# RTIndeX: Exploiting Hardware-Accelerated GPU Raytracing for Database Indexing

Justus Henneberg
Johannes Gutenberg University
Mainz, Germany
henneberg@uni-mainz.de

Felix Schuhknecht
Johannes Gutenberg University
Mainz, Germany
schuhknecht@uni-mainz.de

## ABSTRACT

Data management on GPUs has become increasingly relevant due to a tremendous rise in processing power and available GPU memory. Similar to main-memory systems, there is a need for performant GPU-resident index structures to speed up query processing. Unfortunately, mapping indexes efficiently to the highly parallel and hard-to-program hardware is challenging and often fails to yield the desired performance and flexibility. Instead of proposing yet another hand-tailored index, we investigate whether we can exploit an indexing mechanism that is already built into modern GPUs: The raytracing hardware accelerator provided by NVIDIA RTX GPUs. To do so, we re-phrase the database indexing problem as a raytracing problem, where we express the dataset to be indexed as objects in a 3D scene, and point/range lookups as rays across the scene. In this combination, coined **RX** in the following, lookups are performed as intersection tests in hardware by dedicated raytracing cores. To analyze the pros, cons, and usefulness of the raytracing pipeline for database indexing, we carefully evaluate **RX** along *fourteen* dimensions and demonstrate its competitiveness and potential in a large variety of situations.

**PVLDB Reference Format:**
Justus Henneberg and Felix Schuhknecht. RTIndeX: Exploiting Hardware-Accelerated GPU Raytracing for Database Indexing. PVLDB, 16(13): 4268 - 4281, 2023.
doi:10.14778/3625054.3625063

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://infosys.informatik.uni-mainz.de/rtindex.

## 1 INTRODUCTION

Implementing performant index structures for highly-parallel GPU architectures is a challenging task [1–7, 16, 20, 23, 24, 28, 32, 33, 44, 49, 54, 57, 58]. But do we really have to implement a high-performing data structure from scratch? Can we maybe utilize the hardware indexing mechanism that is *already integrated* in modern GPUs?

### 1.1 Hardware Accelerated Indexing on GPUs

This indexing mechanism appears on NVIDIA's RTX workstation and consumer GPUs in form of a *raytracing hardware accelerator*. This accelerator enables rendering of ray-traced scenes in real time at a high frame rate. The concept is simple: A 3D scene contains many objects, which are usually approximated by thousands of triangles each, and a virtual camera observing the scene from a certain position. To create a realistic image, the GPU simulates the light rays entering the camera. For efficiency reasons, rays are cast in reverse, i.e., they originate at the camera lens and then travel in the direction the camera is facing until they hit the closest object. To speed up detecting intersections between rays and objects, graphics applications build a so-called *bounding volume hierarchy (BVH)* over all objects of the scene. Using the BVH, modern GPUs can perform the intersection test efficiently in hardware for a large number of rays in parallel using specialized raytracing cores that exist solely for this purpose.

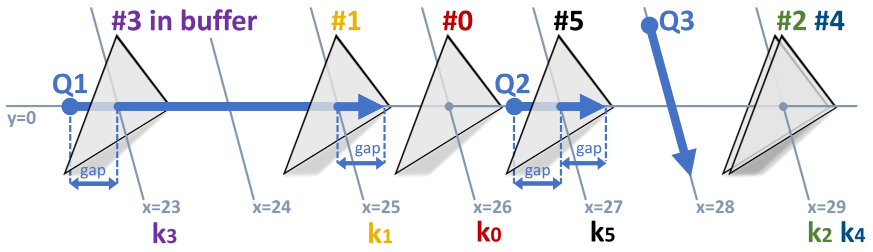
Conceptually, finding intersections this way is nothing but a hardware-accelerated indexing mechanism. While the 3D objects in the scene resemble a dataset, the BVH serves as an auxiliary index structure on top of it. Casting a ray resembles a lookup: If a ray intersects with an object, the lookup returns a unique identifier associated with the object. This enables us to map other indexing problems, such as database indexing, to this mechanism, and to exploit the built-in hardware acceleration for fast lookups. To create a secondary index on a table column, we express all entries in the column as 3D objects, ordered by their magnitude in the coordinate system of the scene. We associate each object with the rowID of the corresponding entry in the table, which is retrieved when a ray hits the object. To perform a lookup, we fire a ray through the area of interest, let the hardware detect all collisions, and return all associated rowIDs.

Unfortunately, expressing database indexing using this raytracing mechanism is not as straight-forward as it sounds at first glance and comes with a surprising amount of design choices to make. First, to encode each dataset entry as a 3D object, we use the powerful but non-trivial OptiX [15, 42] computing API, which allows us to freely program parts of the raytracing pipeline. As OptiX provides numerous options to set up the scene, for example in the types of primitives to use and the way intersection tests are performed, we can identify several drastically different ways to express database indexing. Second, the raytracing hardware imposes certain restrictions that we have to respect. For example, OptiX only supports single-precision floating-point numbers, while we want to index up to 64-bit wide integer keys. Consequently, we have to work around this problem, and again, there are multiple different ways to do so. Third, the raytracing mechanism is a proprietary implementation by NVIDIA, where details about the internal structure and behavior are intentionally not made available to the public [41]. Therefore, it is highly unclear how well the problem of database indexing

Proceedings of the VLDB Endowment, Vol. 16, No. 13 ISSN 2150-8097.
doi:10.14778/3625054.3625063

| rowID | Article | Category |
|---|---|---|
| 0 | Juice | 26 ($k_0$) |
| 1 | Bread | 25 ($k_1$) |
| 2 | Cookies | 29 ($k_2$) |
| 3 | Coffee | 23 ($k_3$) |
| 4 | Donuts | 29 ($k_4$) |
| 5 | Wine | 27 ($k_5$) |

**(a) Exemplary database table.**



**(b) Corresponding triangle arrangement for the *Category* column.**

**Figure 1: Visualization of how our indexing approach RX represents a secondary index on the *Category* column. For each key $k_i$ in *Category*, we create a triangle centered around the point ($k_i$, 0, 0), where the triangles are stored internally in the same order as the keys they represent. For each lookup, we fire a ray that is tested for intersection with all triangles. For example, the range lookup Q1 tests for all keys in the range $[23, 25]$ and consequently hits triangles #3 and #1, returning rowIDs 3 and 1.**

maps to the architecture and how it reacts to certain workloads, like dense/sparse key sets or the hit/miss ratio of lookups.

## 1.2 Contributions and Structure of the Paper

As a consequence of these observations, in the following work, we investigate whether and in which form hardware accelerated indexing can be used to realize database indexing on GPUs. We organize our work along *fourteen* different dimensions composed of *five* configuration dimensions and *nine* experimental dimensions:

First, we present how to re-phrase database indexing as a raytracing problem. Based on that, we discuss our implementation, coined **RX**, using the OptiX computing API. **RX** supports hardware-accelerated point and range lookups on 64-bit integer columns on NVIDIA RTX GPUs.

Second, we discuss the configuration options of **RX** along five dimensions: We implement (1) three different ways to express keys, (2) three different types of scene primitives to express the indexed dataset, (3) three different ways to express point and range lookups, (4) a flexible key decomposition, and (5) two different options to perform updates. Empirically, we identify the most suitable configuration that we will use throughout the rest of the paper.

Third, we perform an in-depth experimental evaluation where we compare **RX** against three GPU-resident index structures along nine experimental dimensions: We vary (6) the number of indexed keys and fired lookups, (7) the multiplicity of the indexed keys, (8) the order of the keys and lookups, (9) the batch size, (10) the hit/miss ratio, (11) the selectivity of range lookups, (12) the key size, and (13) the distribution of keys and lookups. Also, we (14) compare the performance on the three latest hardware architectures. Note that we perform a stand-alone evaluation of **RX** and its baselines to clearly identify the impact of the individual dimensions in isolation and without the interference of other system components.

## 2 DATABASE INDEXING → RAYTRACING

We start by re-phrasing database indexing as a raytracing problem. Figure 1 visualizes the high-level principle of the approach with a simple example: Assume we want to create a secondary index for the integer column *Category* of the exemplary database table shown in Figure 1a. To do so, we want to represent each key in the column by a corresponding primitive and associate it with its rowID. For simplicity, we limit the discussion to triangles for now and discuss other primitive types in Section 3.5.

