In this experiment, our goal is to test whether the problem in Section 5.6.2 is solved by compressing the duplicated keys into buckets. We will compare the probe time of compressed RTX for point queries where log_key_multiplicity $>$0 and the result in Figure 5.6. Since compression itself introduces additional overhead, we will also analyze the **ratio** of compression time to build time across different log_key_multiplicity values.

We will set up an RTX point query experiment, varying log_key_multiplicity as 0, 2, 4, 6, 8, and 10, while keeping all other parameters at their default values.

5.10 Experiments for Compression on Duplicate Elements

In this section, I will present the results of the experiment in Section 5.9 which I conducted by varying the parameters declared in that section, and will derive insights from it.

5.10.1 Results for probe time

The result in Figure 5.10 demonstrates that the performance loss issue discussed in Section 5.6.2 has been resolved. Compared to the results in Figure 5.6, the probe time has significantly improved for different num_key_multiplicity values, reaching a performance level nearly identical to the range query results in Figure 5.5, where no performance loss was observed. What could be an explanation for it?

We initially thought that placing too many triangles at the same location might mess up the BVH structure. However, by comparing the results of range queries in Figure 5.5 and point queries with duplicated keys in Figure 5.6 in the original RTX, we found that the original RTX did not experience the same **performance degradation**. Therefore, we suspect that this issue is related to our algorithm implementation. One possible explanation is that we use the

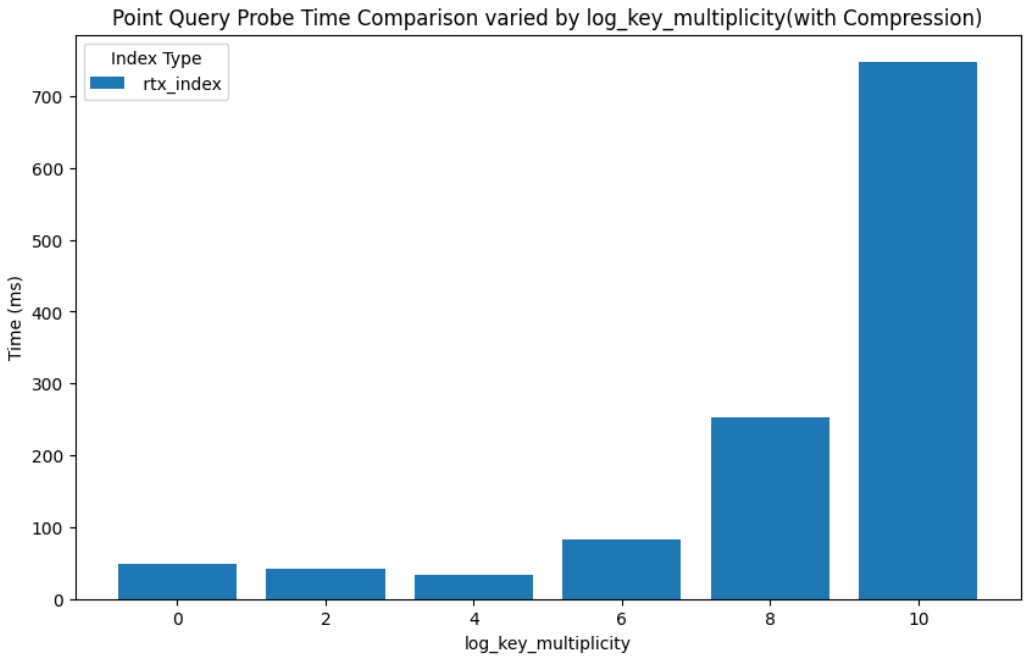

Figure 5.10: Probe Time Comparison Varied by Log Key Multiplicity

closest hit shader for our new range query algorithm, the closest hit shader needs to compute the distances to all intersecting triangles and determine the shortest one. When all triangle distances are the same, it returns the triangle with the smallest index in the BVH triangle buffer. This may introduce **extra computations**. However, the original RTX uses any hit shader, which does not require distance calculations, so its performance remains unaffected.

5.10.2 Results for compression time

The **tradeoff** of the improvement in the last section is the computation time for compression. The result in Figure 5.11 shows that the compression time is nearly 3 ms in the worst case. Comparing this to the build time of nearly 50 ms in Figure 5.9, the compression time accounts for approximately 1/15 of the build time. Overall, the time required for compression has a very tiny impact on overall performance.

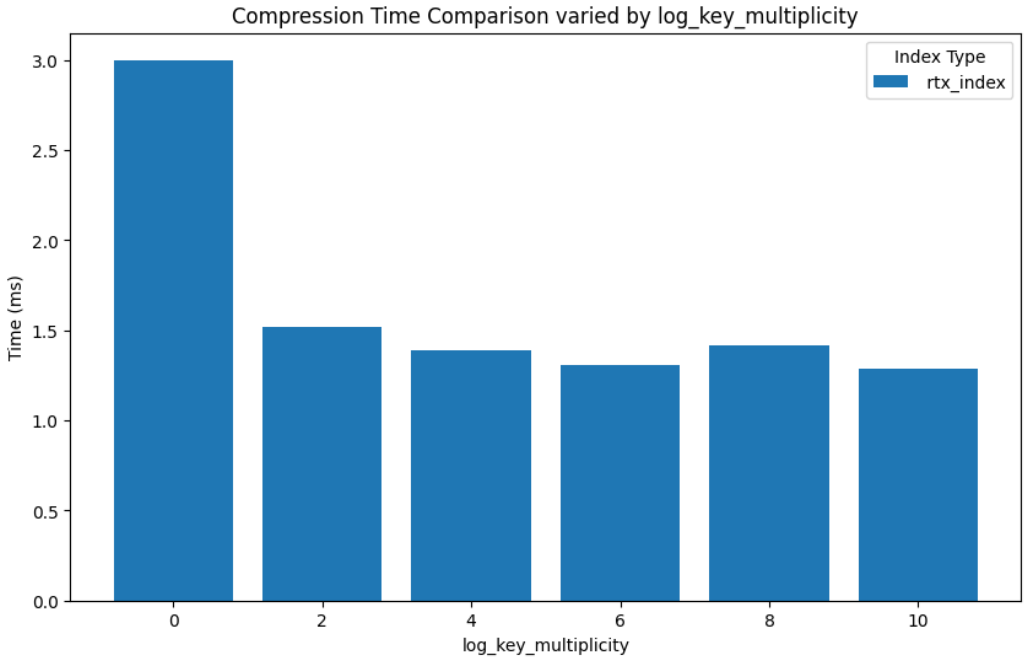

Figure 5.11: Compression Time Comparison Varied by Log Key Multiplicity

5.11 Experiment Design for Sorting the Query Set

In this experiment, our goal is to understand the impact of sorting the query set on performance. We will compare two revised RTXs under the same conditions, the only difference is that one with a **sorted** probe set and the other being **unsorted**. We will evaluate their performance on both range queries and point queries. We will also look at the sorting time as it is a trade-off.

We will conduct two sets of experiments. The first experiment focuses on point queries, while the second focuses on range queries. In both experiments, we will vary the log_probe_size with values of 21, 22, 23, 24, and 25, while keeping all other parameters at their default values. When comparing the results, we will combine the probe_time and sort_time to evaluate the overall performance.

5.12 Experiments for Sorting the Query Set

In this section, I will present the results of the experiment in Section 5.11 which I conducted by varying the parameters declared in that section, and will derive insights from it.

5.12.1 Results for Point Query

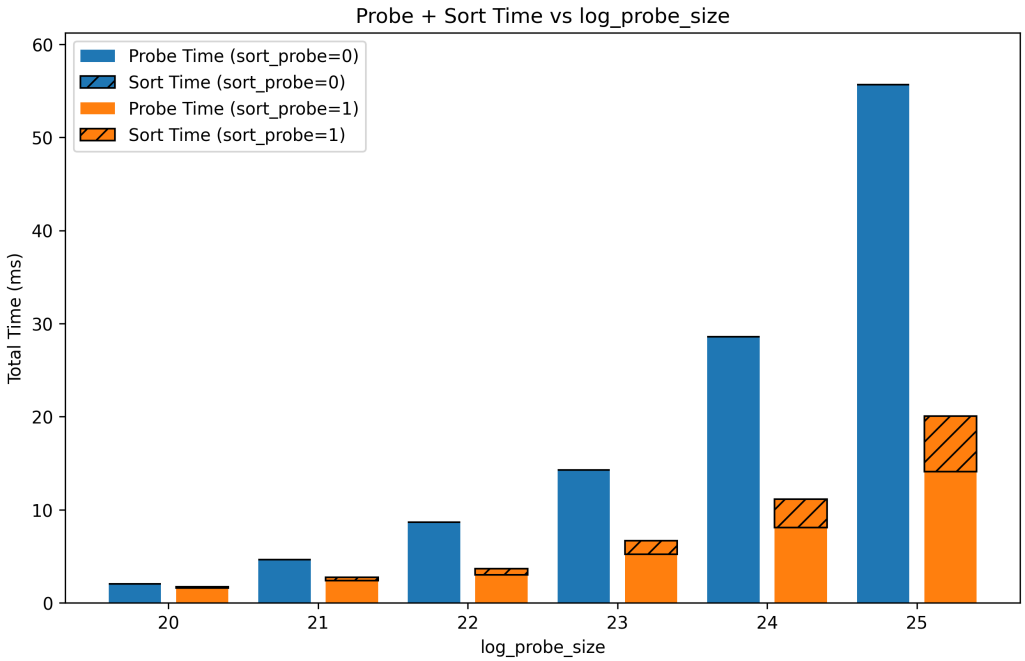

Figure 5.12: Total Probe Time Comparison Varied for Point Query by log probe size

The result for the point query meets the expectation. As shown in Figure 5.12, the RTX with probe_set sorted consistently achieves better performance than the one without sorting across all log_probe_size values. Moreover, the performance gain increases as probe_size grows. When accounting for the sorting overhead, the speedup reaches up to 3× at its peak.

5.12.2 Results for Range Query

The result for the range query shows greater improvement compared to the point query. As shown in Figure 5.13, the range query consistently maintains around a 4× speedup, regardless of the log_probe_size, even when accounting for the sorting overhead. The proportion of sorting time is smaller in range queries because the time required to sort the probe set is fixed. Since range queries take longer, the sorting time accounts for a smaller fraction of the total execution time.

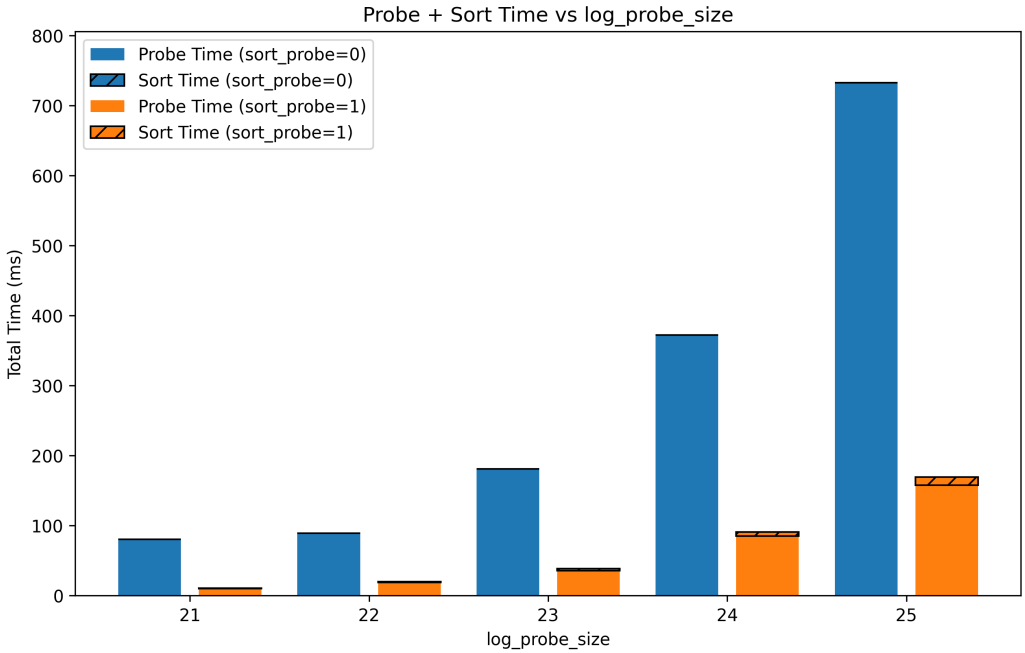

Figure 5.13: Total Probe Time + Sort Time Comparison Varied for Range Query by log probe size

5.12.3 Summary for Sorting Probe Set Experiment

Overall, sorting the probe set can effectively improve query efficiency in most cases for both range queries and point queries, even with the additional sorting overhead.

5.13 Experiment Design for Inverse Mapping

In this experiment, our goal is to determine whether we can achieve the same performance as that obtained with the inverse-mapped log key size and log probe size. There is no need to test the build size here, as a smaller map size generally ensures a smaller BVH size. We will only do the test on point query.

We will conduct 1 point query experiment. We set **3 runs** in this experiment. For the first run, we set the log_build_size to 21 and the log_probe_set to 23. For the second run, we set the log_build_size to 23 and the log_probe_set to 21. For the third run, we set the log_build_size to 23, the log_probe_set to 21 and inverse_mapping_option to 1. Under this setting, we can observe whether the expected effect of inverse mapping is achieved by comparing run1 and run3. Additionally, by comparing run2 and run3, we can evaluate the performance

improvement brought by inverse mapping compared to the non-inverse mapping approach.

5.14 Experiment for Inverse Mapping

In this section, I will present the results of the experiment in Section 5.13 which I conducted by varying the parameters declared in that section, and will derive insights from it.