## 2.1 Building the Index

To build the index, we first convert each key $k_i$ of the table column into a corresponding triangle $T_i$, where the 3D point $(k_i, 0, 0)$ should be a part of $T_i$. In other words, we are using the numerical *Category* value as our $x$-coordinate. In the 3D scene, this results in a line of triangles with gaps of varying sizes in between, as visualized in Figure 1b. If an entry occurs multiple times in the table (as is the case for key 29), multiple triangles will be created at the same location. One way of constructing $T_i$ is by slightly offsetting each of the three triangle corners in a different direction, e.g., $(k_i, -0.5, -0.5)$, $(k_i + 0.5, -0.5, 0.5)$, and $(k_i - 0.5, 0.5, 0.5)$. OptiX requires all triangles to be stored in a so-called *vertex buffer*. The position at which triangle $T_i$ is stored in the vertex buffer is not arbitrary, but must correspond to its rowID $i$, as this position serves as the unique identifier that is returned by OptiX if a collision with $T_i$ is detected. Once the vertices are arranged in the buffer, we can pass it to `optixAccelBuild()` to generate the BVH. A BVH is a tree-like data structure in which the individual triangles form the leaves of the tree. These triangles are then combined into small disjoint groups. For each group, the BVH stores a three-dimensional cuboid, called a *bounding volume*, which encloses all triangles in the group. These bounding volumes are then iteratively grouped and enclosed in the same way until only one group remains, forming the root of the tree. Figure 2 visualizes a bounding volume hierarchy for seven triangles in two dimensions. On modern RTX GPUs, both the traversal of the BVH as well as the intersection tests between the ray and the candidate triangles in the bounding volume are hardware-accelerated.

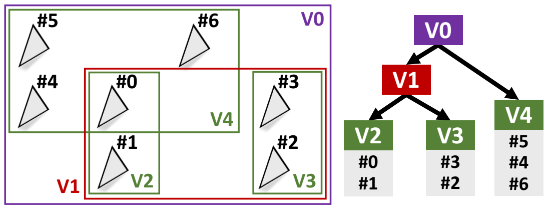


**Figure 2: Exemplary bounding volume hierarchy for a (two-dimensional) triangle arrangement.**

## 2.2 Performing Lookups

Now that the BVH is prepared, let us see how we utilize OptiX to answer lookups, where we start with the more general range lookups. Conceptually, to answer a range lookup on the *Category* column, such as $[23, 25]$ (**Q1**), we have to cast a ray along the line of triangles, starting just before the $x$-coordinate 23 and ending right after the $x$-coordinate 25. In Figure 1, the range lookup ray hits triangles #1 and #3, implying that rows with rowID 1 and 3 in the original table satisfy the range predicate. To realize point lookups, we can either formulate single-key range lookups (as shown in **Q2**), or cast a short ray perpendicularly to the line of triangles (as shown in **Q3**). In practice, we can answer a large number of range lookups $[l^{(i)}, u^{(i)}]$ concurrently to exploit the parallel nature of the hardware. Here, $l^{(i)}$ and $u^{(i)}$ denote the inclusive lower bound and upper bound, respectively, of the $i$-th range lookup. We formulate our batch of range lookups by specifying a corresponding ray for each pair of bounds, where each ray consists of a three-dimensional *origin point* $o$ and a *direction vector* $d$. A ray intersects a triangle $T$ if there exists a $t > 0$ such that the point $p = o + t \cdot d$ is part of the triangle $T$, where $t$ is called the *intersection parameter*. Note that we can restrict the intersection range by providing two additional parameters, $t_{\min}$ and $t_{\max}$. In this case, we will only detect intersections that also satisfy $t_{\min} < t < t_{\max}$.

To implement the lookups in OptiX, we have to set up a programmable OptiX pipeline. The pipeline consists of multiple user-provided functions, called *programs*, as well as some additional configuration options, and it can be launched similar to a CUDA kernel from the host CPU. When we start the pipeline, it spawns a CUDA thread for each lookup, where each thread calls the *ray generation program*. Therein, we concurrently convert each lookup range into the two ray parameters $o$ and $d$, then pass those to the `optixTrace()` API function to initiate the hardware-accelerated tracing procedure. Precisely, for range lookup $[l^{(i)}, u^{(i)}]$, `optixTrace()` receives $o = (l^{(i)} - 0.5, 0, 0)$, $d = (1, 0, 0)$, $t_{\min} = 0$, and $t_{\max} = u^{(i)} - l^{(i)} + 1$, along with a reference to the pre-computed BVH. To obtain intersection information, we also have to define the so-called *any-hit program*, which is called when the tracing procedure finds a ray-triangle intersection. The any-hit program receives the offset of the triangle within the vertex buffer, which corresponds to the rowID in the original table, for further processing.

## 3 DESIGN CHOICES

After discussing the core principle, let us discuss five fundamental design choices we face. We carefully evaluate all options to identify the strengths and weaknesses of each choice. Before that, let us introduce our evaluation setup.

### 3.1 Experimental Setup and Methodology

As we purely target GPU-resident data management, which becomes increasingly attractive due to an increase in available GPU memory, we assume there exists an array containing our key set in GPU memory. From this array, we construct the actual index, where each key's rowID is determined by its position in the array. Looking up a key in the index returns a set of rowIDs, which we subsequently use to retrieve values from a second GPU-resident array of the same size. This simulates the typical usage of a secondary index. As a final result, we compute the sum of all retrieved values. In our evaluation, we perform both *point lookups*, where we look up an exact key $k$ in the index, as well as *range lookups*, where we look up all keys within a range $[l, u]$. As mentioned, we always perform batch lookups to utilize the parallel nature of the hardware. In this case, all results for the batch of lookups are stored in a corresponding result array. Note that if a lookup does not return any rowIDs, a reserved *miss value* is written into the result array instead.

In our initial set of experiments, we fill the key array with $2^{26}$ consecutive unsigned 32-bit integers, starting at zero, where the keys are shuffled arbitrarily. We use a dense key set here to ensure a predictable number of hits – later on in Section 4, we will evaluate sparse key sets, duplicate keys, and varying hit rates as well. To generate the point lookups, we uniformly and randomly choose keys from the key array. For range lookups, we also uniformly pick a lower bound $l$ from the key array and increase it by the desired number of hits to generate the upper bound $u$. In total, we generate $2^{27}$ lookups for every experiment and fire them in a single batch unless specified otherwise. As we will see in the evaluation, only batch processing workloads, which, for instance, arise naturally in index-based joins, are able to fully saturate the GPU. We always report the average of five runs (after an initial "warmup run" to check for correctness and to ensure that the GPU does not wait for the release of resources anymore). Regarding hardware, our system contains an NVIDIA RTX 4090 GPU with 24 GB of VRAM and 128 raytracing cores. This GPU implements the most recent Ada Lovelace architecture and is the fastest consumer RTX GPU currently available. In Section 4.10, we compare the performance with three other GPUs of two older RTX architectures.

Note that whenever experiments require a deeper investigation, we use the following two GPU profiling tools: NVIDIA's *Nsight Systems* [38] tool can visualize the order and run time of GPU activities, such as kernel launches and memory allocations. Another tool, *Nsight Compute* [37], provides detailed hardware metrics for individual kernels and for the user-programmable parts of the raytracing pipeline. Unfortunately, Nsight Compute does not provide a cost breakdown for the fixed-function parts of the raytracing pipeline. However, we ran experiments to ensure all memory counters, which we frequently rely on, cover the pipeline end-to-end.

### 3.2 How Can We Express Keys?

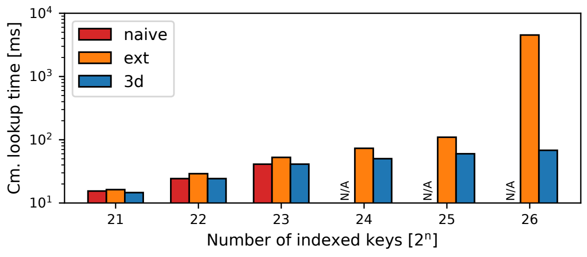
The first configuration dimension centers around how we can express our keys in a 3D scene. This question is more challenging than it seems since the straightforward implementation we described so far omitted an inconvenient detail: OptiX only supports single-precision floating-point numbers (`float32`) to represent 3D coordinates. As a consequence, simply casting a 32-bit integer key (or even a 64-bit key) to a floating-point coordinate will result in a loss of precision, and therefore, wrong results. In the following, we propose three different options to still express keys. With each presented option, we extend the supported key range up to 64-bit.

**Naive Mode**. We start with the Naive Mode, where the high-level idea is to naively express an integer key as a `float32` vertex coordinate. To understand the problem with this approach, let us look at how the integer 22 can be represented as a `float32`. The binary representation of 22 is $(10110)_2$ or $(10110.0)_2$ with a binary point. To store this number as a `float32`, we first convert it into its normalized form, which means the binary point is shifted next to the most significant bit. In our example, this results in a shift of four positions, which we can express as $(10110.0)_2 = 2^4 \times (1.011)_2$.

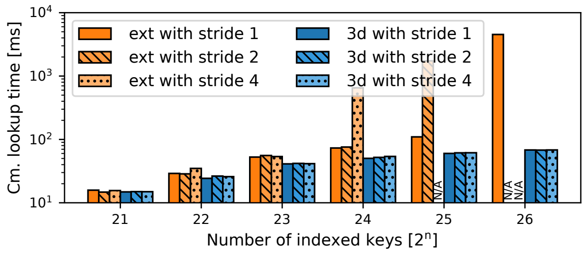
**Table 1: Overview of our proposed order-preserving methods for converting integers to floating-point numbers.**

| Mode | Distinct Keys | Conversion Formula | Gap Creation | Triangles | Spheres | AABBs |
|---|---|---|---|---|---|---|
| **Naive** | $2^{23}$ | $k \mapsto (\texttt{float}(k), 0.0, 0.0)$ | $\pm 0.5$ | Y | Y | Y |
| **Extended** | $2^{29}$ | $k \mapsto (\texttt{bit\_cast<float>}(2k + C), 0.0, 0.0)$ | `nextafter()` | Y | N | Y |
| **3D** | $2^{64}$ | $k \mapsto (\texttt{float}(k_{22:0}), \texttt{float}(k_{45:23}), \texttt{float}(k_{63:46}))$ | $\pm 0.5$ | Y | Y | Y |

Here, 4 is called the exponent $e$, whereas $(1.011)_2$ is called the significand $m$. In a `float32`, the (signed) exponent is represented using 8 bits, whereas the significand can have at most 24 bits. Unfortunately, with a significand of 24 bits, the contiguous range of non-negative integers that can be stored in a `float32` is at most $2^{24}$. However, the situation is even worse: OptiX requires us to leave a gap between the start/end of the ray and the adjacent triangles for the hit to be registered (see Figure 1b). Thus, we also need to make sure that for each key $k$, $k \pm 0.5$ can be represented as a `float32`. Therefore, we have to conservatively restrict the key range to $2^{23}$. Only then, querying the very last key $2^{23} - 1$ will work, as we can express $t_{max} = 2^{23} - 1 + 0.5$ without a loss of precision. This would not be the case for $2^{24} - 1$, where $t_{max} = 2^{24} - 1 + 0.5$ cannot be represented.



**(a) Standard conversion.**



**(b) Introducing stride.**

**Figure 3: Effects of key representations on lookup time. Notice the irregular behavior of Extended Mode ("ext").**

**Extended Mode**. So far, we have converted each integer key $k$ to its corresponding `float32` representation. This limited our supported key range to $2^{23}$. However, the range of floating point numbers that can be represented by a `float32` is significantly larger than $2^{23}$. To exploit this larger range, we need an order-preserving mapping from an integer key $k$ to a corresponding floating point number $f$. We propose the following mapping: Each integer key $k$ is mapped to the $2k$-th representable floating-point number using `bit_cast<float>(2k)`. Mapping to every second `float32` ensures that there is always a "gap value" between adjacent keys. The gap values next to a key $k$ can be identified by passing $k$ to the `nextafter()` function from the C standard library (instead of computing $k \pm 0.5$). Finally, note that we offset $2k$ by a constant $C$ prior to casting, as not offsetting yields wrong results due to implementation details related to `float32` processing. We found $C =$ `bit_cast<uint32_t>(0.5f)` to produce correct results for all keys up to $2^{29}$.

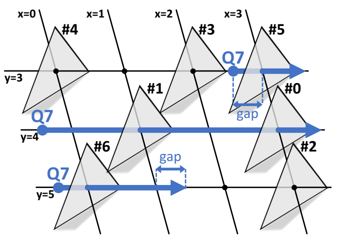
**3D Mode**. While we can already express $2^{29}$ distinct keys in Extended Mode, a general-purpose index structure should be able to operate with 64-bit keys. To achieve this, and since all vertices in OptiX are three-dimensional anyway, we now decompose the key bits into three smaller integers. We covert these integers to `float32` individually, and use them as three-dimensional coordinates to center each triangle around. In our case, for a 64-bit key $k$, we use the 23 least significant bits as the $x$ coordinate (written as $x = k_{22:0}$), the next 23 bits constitute the $y$ coordinate, and the remaining 18 bits form the $z$ coordinate. In Section 3.4, we evaluate other decompositions as well. To support 32-bit keys using this method, we extend each key to 64 bits by padding it with zeros. This mode is identical to Naive Mode for all keys smaller than $2^{23}$.

Note that this approach requires slight modifications to point lookups and range lookups. For point lookups, the origin $o$ now requires a three-dimensional offset. For range lookups, a single ray might now be insufficient, since the triangles do not form a single "line" anymore, but are scattered across the 3D scene. Instead, we have to cast distinct rays for each integer between $l_{63:23}$ and $u_{63:23}$. If a range lookup spans at most $2^{23}$ integers, it can be answered by casting only one or two rays. Figure 4 demonstrates how to answer the range lookup $[l, u] = [15, 21]$ (**Q4**) in 3D Mode. To visualize the principle in the example, we assume to have only two dimensions, where the $x$ coordinate is determined by the two least significant bits of the corresponding key, and the $y$ coordinate by all remaining bits. Table 4a shows the indexed keys along with their $x$ and $y$ coordinates in decimal representation. We first split the upper and lower bounds into their coordinates, i.e., $l = 15$ into $x_l = l_{1:0} = 3$ and $y_l = l_{63:2} = 3$, and $u = 21$ into $x_u = u_{1:0} = 1$ and $y_u = u_{63:2} = 5$. As the range of interest along the $y$-axis ranges from 3 to 5, we have to fire three corresponding rays in parallel to the $x$-axis. The first ray we fire starts at $x_l - 0.5 = 2.5$, the last ray ends at $x_u + 0.5 = 1.5$, whereas all intermediate rays (for $y = 4$ in this example) are unbounded, i.e., they hit all triangles along that line.

Let us now see how the three key conversion methods perform in comparison. Figure 3a shows the cumulative lookup times when varying the build size from $2^{21}$ to $2^{26}$. As we can see, the lookup times are very similar between all three conversion methods, apart from one oddity: With Extended Mode, lookup takes an extraordinary amount of time as soon as the build size exceeds $2^{25}$. One could suspect that the *magnitude of keys* might be responsible, since the numbers produced in Extended Mode become very large at some point, up to around $2^{62}/10^{18}$. We tested this suspicion by multiplying each key by $\pm 2^{20}$ after conversion, but for all conversion methods, the lookup time was identical. Instead, the *value range*

| # | $k$ | $x_k = k_{1:0}$ | $y_k = k_{63:2}$ |
|---|---|---|---|
| 0 | 19 | 3 | 4 |
| 1 | 17 | 1 | 4 |
| 2 | 23 | 3 | 5 |
| 3 | 14 | 2 | 3 |
| 4 | 12 | 0 | 3 |
| 5 | 15 | 3 | 3 |
| 6 | 20 | 0 | 5 |

(a) Keys.



(b) Visualization.

**Figure 4: Answering the range lookup [15,21] in 3D Mode (simplified to two dimensions).**

*of keys* appears to be responsible for this behavior (i.e., the ratio $q$ between the largest and the smallest inserted key), and we introduced *key stride* to verify this: Instead of inserting keys 1, 2, 3, etc., we insert $1s$, $2s$, $3s$, etc., with stride parameter $s = 1$, $s = 2$, and $s = 4$ in Figure 3b. The lookup times increase drastically as soon as $q$ hits $2^{26}$, and for any larger $q$, the experiment timed out after five minutes. This is confirmed by profiling, which reveals that in Extended Mode, the raytracing pipeline loads 76× more data from main memory, and 93× more data from the internal L2 cache when compared to 3D Mode, despite the build/lookup setup being identical. This indicates that the BVH has to traverse more triangles internally, implying that less triangles could be excluded outright, and cache bandwidth becomes the bottleneck.