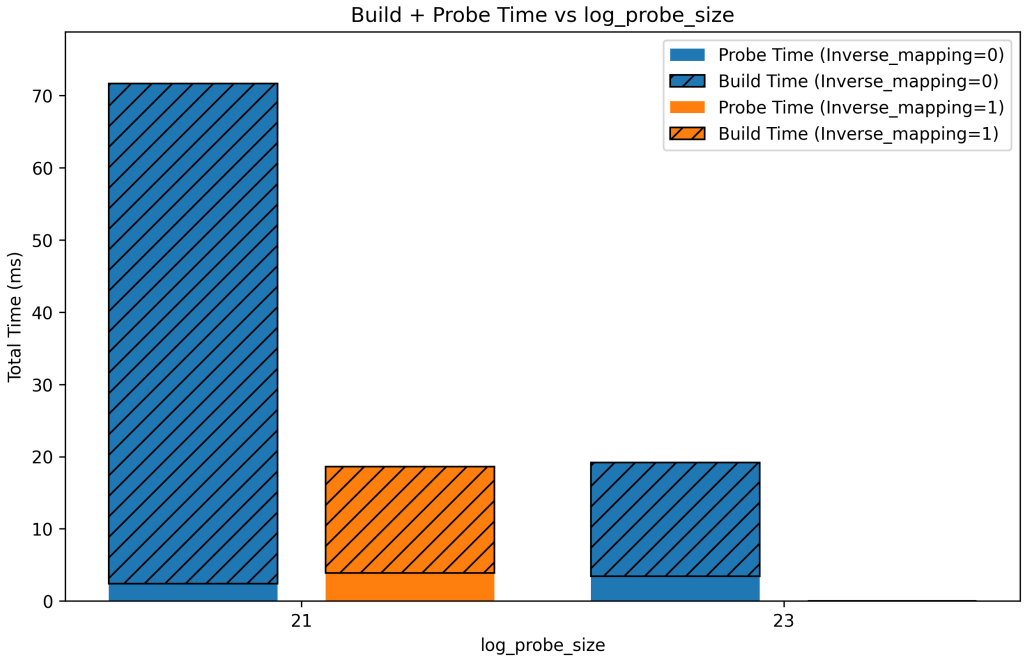

Figure 5.14: Total Probe Time + Build Time Comparison for Point Query Varied by log probe size

In Figure 5.14, the comparison between run1 (the blue cylinder on the right) and run3 (the orange cylinder) shows nearly identical execution times, indicating that inverse mapping can achieve the same probe time and build time. The comparison between run3 (the orange cylinder) and run2 (the blue cylinder on the left) demonstrates that reverse mapping achieves a 3.5× performance improvement compared to the case without inverse mapping.

These results indicate that when log_build_size >log_probe_size, inverse mapping has the potential for improving performance.

5.15 Experiments Design for Optimized RTX

This experiment provides a comprehensive comparison of the **optimized RTX** against cgRX and other baselines used in cgRX. The optimized RTX applies the following optimization techniques:

- Applying the range query revision that was tested in Section 5.6
- Applying the BVH partitioning algorithm that was tested in Section 5.8, the choice for the number of sub-BVHs is 2^4
- Applying the Compression technique tested in Section 5.10
- Sorting the query set that was tested in Section 5.12
- The inverse mapping technique will not be applied here to provide a more direct performance comparison with other baselines.

The baselines we are going to compare are shown in Table 5.4:

Table 5.4: Description of methods used in the experiment.

Method	**Description**
cgRX5x4	5×4 refers to a bucket size of 2^5.
cgRX8x4	8×4 refers to a bucket size of 2^8.
Sorted Array (SA)	Referred to as SA in the figure.
Hash Table	Does **not** support range queries.
B+ Tree	Referred to as B-link-tree 16 in the figure. Does **not** support point queries when `log_key_multiplicity` > 0.

The **optimized** settings for baselines as these settings provide the best query performance for these baselines shown in Table 5.5:

Table 5.5: Sorting settings for different methods.

Method	**Sorting the Key Set**	**Sorting the Query Set**
cgRX5x4	Yes (1)	Yes (1)
cgRX8x4	Yes (1)	Yes (1)
Sorted Array	Yes (1)	Yes (1)
Hash Table	No (0)	Yes (1)
B+ Tree	No (0)	Yes (1)

Under these settings, we add the sorting time to the build time of cgRX5x4, cgRX8x4, and Sorted Array. However, we do not add the sorting time for the query set, as all methods are configured to sort it by default.

Figure 5.15 and Figure 5.16 illustrate these settings. Figure 5.15 shows that all methods benefit from performance improvements after sorting the query set. Even when factoring in the sorting time, most methods achieve significant gains, except for B-tree, which experiences a slight increase in overall time.

Meanwhile, Figure 5.16 indicates that sorting the key set provides no improvement for B-tree and hash table, so their sorting key set option is set to zero. By default, sorting the key set is set to 1 for RTX, cgRX, and Sorted Array.

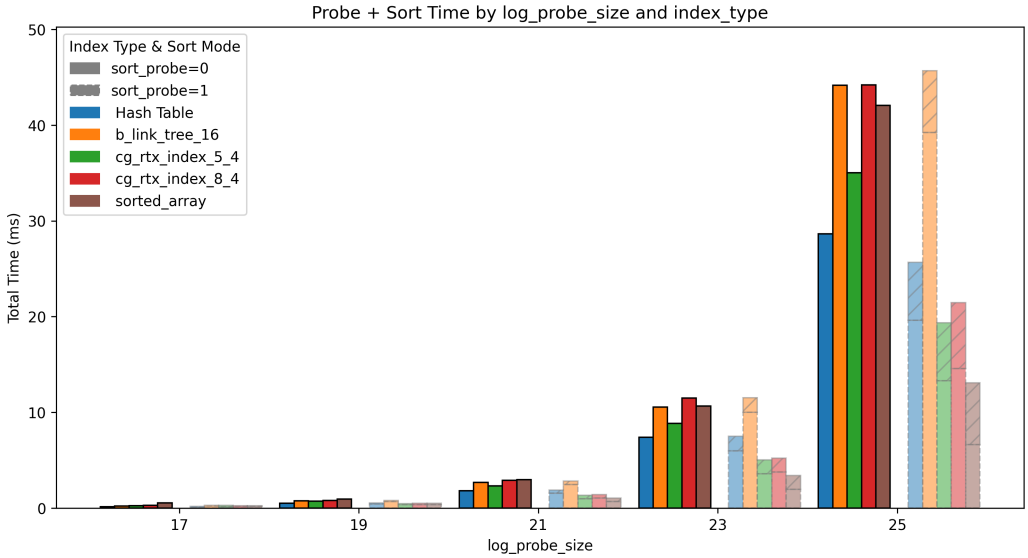

Figure 5.15: Probe time + sort time for query set Comparison Varied by log build size

In this experiment, our goal is to reproduce the cgRX experiment while replacing the original RTX with the optimized RTX to ensure a fairer comparison. Therefore, our experimental setup will remain largely consistent with that in cgRX.

Overall, our experiments are divided into **three** main modules: the build experiment, the point query experiment, and the range query experiment. The specific experiments included in each module will be listed below.