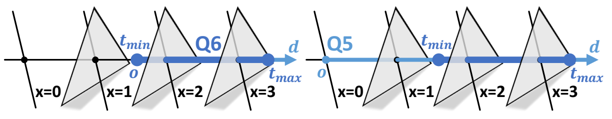
**Selected Configuration**. In conclusion, only 3D Mode can represent 64-bit keys, and it also exhibits stable scaling behavior. Therefore, we will use this mode as the default key conversion method for all subsequent experiments.

**Handling other data types**. Before continuing, we want to emphasize that **RX** does not only support unsigned 64-bit integers, but can handle other data types as well. All native C data types can be mapped to a `uint64` while preserving their relative order (this technique is traditionally used in radix sorting), and can therefore be indexed by **RX**. In particular, `float32` and `float64` values should always be converted in this way, and never be indexed directly, since the ratio $q$ between the smallest and largest value might be very large, leading to immense slowdowns. Composite data types (structs, arrays, strings) can still benefit from **RX** if their natural ordering is lexicographic. For these data types, we can convert the first few components (e.g., the first eight characters in a string) into unsigned integers individually, then densely pack them into a single 64-bit integer that can be passed to **RX**. This results in hardware-accelerated point and range lookups for the first 64 bits of the data type. The remaining components have to be compared and filtered in software.

## 3.3 How Should We Cast Rays for Lookups?

Next, let us discuss and evaluate the options we have in casting rays for range and point lookups. Let us first look at answering a range lookup $[l, u]$, which also covers point lookups by setting $l = u$.

Figure 5 shows the two options we have to look up the range [2, 3]: In **Parallel from offset** (Figure 5a), which we used so far, the ray (**Q5**) originates at $l - 0.5$ and ends at $u + 0.5$. Thus, the origin of the ray is offset from zero. In **Parallel from zero** (Figure 5b), the ray (**Q6**) always originates from 0 and ends after $u + 0.5$. To avoid false positives, the ray parameter $t_{\min}$ is set to $l - 0.5$. For point



(a) Parallel from offset. (b) Parallel from zero.

**Figure 5: Two ways of expressing the range lookup [2, 3].**

lookups, we have a third option available: Using **Perpendicular** (Figure 1b), the ray always targets only one specific triangle and is fired from a perpendicular angle to it. Table 2 summarizes the three options with their respective ray parameters. Note that in 3D Mode, we additionally shift the origin to the correct $y$ and $z$ coordinates (see Section 3.2).

**Table 2: Ray configuration parameters.**

| Method | $o$ | $d$ | $t_{\min}$ | $t_{\max}$ |
|---|---|---|---|---|
| **Para. offset (Q5)** | $(l - 0.5, 0, 0)$ | $(1, 0, 0)$ | 0 | $u - l + 1$ |
| **Para. zero (Q6)** | $(0, 0, 0)$ | $(1, 0, 0)$ | $l - 0.5$ | $u + 0.5$ |
| **Perpend. (Q3)** | $(l, 0, -0.5)$ | $(0, 0, 1)$ | 0 | 1 |

In Figure 6, we first evaluate whether point lookups should be expressed as parallel rays or as perpendicular rays. Note that Extended Mode does not support offsetting the ray origin due to `float32` precision limits. Therefore, we only test rays starting from zero in this experiment. We can clearly see that perpendicular rays consistently yield better lookup times than parallel rays. This is because the parallel ray, by definition, intersects with the majority of the bounding boxes and has to rely on $t_{min} < t < t_{max}$ to exclude potential hits, while the perpendicular ray misses most bounding boxes by default.

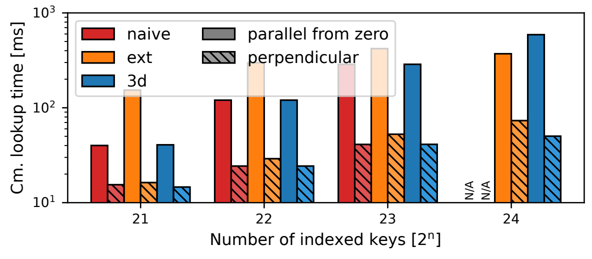


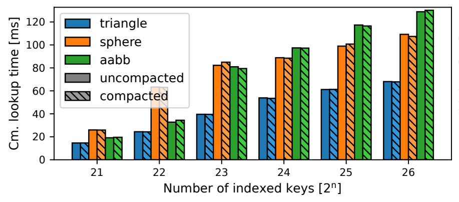
**Figure 6: Lookup time for parallel and perpendicular rays.**

To identify which choice works better for range lookups, in Table 3, we compare the answering time for offset rays, and rays starting at zero. Note that we evaluate only 3D Mode here, since Extended Mode does not support rays with offset, and Naive Mode works almost identically to 3D Mode. We can see that offsetting the origin pays off in all cases over keeping the origin at zero.

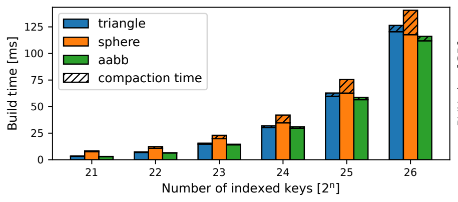
**Selected Configuration. RX** uses perpendicular rays for point lookups and rays with an offset origin for range lookups.

**Table 3: Lookup time for two choices of ray origin for range lookups in 3D Mode.**

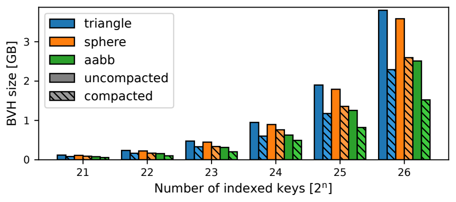
| Number of hits | 1 | 4 | 16 | 64 | 256 |
|---|---|---|---|---|---|
| Parallel from offset [ms] | 61 | 197 | 580 | 2086 | 8025 |
| Parallel from zero [ms] | 61 | 1209 | 1652 | 3382 | 10196 |



**(a) Lookup performance.**



**(b) Build performance.**



**(c) Memory footprint.**

**Figure 7: Comparison of primitive types.**

## 3.4 How Can We Decompose the Key?

In Section 3.2, we have decomposed each key $k$ in 3D Mode as $x = k_{22:0}$, $y = k_{45:23}$, and $z = k_{63:46}$. This decomposition allowed us to support 64-bit keys. Since other decompositions are possible as well, in the following, we test different configurations to see their impact on the performance. From Figure 8, we can see that the choice of decomposition affects lookup performance. When we allocate bits to the $x$ and $z$ components (bars on the right side), the lookup time increases. Remember that we always fire rays along the $z$-axis to answer point lookups. Assigning more bits to the $z$ component means that triangles will increasingly stack along the $z$-axis, which effectively turns the perpendicular ray into a parallel ray (see Section 3.3). In contrast, when all triangles satisfy $z = 0$ (bars of the left side), there is only one layer of triangles from the perspective of the ray, and lookups are fast. Note that we also tested a variant where the keys are not densely packed, but uniformly picked from the entire 64-bit space. In this situation, as expected, the lookup time was not affected by the decomposition at all, since there is no dimensional clustering present in the key set.

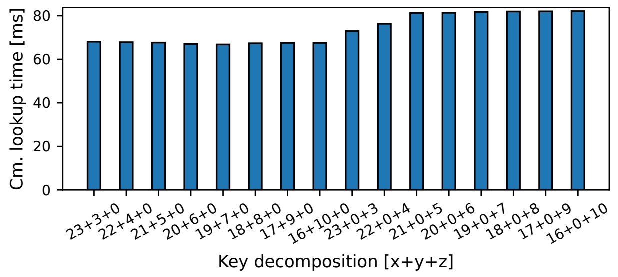


**Figure 8: Point lookups under varying key decompositions.**

In Figure 9, we additionally evaluate different decompositions for range lookups. We can see that the more the bits are assigned to the $x$-dimension, the better the lookup time. Maximizing the number of bits for the $x$-component also reduces the risk of having to cast multiple rays for wide ranges (see Figure 4b), and if multiple rays must be cast, their number is minimized.

**Selected Configuration. RX** uses the decomposition $x = k_{22:0}$, $y = k_{45:23}$, and $z = k_{63:46}$ throughout the rest of the paper, which yields good results for both point and range lookups.

## 3.5 Which Primitive Type Is Ideal?

So far, we have discussed how to express our keys using triangles. As each triangle is stored as nine `float32` (three vertex coordinates for each of the three vertices), let us see whether other primitive types offer a better memory footprint and/or performance. Apart from triangles, OptiX supports *spheres* and *axis-aligned bounding boxes (AABBs)*.

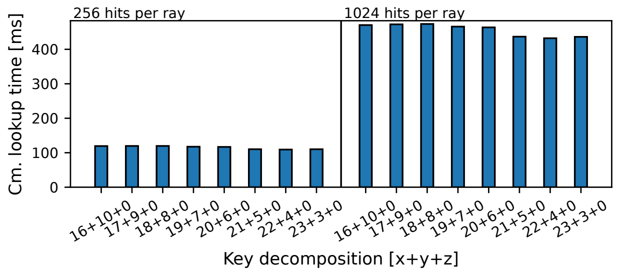


**Figure 9: Range lookups under varying key decompositions.**

**Spheres**. In contrast to triangles, which are planar, spheres are curved surfaces defined by their center point and their radius. A ray-sphere intersection can only occur when the ray enters or exits the volume of the sphere at some point. To ensure that a ray can always start outside of a sphere, we must avoid packing the spheres too densely. Just like with triangles, rays have to start and end in the gaps between adjacent spheres. To leave a sufficiently large gap, we choose $r = 0.25$ as the radius for each sphere. Since the radius is uniform for all spheres, OptiX allows us to specify the radius for all spheres at once. Consequently, each sphere only requires three `float32` to store the center, making this representation comparatively space-efficient.

**AABBs**. Like spheres, AABBs delimit a volume, and only register an intersection when a ray strikes one of the six faces. AABBs are intended to enclose user-defined primitives, e.g., implicitly defined surfaces, allowing them to be part of a larger 3D scene without requiring them to be supported by OptiX natively. Consequently, the user is expected to provide their own *intersection program* to figure out if a ray actually hits the object enclosed by the bounding box. In our case, it is sufficient to move the contents of the any-hit program into the intersection program, and not report a hit in the end. Internally, each AABB is represented by two corner points on opposite sides, requiring six `float32` in total and making AABBs more space-efficient than triangles.

Let us now compare the cumulative lookup time, the build time, and the memory footprint for all three primitive types in Figure 7. We show the results for both the (default) uncompacted variant, as well as when an additional compaction step via `optixAccelCompact()` is performed afterwards. Regarding lookup performance, triangles clearly perform best with a significant margin. We attribute this to the fact that the ray-triangle intersection test is implemented in hardware [12], utilizing the raytracing cores, whereas spheres and AABBs both call a software-based intersection program. Both the uncompacted and the compacted variant perform almost identically for all primitive types. In terms of BVH build time, AABBs perform best, closely followed by trianges, while sphere BVHs take longer to create. Compacting the structure after building it is cheap for all three methods, where especially for triangles, the overhead is negligible. Unfortunately, the memory footprint of (uncompacted) triangles is the highest in comparison. At the same time, we can see that the memory footprint decreases by up to 50% under compaction. Surprisingly, a sphere BVH takes the most time to compact, and the final memory footprint is the largest of the three primitives.

**Selected Configuration**. When optimizing for lookup performance, triangles should be preferred in order to fully utilize hardware acceleration. When optimizing for memory footprint, AABBs should also be considered. As we prioritize lookup performance, **RX** uses triangles. Also, we compact the BVH in all cases.

## 3.6 How Do We Perform Updates?

Next, let us discuss how we perform updates on an already existing index. OptiX natively supports in-place updates to bounding volume hierarchies via `optixAccelBuild()`, albeit with a few restrictions [13]: A special flag has to be set during construction, which disables the effects of compaction. Further, updates still require additional temporary memory, and updates cannot add new primitives or remove existing primitives.

To evaluate OptiX' update behavior, we build the BVH as described earlier and set the aforementioned update flag. We then test two different update workloads: (a) We swap pairs of adjacent positions in the buffer. Since keys are not sorted in the buffer, this simulates updates that significantly change keys. (b) We swap pairs of (rank-)adjacent keys. This simulates updates that change keys by $\pm 1$, since our key set is dense. In both cases, the set of keys itself does not change, only the keys' position within the buffer changes. As such, we would expect the lookup time to remain identical. After applying updates to the key buffer, we again convert all keys into triangles and update the BVH via `optixAccelBuild()`. Note that the triangle buffer does not only contain the updated primitives, but also the untouched ones. After the updates have been applied, we perform the usual lookup phase.

**Table 4: Update and lookup time in milliseconds for $2^{26}$ keys when swapping adjacent buffer positions and adjacent keys.**

| Experiment | Phase | $2^4$ | $2^8$ | $2^{12}$ | $2^{24}$ | rebuild |
|---|---|---|---|---|---|---|
| Swap adj. positions | Updates | 38.7 | 38.8 | 38.7 | 39.6 | 126.5 |
| | Lookups | 68.1 | 129.1 | 5361.3 | - | 68.1 |
| | Total | 106.8 | 167.9 | 5400.0 | - | 194.6 |
| Swap adj. keys | Updates | 39.5 | 39.5 | 39.5 | 39.5 | 126.5 |
| | Lookups | 68.1 | 68.2 | 68.2 | 68.2 | 68.1 |
| | Total | 107.6 | 107.7 | 107.7 | 107.7 | 194.6 |

From Table 4, we can make a set of interesting observations: (1) The time required to update the structure is independent from the number of applied updates. This is the case since the entire buffer must be passed to the update routine – not only the updated entries. (2) Updating is still more than three times faster than rebuilding the BVH from scratch. This suggests that the BVH is not completely rebuilt, but the existing bounding volumes are merely adjusted. (3) This adjustment can drastically impact the lookup time. When swapping more than $2^8$ keys in adjacent buffer positions, the adjusted bounding volumes are much larger than before, increasing the amount of required intersection tests. In such a situation, a full rebuild should be preferred. This behavior of adjusting the BVH is confirmed by the stable lookup times when swapping rank-adjacent keys instead of adjacent positions. A profiler can provide more insight via memory statistics: Although the amount of main memory accesses does not differ significantly between the two update variants, swapping adjacent positions leads to an immense increase in L1/L2 cache reads, while the increase is barely noticeable when swapping adjacent values. Again, this indicates that the BVH cannot exclude as many potential hits as before. The OptiX documentation confirms this behavior [13], claiming that the quality of the BVH will degrade substantially when too many triangles are relocated.

**Selected Configuration**. We conclude that updates in **RX** should be realized via a full rebuild in favor of lookup performance.

## 4 EXPERIMENTAL EVALUATION

After identifying a reasonable and well-performing configuration, we will now vary the experimental setup along nine dimensions and compare **RX** against three traditional GPU-resident index structures as baselines. The experimental setup in this section is the same as in Section 3.1, however, instead of restricting the key set to consecutive integers, we now permit the full 32-bit integer range as keys (the B+-Tree baseline does not support 64-bit keys). Any changes to this setup will be stated explicitly.

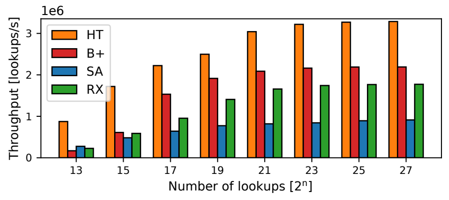
## 4.1 Traditional GPU Indexes as Baselines

We compare against the following GPU-resident index structures:

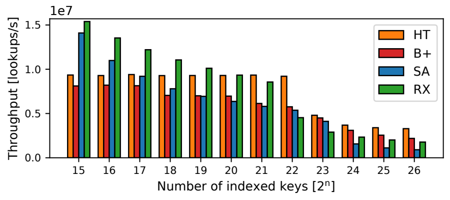
**HT**. Our first baseline is WarpCore, a state-of-the-art GPU hashtable [25, 26]. WarpCore implements *cooperative probing*, where each key is assigned to a group of threads during inserts or lookups, and each group accesses neighboring slots in the hashtable. This accelerates the task of identifying an empty slot for insertion, or discovering the key to be looked up, while still using the GPU's cache and load-store units to the maximum extent possible. The authors show that WarpCore outperforms other recent GPU hashtables, such as SlabHash [4] and cuDPP [2, 3], especially when the majority of slots is occupied. Just like the authors, we use a target load factor of 0.8 and fix the group size for cooperative probing to 8. Since there is no bulk-loading for hashtables, we insert each key separately during the build phase.