- build module: For all the experiments in this module, we will vary the log_build_size by 15, 17, 19, 21, and 23 while all the other parameters are set to default. The experiments in this module include:
  - build size experiment
  - build time experiment

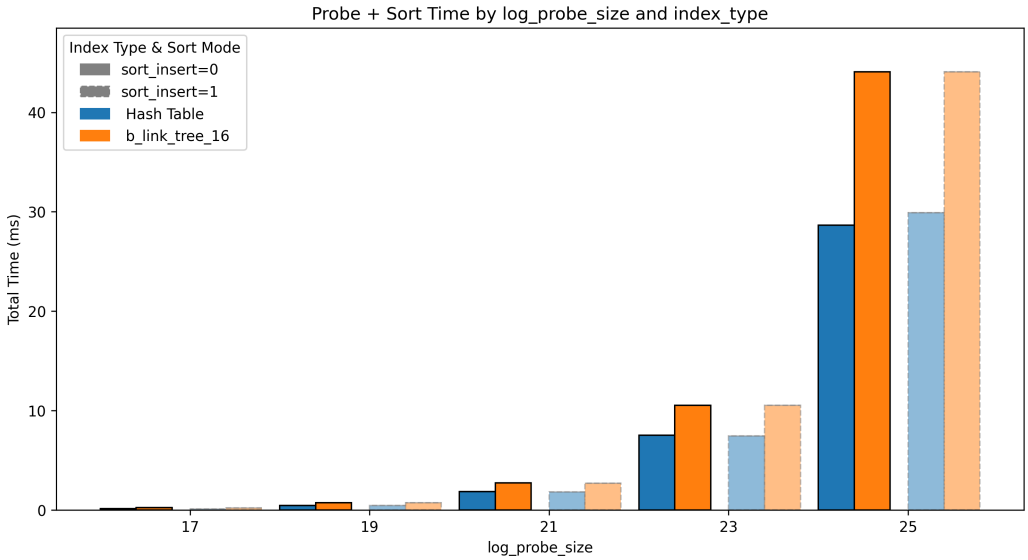

Figure 5.16: Probe Time Comparison Varied by sort_insert and log build size

- point query module: This experiment includes multiple sub-experiments, each focusing on different data distributions. The experiments in this module include:
  - basic point query experiment, for this experiment, we vary the log_probe_size by 17, 19, 21, 23, and 25 while all the other parameters are set to default.
  - point query with duplicated key experiment, we vary the log_key_multiplicity by 2, 4, 6, 8, and 10 while all the other parameters are set to default.
  - point query varied by key uniformity. We vary the build_key_uniformity_percent_options by 0%, 1%, 5%, 20%, 40%, 60%, 80%, and 100% while all the other parameters are set to default.
  - point query with different hit ratios. We vary the misses_percent_options by 0%/0%, 1%/0%, 10%/0%, 30%/0%, 50%/0%, 70%/0%, 90%/0%, 99%/0%, 100%/0%, 50%/50%, and 0%/100%.
  - point query with different look up skews. We vary the Zipf coefficient by 0.0, 0.25, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0, and 5.0.
- Range query module: This experiment includes multiple sub-experiments, each focusing on different data distributions or ranges expected to hit. The experiments in this module include:

– basic range query experiment. For this experiment, we vary the log_probe_size by 17, 19, 21, 23, and 25 while all the other parameters are set to default.

– range query with the different expected hit. We vary the log_expected_hit by 0, 2, 4, 6, 8, and 10 while all the other parameters are set to default.

5.16 Experiments for Optimized RTX

In this section, I will present the results of 3 experiment modules in Section 5.15 and compare them to the result in cgRX, finally deriving insights from it.

5.16.1 Results for Build Module

In this subsection, I will present the results for the build module.

Result for Build Size

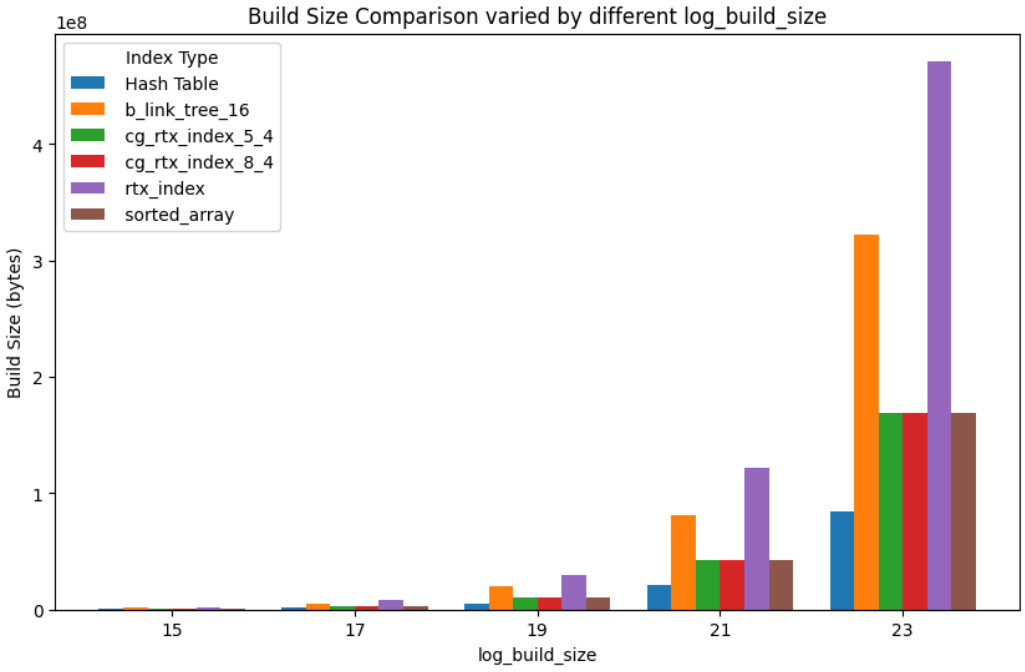

Figure 5.17: Build Size Comparison Varied by log build size

Due to the applied BVH partition, the result in Figure 5.17 shows that the memory footprint of cgRX, which was originally about 4.5x that of optimized RTX in the cgRX paper, has been compressed to approximately 2.5x the size for any different log_build_size. Compared to other baselines, the largest gap is around 2.5× larger than the sorted array, while the smallest, the B+ tree, is only about 1.25× larger.

This comparison shows that BVH partitioning can mitigate RTX's weakness in memory footprint, but RTX still faces **challenges** in this aspect.

Result for Build Time

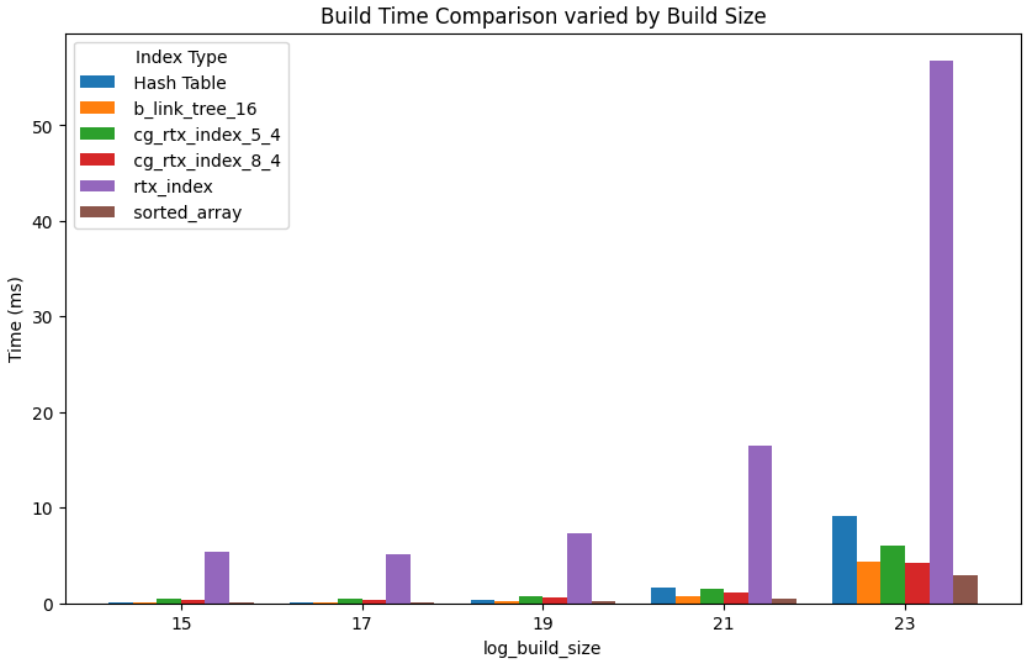

Figure 5.18: Build Time Comparison Varied by log build size

The result in Figure 5.18 shows that optimized RTX's build time significantly exceeds that of other baselines for any different log_build_size. The optimized RTX build time is at best, just over 5× that of the hash table. This factor increases to 8–10× when compared to the two cgRX variants and the B-tree, and at worst, it reaches up to 15× when compared to the sorted array. However, the impact of build time depends on how many times the index is used—if the index is queried multiple times, the build time overhead is effectively amortized across those queries.