**B+**. Our second baseline is the state-of-the-art GPU B+-Tree by Awad et al. [6] in its recently updated version [7, 22]. This baseline traverses the tree in groups of 16 threads, so that lookups within a node can be done synchronously using warp intrinsics. The build phase sorts the keys using CUB's key-value `DeviceRadixSort` [14], an out-of-place GPU radix sort that is considered the fastest way to sort integers using a CUDA-enabled GPU, then bulk-loads the tree with the sorted key-value pairs. Unlike **HT** and **RX**, the B+-Tree only supports 32-bit keys. In comparison to a GPU LSM tree [5], the B+-Tree yields better lookup performance, making it an ideal baseline for the read-only benchmarks in this section. Note that we had to slightly modify the range lookup code for the B+-Tree in order to support efficient aggregation.

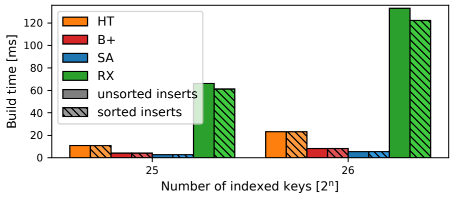
**SA**. Our third baseline is a sorted array, which we combine with a naive binary search for lookups. This mimics the access patterns that would occur during lookups in a balanced binary tree, while



**(a) Throughput (varying number of lookups).**



**(b) Throughput (varying build set size).**



**(c) Impact on the build time.**

**Figure 10: The scaling behavior of all indexing methods.**

requiring less overall memory. Insertions and deletions cannot take place after the index has been constructed. Still, this index is trivial to implement, and serves as a great baseline for more sophisticated approaches. Again, we utilize CUB's radix sort to sort the array.

## 4.2 Varying the Number of Lookups and the Number Of Indexed Keys

We first compare the scaling behavior, where we focus on lookup throughput and build time, along with memory footprint.

In Figure 10a, we first vary the total number of point lookups on the *x*-axis from $2^{13}$ to $2^{27}$ while keeping the number of indexed keys constant at $2^{26}$. In this experiment, **HT** clearly outperforms all other indexes. However, **RX** remains competitive in comparison to the other order-based index structures **B+** and **SA**. We can also observe that the throughput of all methods starts saturating at around $2^{21}$ lookups per batch. Below that, the workload is too small to fully utilize the GPU resources. Let us investigate this behavior for **RX**: Each processing element on an NVIDIA GPU (called *SM*) executes threads in groups of 32 (called *warps*). When running **RX**, a single SM can dynamically schedule up to 16 warps, which allows the GPU to hide memory latencies, similar to hyper-threading in modern CPUs. Assigning less than 16 warps to an SM means that the SM is more likely to idle while waiting for a memory dependency. At the same time, all SMs on the GPU are limited by the peak memory bandwidth, which remains under-utilized when the number of memory accesses is low. Table 5 shows the average number of active warps per SM and the percentage of the peak GPU memory bandwidth utilization: Increasing the number of lookups rapidly saturates the limit of 16 warps per SM, but also approaches the peak bandwidth at the same time, which eventually leads to constant throughput. Note that our default of $2^{27}$ lookups will always ensure that the GPU resources are fully utilized during each experiment.

**Table 5: Average number of active warps per SM and percentage of the peak GPU memory bandwidth utilization.**

| Number of lookups | $2^{13}$ | $2^{15}$ | $2^{17}$ | $2^{19}$ | $2^{21}$ |
|---|---|---|---|---|---|
| Active warps per SM | 3.89 | 6.68 | 12.46 | 13.79 | 14.25 |
| Memory BW [% of peak] | 39.61 | 61.36 | 75.12 | 77.97 | 78.96 |

In Figure 10b, we continue by varying the number of indexed keys from $2^{15}$ to $2^{26}$ while keeping the number of lookups constant at $2^{27}$. We can see that for smaller key sets of up to $2^{19}$ keys, **RX** shows the best lookup performance of all methods. With an increase in the number of keys, the throughput of **RX** unfortunately falls below **HT** and **B+**. Profiling provides an explanation for this behavior: When the build set is small, all methods read exactly the same amount of GPU main memory during the lookup phase, indicating that all index structures fit into the GPU's L2 cache. The profiler also shows that the relative performance of each index structure loosely correlates with the total number of executed instructions: For $2^{15}$ inserted keys, **RX** requires very few instructions, since the BVH traversal is done in hardware, while **B+** requires around 40× as many. On the other hand, when the size of the build set increases beyond $2^{20}$, the index structures no longer fit into the L2 cache, and the lookup performance is now bounded by GPU memory. **RX** and **B+** load a comparable amount of memory, but both load more than **HT** and **SA**. However, **SA** falls behind due to latency overhead from unfavorable (random) memory access patterns.

In Figure 10c, we also inspect the scaling of the build time when doubling the number of keys from $2^{25}$ to $2^{26}$ for both a sorted and an unsorted key set. **RX** scales linearly in this regard, however, the BVH creation is significantly more expensive than the build phase for the other indexes. This, in combination with the fact that updates perform poorly (see Section 3.6), indicates that **RX** should be primarily used as a read-only index structure.

**Table 6: Memory footprint for $2^{26}$ keys.**

| Memory Footprint | HT | B+ | SA | RX |
|---|---|---|---|---|
| Final size [GB] | 0.68 | 1.23 | 0.54 | 2.78 |
| Overhead during build [GB] | 0 | 1.35 | 0.81 | 4.37 |

With the build time in mind, let us also inspect the memory footprint of all methods when indexing $2^{26}$ keys. We differentiate between the space required during construction and the space required afterwards. From the results in Table 6, we can see that **RX** consumes considerably more space during construction than the traditional methods. After construction, the footprint shrinks noticeably, but is still around twice as high as for **B+**. This is a consequence of **RX** representing each key as a triangle, where other indexes can store each key as-is. **SA** consumes more space than **HT** during construction (caused by the out-of-place radix sort), but has zero structural overhead afterwards. **HT** consumes slightly more due to a 25% over-allocation to achieve its target load factor.

## 4.3 Varying the Key Multiplicity

So far, our key set was composed of unique keys only. In the following, we will introduce duplicates by varying the the key multiplicity and see the impact on the lookup time. In Figure 11, we vary the key multiplicity from $2^0$ (unique keys) in logarithmic steps to $2^8$ (every key appears 256 times) while keeping the number of point lookups constant. As the number of results for a point lookup increases with the number of duplicates per key, we normalize the obtained cumulative lookup time by dividing it by the number of duplicates per key. Note that we cannot show **B+** in this experiment, because it does not support key duplicates.

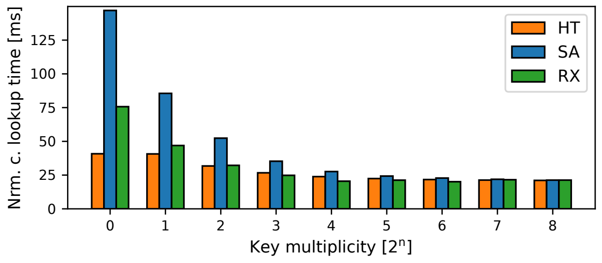


**Figure 11: Impact of key multiplicity on point lookups.**

From the results we can see that an increased key multiplicity generally favors all indexes. For **RX**, duplicate keys lead to the creation of multiple primitives located at exactly the same coordinates in the scene. Hence, they do not increase the size or complexity of the BVH in any way, but lead only to more ray intersection tests, which are carried out very efficiently in hardware. Since each lookup ray hits all duplicate primitives in one go, **RX** handles high key multiplicities well, and marginally wins the comparison for more than 4 duplicates per key. According to the profiler, the number of GPU main-memory loads equalizes across the indexes as the multiplicity increases to $2^7$. Retrieving the value associated with each key now overshadows the cost of traversal.

## 4.4 Ordering Inserts and Lookups

Until now, we assumed that the indexed keys are inherently unsorted. However, in practice, the build keys (and their associated values) could be pre-sorted in the column to index. Also, if sorting is cheap, sorting a batch of point lookups by their requested key might be beneficial. We therefore investigate the impact of sorted inserts and/or sorted point lookups over the unsorted alternatives. However, note that sorting is only possible if a sufficient amount of additional GPU memory is available.

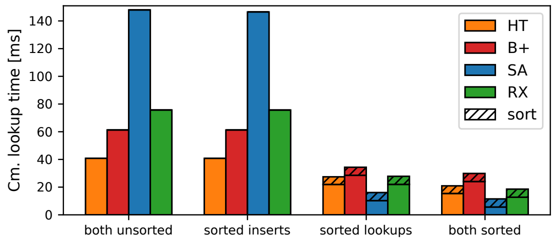


**Figure 12: Impact of sorted keys and sorted point lookups on lookup performance.**

For sorting, we again use CUB's `DeviceRadixSort`. In Figure 12, we analyze the impact on the cumulative point lookup time for all four combinations. When the lookups are unsorted, the build order does not influence lookup times at all. For the baselines, this is not surprising, since we know that they re-order the keys as part of the build process anyway, either by sorting (**B+** and **SA**) or by hashing (**HT**). For **RX**, the same independence is shown by our experiment: As the run time remains identical, it is very likely that keys are reordered during BVH construction. In contrast, sorting the point lookups positively impacts all indexes significantly. This can be attributed to improved memory locality when traversing the index structure, since neighboring lookups are likely to be answered simultaneously by neighboring threads. As a result, the number of GPU main-memory accesses decreases (between -45% for **HT** and -92% for **SA**), and just like we explained in Section 4.2, the number of instructions per lookup now limits the throughput. When both the build set and the lookups are sorted, this locality also extends to the value column: Neighboring threads look up keys with similar magnitude, which means their associated values will be close together in the sorted build set and even share the same cache line. Finally, note that GPU-resident sorting is surprisingly cheap in comparison to the actual lookups.

## 4.5 Varying the Batch Size for Lookups

Next, let us analyze the impact of splitting our $2^{27}$ lookups into multiple batches, which we fire consecutively. A submission pattern of multiple smaller batches occurs in practice if (a) only a limited number of lookups are waiting to be answered simultaneously, (b) the latency is required be small, or (c) we want to sort the lookups like in Section 4.2, but there is not enough space available to do it in one go.

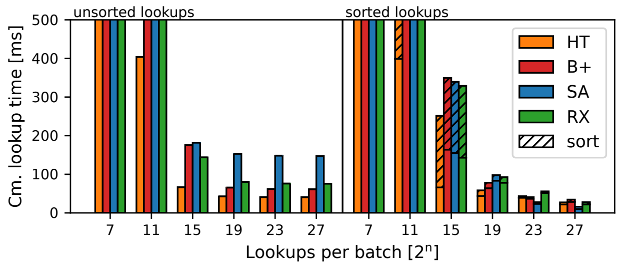


**Figure 13: Impact of batching lookups (capped at 500 ms).**

Thus, in Figure 13, we vary the number of batches to submit from 1 batch ($2^{27}$ lookups per batch) to $2^{20}$ batches (128 lookups per batch) and report the cumulative lookup time on the $y$-axis. Again, we evaluate both ordered and unordered lookups, to observe the effect of sorting. We can see that increasing the number of batches generally has a negative effect on all methods. While the performance for $2^0$, $2^4$, $2^8$, and $2^{12}$ batches remains relatively constant, the performance degrades heavily for more batches due to two reasons: (1) From $2^{16}$ batches onwards, each batch is too small to saturate GPU resources (see Section 4.2). (2) The total amount of overhead caused by launching CUDA kernels increases, since we have to perform one launch per batch. Further, we can see that sorting many small batches is more expensive than sorting few larger ones. In a separate experiment, we confirmed that the runtime of CUB's `DeviceRadixSort` stabilizes at a lower bound for batch sizes below $2^{20}$. Consequently, sorting lookups does not improve performance for batch sizes smaller than $2^{19}$. For **RX**, a sweet spot is reached for $2^{12}$ batches (32, 768 lookups per batch), where it surpasses the performance of the other order-based indexes **B+** and **SA**.

## 4.6 Varying the Hit Rate of Lookups

Up to this point, we ensured that all lookups were hits and therefore returned a non-empty result. However, depending on the workload, misses might occur as well, potentially affecting the performance. Thus, in Figure 14, we vary the hit rate $h$, i.e., the fraction of lookups that return a non-empty result, on the $x$-axis, and observe the impact on the cumulative lookup time on the $y$-axis. Again, we test both unordered and ordered lookups.

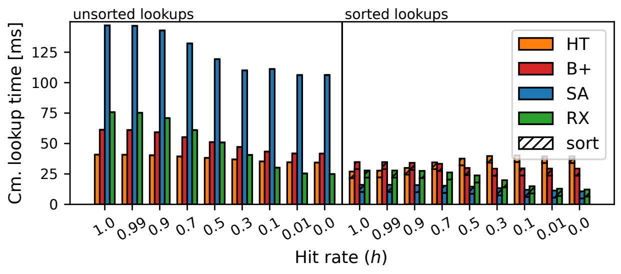
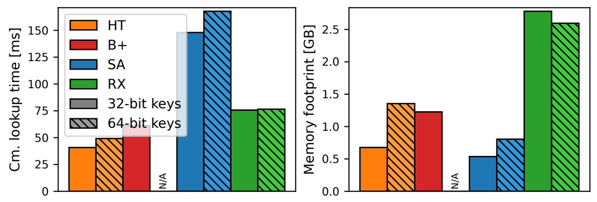


**Figure 14: Varying the hit rate $h$.**

With a decrease of $h$, for unsorted lookups, we observe a notable decrease in lookup time for all indexes except for **HT**. A constant portion of the decrease can be attributed to the fact that only hits require us to retrieve a corresponding value from the projected column, whereas each miss can skip this step. Apart from that, we see a drastically different impact of the hit rate on the different indexes. Most notably, **RX** runs disproportionately faster (up to 3× when going from $h = 1.0$ to $h = 0.0$), outperforming **B+** and **SA** for $h \leq 0.5$ and even **HT** for $h \leq 0.1$ under unordered lookups. Here we see the advantage of the BVH over regular search trees: The BVH traversal can be aborted as soon as no bounding volume at the next lower level covers the searched key. These early aborts can be seen during profiling as a disproportionate reduction in GPU main-memory accesses (-63% from $h = 1.0$ to $h = 0.0$). In contrast, aborting the traversal early is not possible on regular trees, which always have to do a full traversal. We confirmed that early abort is indeed the reason for the good performance of **RX** in an additional separate experiment, where we evaluated the extreme case that all misses lie outside the value range of the key column, i.e., each missed key is smaller/greater than the smallest/greatest key in the key set: In this situation, **RX** performs even better than shown in Figure 14, as the BVH traversal can already be aborted at the root node. Lastly, as expected, all methods are generally faster under ordered lookups with **SA** being the dominant method – however, as stated before, sorting is only an option if additional memory is available. In contrast, the performance of **HT** suffers when the hit rate decreases. **HT** implements open addressing, where a miss usually causes longer probe sequences than a hit. For sorted lookups, and excluding the sorting phase, a hit rate of $h = 0.0$ leads to 36% more instructions being executed than for $h = 1.0$, and the amount of GPU main memory accesses doubling.

## 4.7 Impact of the Size of Keys

So far, we focused on 32-bit keys, as **B+** does not support larger keys. Nevertheless, let us now extend the key size to 64 bits to see how the remaining indexes react.



**(a) Impact on the lookup times. (b) Impact on the index size.**

**Figure 15: Impact of the key size (32-bit vs 64-bit).**

In Figure 15, we compare the lookup time and index size for 32-bit and 64-bit keys. We also show the 32-bit results for **B+** as a point of reference. Regarding cumulative lookup time, Figure 15a shows that **RX** is unaffected by the increase in key size, since it does not differentiate between 32-bit keys and 64-bit keys when converting them to triangles. In contrast, **SA** and **HT** slow down under 64-bit keys, which we attribute to the increased cost of 64-bit integer comparisons on GPUs, in combination with the increased memory footprint of the underlying data structure. Figure 15b shows the memory footprint of all indexes. Since **RX** treats 32-bit keys like 64-bit keys during construction, the size of the BVH is mostly the same, apart from small random variations pertaining to the choice of inserted keys. In contrast, both **SA** and **HT** store each key in its original representation, and therefore, 64-bit keys lead to a noticeable increase in memory consumption.