This result indicates that the optimized RTX has not achieved significant improvements in build time compared to before, which might be the **biggest challenge** at present. This is also reflected in Figure 5.8 and Figure 5.9, where BVH partitioning effectively reduces the build size of RTX but does not lead to substantial improvements in build time.

5.16.2 Results for Point Query Module

Result for Basic Point Query Experiment

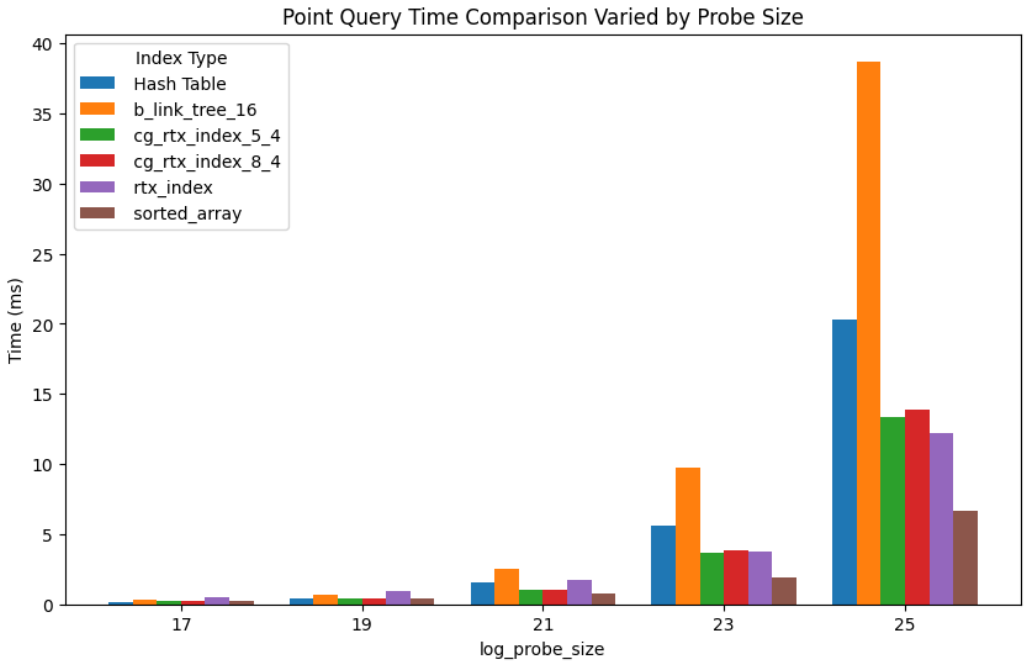

Figure 5.19: Probe Time Comparison for Point Query Varied by log probe size

The result in Figure 5.19 is that the query time of optimized RTX is generally slower than most other baselines in most cases when log_probe_size is less than 23. This difference is less than 2 ms. Optimized RTX surpasses all baselines except for the sorted array and matches cgRX5_4 when log_probe_size equals 23 and outperforms all baselines except for the sorted array when log_probe_size is greater than 23. The sorted array consistently maintains the fastest performance in all cases, leading by at least 2× compared to other methods.

What could be the explanation for the fact that optimized RTX can outperform cgRX when the probe size is large enough? Both optimized RTX and cgRX

can benefit from the acceleration brought by sorting the query set, but RTX achieves a greater acceleration ratio. The reason is that RTX's probe time only includes **tracing time**, whereas cgRX's probe time consists of both **tracing** and **partition search**. According to the results in cgRX, in cgRX 5x4, the tracing time and partition search time are the same, while in cgRX 5x8, the partition search time is 1.3× the tracing time. As the probe size increases, the acceleration effect for RTX becomes even more significant.

Overall, the optimized RTX performs poorly when the number of queries is small. However, when the query size exceeds 2^{23}, it surpasses all other methods except for the sorted array. The performance ranking of these methods is **completely different** from the results of cgRX. A detailed discussion will be provided in the result for the Point Query with Different Key Uniformity section.

Result for Point Query with Duplicated Key Experiment

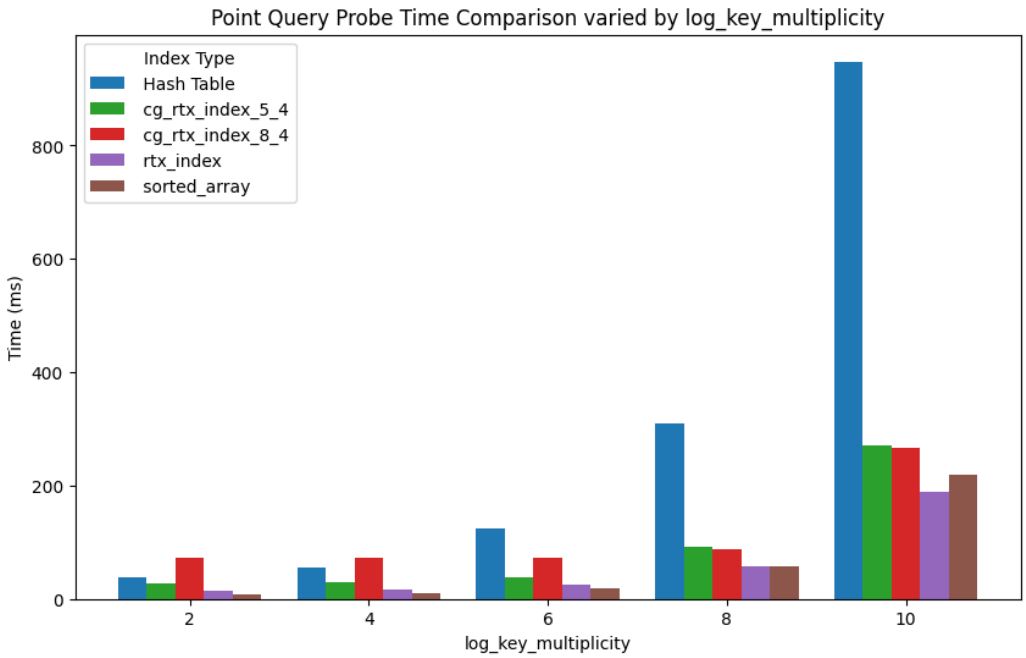

Figure 5.20: Probe Time Comparison for Point Query Varied by log key multiplicity

The optimized RTX performs well when duplicate keys appear in the key set. As shown in Figure 5.20, optimized RTX outperforms all baselines except for the sorted array when num_key_multiplicity is any value from 2 to 10. However, the overall trend shows that as the number of duplicate keys increases, the

performance lead of optimized RTX decreases. Sorted array is even more affected by this trend, with its probe time matching optimized RTX when the number of duplicate keys reaches 2^8, and being surpassed by optimized RTX at 2^{10}.

Result for Point Query with Different Key Uniformity

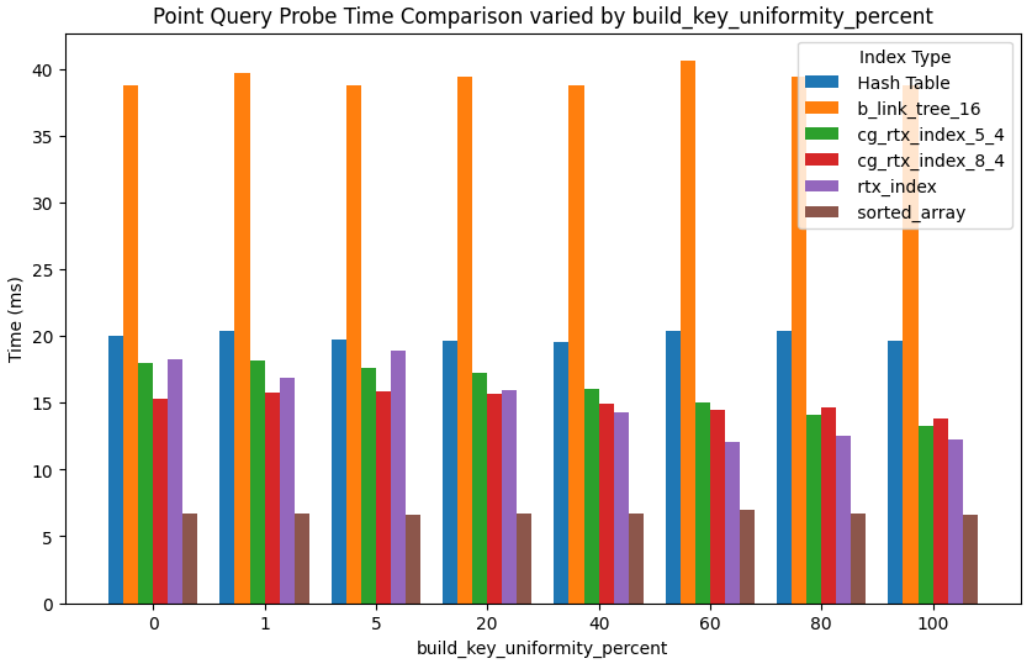

Figure 5.21: Probe Time Comparison for Point Query Varied by Key Uniformity

The results in Figure 5.21 differ from those of cgRX. Due to the **impact of sort_probe**, cgRX, and optimized RTX are now the second-fast methods. The sorted array benefits from the sort_probe the most, and has become the best-performing method, maintaining at least twice the performance advantage over the others. However, Hash table and B+ tree do not benefit from sort_probe. The hash table, which originally had at least twice the performance advantage over other methods, has now been outperformed by all methods except for the B+ tree. The B+ tree, which previously had a similar performance to RTX, is now at least twice as slow as the other methods in this experiment.

Result for Point Query with Different Miss Percentages

The results in Figure 5.22 for different Miss percentages experiment shows a similar result to that in the last experiment. The only difference is that optimized

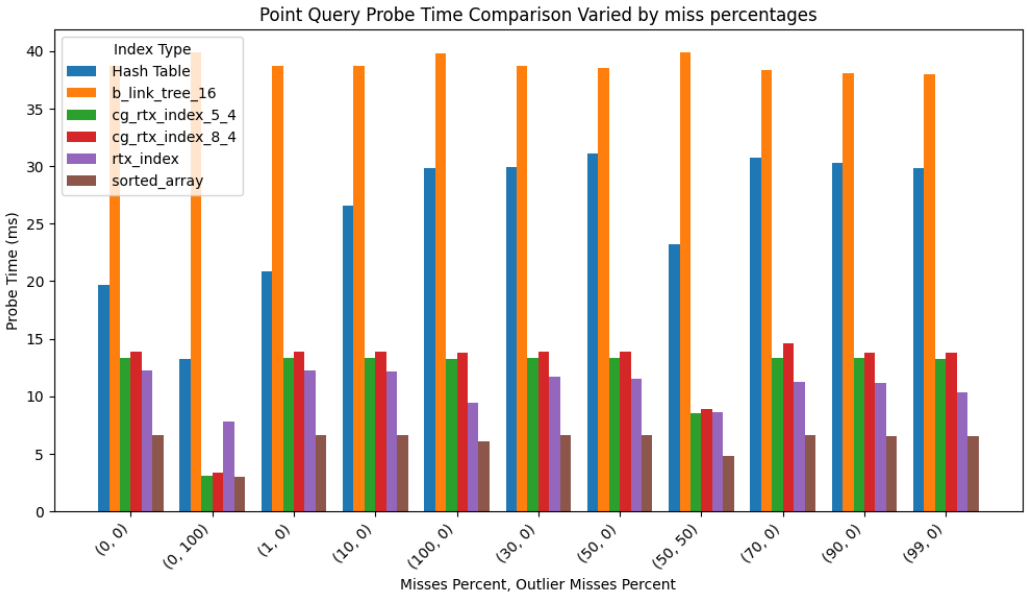

Figure 5.22: Probe Time Comparison for Point Query Varied by miss percentages

RTX is slightly faster in most cases. The explanation for this is that cgRX needs extra time to search in the bucket to determine whether is a **miss** while RTX does not. The only exception occurs when the outlier miss percentage is 0, where optimized RTX is twice as slow as cgRX. In this case, the search is trivial, so the cgRX can finish very quickly.

Result for Point Query with Different Look-Up Skews

In the results of this experiment shown in Figure 5.23, the main difference from the last one is that when the Probe_zipf_Coefficient is greater than 1, the probe time of RTX is almost twice that of cgRX. This result indicates that cgRX benefits more from sorting the probe set in this case. The explanation for this is that when the probe set is more skewed, it means there are more repeated queries. After sorting the queries, these identical queries cause more threads to search for the same key during the "search in bucket" step. As a result, they finish at the same time, improving overall performance.

5.16.3 Results for Range Query Module

In this section, I will present the result for the Range Query Module.

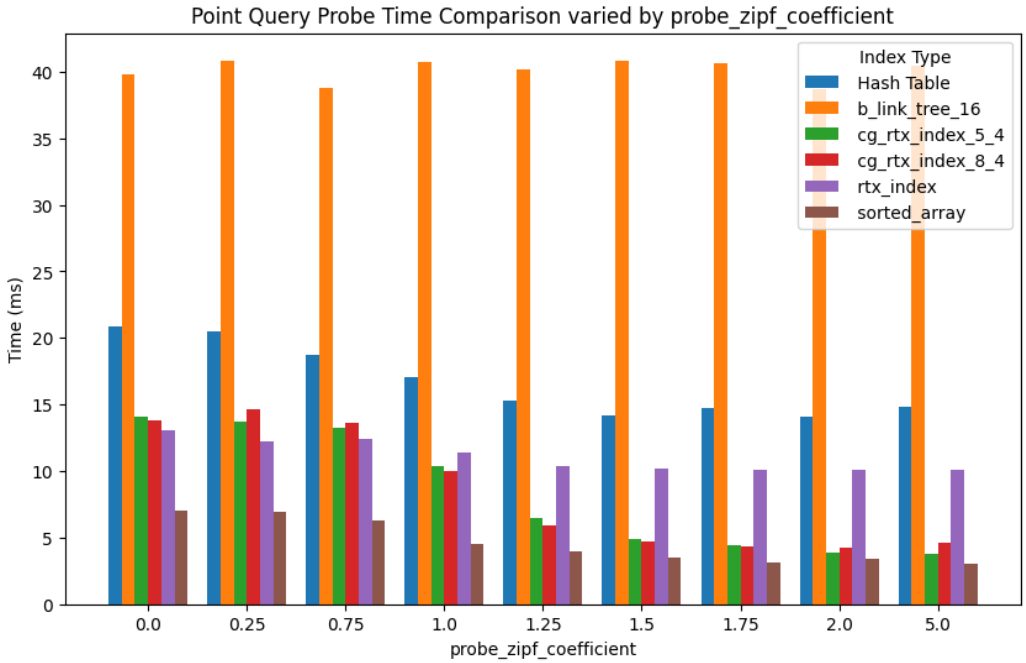

Figure 5.23: Probe Time Comparison for Point Query Varied by zipf coefficient

Result for Basic Range Query Experiment

The result for the Basic Range Query experiment shown in Figure 5.24 is completely different from that of cgRX. In the results of cgRX, RTX is the slowest among all methods for all log_probe_size values, with its worst case being up to four times slower than cgRX. The sorted array is the second slowest, reaching up to twice the time of cgRX at its worst. B+ tree is the fastest, with a runtime only half that of cgRX. The results in this experiment show that optimized RTX exhibits excellent performance, with probe time approximately half that of cgRX and one-third that of B+ tree. In only 1 case, it even slightly outperforms the sorted array. The sorted array achieves the best performance in almost all cases but only slightly outperforms optimized RTX.

Result for Range Query Experiment with Different Expected Hit

The results for this experiment shown in Figure 5.25 are very similar to the last experiment. The two differences are that when the **hit_range** is small: first, the performance advantage of optimized RTX and Sorted Array becomes more pronounced; second, the performance improvement of the sorted array is even greater, reaching up to twice that of optimized RTX.

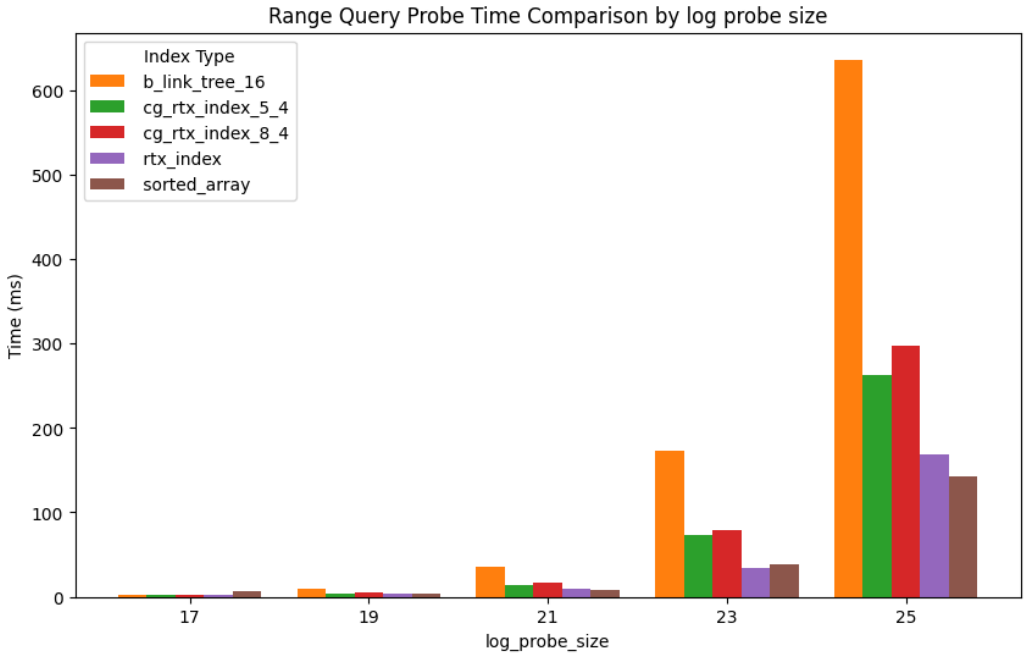

Figure 5.24: Probe Time Comparison for Basic Range Query Experiment Varied by log probe size

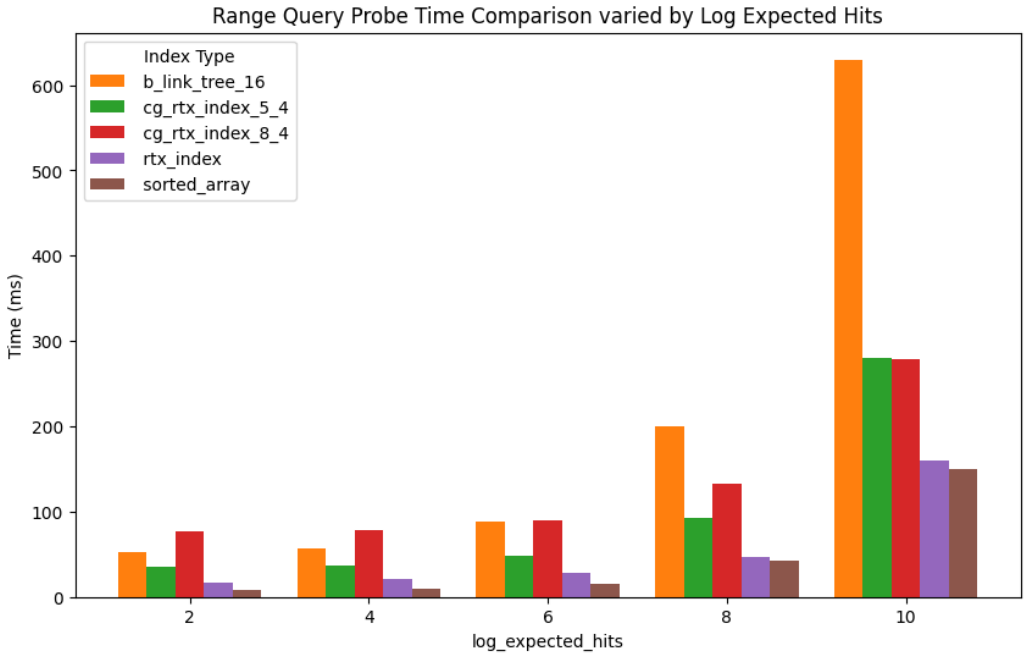

Figure 5.25: Probe Time Comparison for Range Query Experiment Varied by log expected hit

5.17 Chapter Summary

In this chapter, we first conducted an ablation study by individually testing each optimization technique. We specifically analyzed the impact of each optimization technique on RTX's performance and investigated the underlying reasons. Overall, these methods effectively improve RTX's build size, point query, and range query performance.

Next, we selectively applied these methods to RTX to replicate the experimental groups in the cgRX paper. Additionally, we optimized the settings for other baselines to achieve their best performance. The final results differed significantly from those reported in the cgRX paper.

Chapter 6

Conclusion and Future Work

The goal of this chapter is to provide a summary of the steps taken in this project, the experiments conducted, and the results obtained. Additionally, potential future directions for expanding the scope of this project will be discussed.

6.1 Conclusion

The development of GPU databases and GPU indexing is almost certain to increase in the future due to larger GPU memory capacity and massive parallel computing. The appearance of RTX is a very novel approach and an exploration of new, efficient GPU indexing methods. The cgRX improves the efficiency of construction, range queries, and updates, and makes ray-tracing index even more competitive compared to RTX. However, the experimental results of cgRX may be inaccurate because an unoptimized RTX and unoptimized settings were used for other baselines.

To explore a GPU database index with higher performance, we first revised RTX's range query algorithm and explored various OptiX optimization techniques, evaluating the impact of each optimization through experiments. Then, we applied the best configurations to cgRX and all other baselines and reproduced the experiments used in cgRX.

Our experimental results show that the sorted array is the overall best approach for GPU index. cgRX matches the sorted array in build performance but falls behind in range query and point query speed. In contrast, RTX achieves nearly the same range query performance as the sorted array but has a noticeable disadvantage in both build time and point query speed.

6.2 Future work

The work presented in this thesis provides new insights and opens multiple directions for future research.

A potential future work can be exploring applying BVH partitioning algorithm to cgRX since BVH paritioning has a chance to further reduce the GPU memory requirement during build. The implementation may be challenging due to the complexity of cgRX's construction algorithm.

Another potential future work can be exploring applying inverse mapping to range query. This algorithm itself may not be difficult to conceive, but optimizing the extensive global memory reads and atomic additions in the linear scan phase can be challenging.

Bibliography

[APO23] Awad, M. A., Porumbescu, S. D., AND Owens, J. D. A GPU Multiversion B-Tree. In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. PACT '22. Association for Computing Machinery, Chicago, Illinois, 2023, 481–493. ISBN: 9781450398688. DOI: 10.1145/3559009.3569681. URL: https://doi.org/10.1145/3559009.3569681.

[BM72] Bayer, R., AND McCreight, E. M. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1 (1972), 173–189. URL: https://doi.org/10.1007/BF00288683.

[Cor+09] Cormen, T. H., ET AL. *Introduction to Algorithms*. MIT Press, 2009. URL: https://mitpress.mit.edu/9780262046305/introduction-to-algorithms-fourth-edition/.

[Eva+21] Evangelou, I., ET AL. Fast Radius Search Exploiting Ray Tracing Frameworks. *Journal of Computer Graphics Techniques (JCGT)* 10, 1 (2021), 25–48. ISSN: 2331-7418. URL: http://jcgt.org/published/0010/01/02/.

[Gla89] Glassner, A. S. *An Introduction to Ray Tracing*. Academic Press, 1989. URL: https://archive.org/details/introductiontora0000unse_l1j8/page/n9/mode/2up.

[GLZ24] Geng, L., Lee, R., AND Zhang, X. RayJoin: Fast and Precise Spatial Join. In: *Proceedings of the 38th ACM International Conference on Supercomputing*. ICS '24. Association for Computing Machinery, Kyoto, Japan, 2024, 124–136. ISBN: 9798400706103. DOI: 10.1145/3650200.3656610. URL: https://doi.org/10.1145/3650200.3656610.

[Gra] Graefe, G. *Query Evaluation Techniques*. URL: https://people.eecs.berkeley.edu/~fox/summaries/database/query_eval_5-8.html.

[Gro24] Group, O. *MVGpuBTree: Multi-Version B-Tree on GPU*. Accessed: 2025-02-28. 2024. URL: https://github.com/owensgroup/MVGpuBTree.

[Gut84] Guttman, A. R-trees: a dynamic index structure for spatial searching. In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. SIGMOD '84. Association for Computing Machinery, Boston, Massachusetts, 1984, 47–57. ISBN: 0897911288. DOI: 10.1145/602259.602266. URL: https://doi.org/10.1145/602259.602266.

[He+09] He, B., ET AL. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.* 34, 4 (Dec. 2009). ISSN: 0362-5915. DOI: 10.1145/1620585.1620588. URL: https://doi.org/10.1145/1620585.1620588.

[Hen+24] Henneberg, J., ET AL. *More Bang For Your Buck(et): Fast and Space-efficient Hardware-accelerated Coarse-granular Indexing on GPUs*. 2024. arXiv: 2406.03965 [cs.DB]. URL: https://arxiv.org/abs/2406.03965.

[HS23] Henneberg, J., AND Schuhknecht, F. RTIndeX: Exploiting Hardware-Accelerated GPU Raytracing for Database Indexing. *Proc. VLDB Endow.* 16, 13 (Sept. 2023), 4268–4281. ISSN: 2150-8097. DOI: 10.14778/3625054.3625063. URL: https://doi.org/10.14778/3625054.3625063.

[Jun+20] Junger, D., ET AL. WarpCore: A Library for Fast Hash Tables on GPUs. In: *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE Computer Society, Los Alamitos, CA, USA, Dec. 2020, 11–20. DOI: 10.1109/HiPC50609.2020.00015. URL: https://doi.ieeecomputersociety.org/10.1109/HiPC50609.2020.00015.

[Kal+12] Kaldewey, T., ET AL. GPU join processing revisited. In: *Proceedings of the Eighth International Workshop on Data Management on New Hardware*. DaMoN '12. Association for Computing Machinery, Scottsdale, Arizona, 2012, 55–62. ISBN: 9781450314459. DOI: 10.1145/2236584.2236592. URL: https://doi.org/10.1145/2236584.2236592.

[NVI19] NVIDIA *GeForce GTX Graphics Cards Now Support DXR Ray Tracing*. Accessed: 2025-02-26. 2019. URL: https://www.nvidia.com/en-us/geforce/news/gfecnt/geforce-gtx-dxr-ray-tracing-available-now/.

[NVI24] NVIDIA *OptiX 8.0 Programming Guide*. Accessed: 2025-02-27. 2024. URL: https://raytracing-docs.nvidia.com/optix8/guide/index.html#preface#.

[NVI25] NVIDIA *CUB: CUDA Unbound*. Online. Accessed: 2025-02-27. 2025. URL: https://nvidia.github.io/cccl/cub/.

[Par+10] Parker, S. G., ET AL. OptiX: a general purpose ray tracing engine. *ACM Trans. Graph.* 29, 4 (July 2010). ISSN: 0730-0301. DOI: 10.1145/1778765.1778803. URL: https://doi.org/10.1145/1778765.1778803.

[TZB25] Tao, X., Zhang, A., AND Basudan, S. Efficient and Privacy-Preserving Multi-user, Multi-database and Multi-dimensional Range Query. In: *Data Security and Privacy Protection*. Ed. by Chen, X., Huang, X., AND Yung, M. Springer Nature Singapore, Singapore, 2025, 165–184. ISBN: 978-981-97-8546-9.

[Whi80] Whitted, T. An improved illumination model for shaded display. *Commun. ACM* 23, 6 (June 1980), 343–349. ISSN: 0001-0782. DOI: 10.1145/358876.358882. URL: https://doi.org/10.1145/358876.358882.

[Wik25] Wikipedia *Nvidia RTX*. Accessed: 2025-03-06. 2025. URL: https://en.wikipedia.org/wiki/Nvidia_RTX.

[Zhu22] Zhu, Y. RTNN: accelerating neighbor search using hardware ray tracing. In: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '22. Association for Computing Machinery, Seoul, Republic of Korea, 2022, 76–89. ISBN: 9781450392044. DOI: 10.1145/3503221.3508409. URL: https://doi.org/10.1145/3503221.3508409.