## 4.8 Varying the Skew

We have seen how the indexes perform on a set of uniformly distributed keys, answering uniformly distributed lookups. In the following, we will introduce skew to both the lookup distribution as well as the key distribution and observe the effects on the indexes.

In Figure 16, we introduce skew to the lookups while keeping the indexed key set uniformly distributed. The keys of our lookups follow a Zipf distribution, where we vary the Zipf coefficient from 0.0 (resembling a uniform distribution) up to a very high skew of 2.0 on the $x$-axis and observe the cumulative lookup time on the $y$-axis. Again, we evaluate both sorted and unsorted lookups.

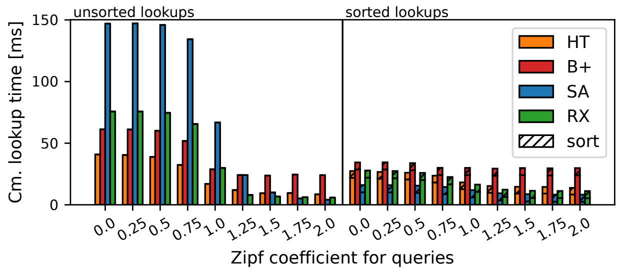


**Figure 16: Varying the skew of point lookups on uniformly distributed keys.**

From the results we can see that skewed lookups have a positive effect on the performance of all methods, which holds especially for unsorted lookups. We attribute this to improved access locality under skew, as the lookups now focus on a smaller key range. Again, the resulting decrease in memory accesses heavily benefits **RX** (see Section 4.2), such that **RX** outperforms the competing indexes under high lookup skew. Let us analyze this in detail by comparing **RX** and **B+**. When inspecting the cache hit rate for both methods while varying the skewness in Table 7, we can see that as long as the skewness is low, the cache hit rate is low as well, and both methods are bandwidth bound. In this case, **B+** is superior since it has to read less data from GPU main memory. However, as soon as the cache hit rate increases due to a higher skewness, both variants become compute bound. Now **RX** outperforms **B+** since it executes around 56× fewer instructions due to the hardware acceleration. With ordered lookups, the performance improvement is less noticeable, as sorted lookups already ensure good access locality (see Section 4.4). In a separate experiment, we introduced skew to the key distribution while retaining a uniform lookup distribution (only hits). However, all methods essentially remained unaffected by a skewed key set. This is reasonable because each index is able to partition the key set equally well no matter whether the keys are widely spread across the domain or not.

**Table 7: Impact of skewness on the data transfers and the number of executed instructions (unordered lookups).**

| Zipf coefficient | Hit rate L1/L2 [%] RX | Hit rate L1/L2 [%] B+ | Memory read [GB] RX | Memory read [GB] B+ | Instructions RX | Instructions B+ |
|---|---|---|---|---|---|---|
| 0.0 | 26/44 | 39/55 | 54.31 | 41.32 | 390M | 22B |
| 0.5 | 26/44 | 39/55 | 53.82 | 40.80 | 390M | 22B |
| 1.0 | 36/71 | 89/78 | 22.19 | 16.18 | 390M | 22B |
| 1.5 | 82/90 | 92/93 | 0.85 | 0.79 | 390M | 22B |

## 4.9 Answering Range Lookups

So far, we solely evaluated point lookups, where each lookup targeted at most one key, and **HT** is the fastest index most of the time. However, as soon as we want to answer range lookups, which are not supported by **HT**, the order-based indexes **RX**, **B+**, and **SA** can show their strengths. Thus, in Figure 17, we compare these indexes in terms of cumulative range lookup performance while varying the number of qualifying entries from $2^0$ (resembling point lookups) to $2^{10}$. Note that we normalize the cumulative lookup time by dividing it by the number of qualifying entries per range lookup.

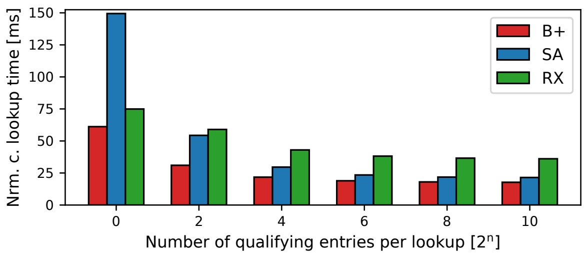


**Figure 17: Evaluating the cumulative range lookup time.**

To easily generate range lookups that return a specific number of qualifying entries, we build each index from a column filled with a dense shuffled key set containing *all* integers in $[0, 2^{26} - 1]$. On such a dense key set, looking up the range $[l^{(i)}, u^{(i)}]$ with a span of $s = u^{(i)} - l^{(i)} + 1$ will return exactly $s$ qualifying entries. Note that this setup represents the worst-case scenario for range lookups, as all potential keys within each range actually exist. Thus, it yields an upper bound for the execution time. From the result, we see that **B+** yields the best performance across all choices of $s$. **RX** initially outperforms **SA** for small range lookups, but then quickly loses its advantage. This can be explained by the fact that both **B+** and **SA** store the keys in an ordered fashion, and therefore, only have to locate the smallest qualifying key in the index. All other qualifying keys can be found by traversing the index structure sideways, until the first non-qualifying key is found, where **B+** implements sideways traversal through a linked list of leaf nodes. In addition, **B+** can utilize warp-level aggregation to accelerate the summation of values, giving it an additional advantage over its competitors. In contrast, **RX** has to identify each qualifying entry individually by detecting a collision with each triangle that represents a qualifying key. Also, we see that the normalized cumulative lookup time of **RX** decreases when the number of qualifying entries increases. This shows that the cost of BVH traversal remains rather constant while varying the number of qualifying entries, whereas the cost for ray intersection tests naturally increases.

With these results at hand, we can actually approximate the cost of both phases. Optimistically assuming that exactly one BVH traversal must be carried out per range lookup and that no interleaving takes place, we can create an overdetermined equation system with six equations from the obtained results, where the equation for $2^n$ qualifying entries has the form:

$$\textbf{LookupTime}(2^n) = \textbf{TraversalTime} + 2^n \cdot \textbf{IntersectTime},$$

containing the two unknowns **TraversalTime** and **IntersectTime**. Approximating a solution to this equation system using the method of non-negative least squares [31] yields 102.85ms for **TraversalTime** and 36.01ms for **IntersectTime**, implying that the cost for the BVH traversal dominates the cost for a ray intersection test.

## 4.10 Varying the Hardware Architecture

Until now, we evaluated all methods on the most recent Ada Lovelace GPU architecture. By testing the two previous architectures, Ampere and Turing, we can find out how much has changed over the generations, in particular given the varying number of available raytracing cores in different core generations. Table 8 provides an overview of the four test systems spanning three generations of RTX GPUs. While each system features a different CPU model, remember that all measurements are fully GPU-resident, and therefore, CPU performance can barely influence the results.

**Table 8: Evaluated GPUs and hardware architectures.**

| Sys. | GPU | Architecture | VRAM | RTX cores | CPU |
|---|---|---|---|---|---|
| **S1** | 4090 | *Ada Lovelace* | 24GB | 128 (3rd gen) | TR 3990X |
| **S2a** | A6000 | *Ampere* | 48GB | 84 (2nd gen) | i9 12900K |
| **S2b** | 3090 | *Ampere* | 24GB | 82 (2nd gen) | i7 11700K |
| **S3** | 2080Ti | *Turing* | 11GB | 68 (1st gen) | i9 9900K |

In Figure 18, we show the cumulative lookup performance for both sorted and unsorted point lookups on all four test systems. We observe a significant performance improvement over the three hardware generations due to an increase in memory bandwidth and number of CUDA cores. However, we can also see that while **RX** was not yet competitive on system **S3**, it is competitive on system **S1**. For sorted lookups, when going from **S3** to **S1**, **RX** shows the most improvement of 3.23×, far more than **HT** (2.41×), **SA** (2.33×), and **B+** (1.88×). We attribute this difference to a significant increase in the number and performance of the available raytracing cores, which doubled their throughput of ray intersection tests with every generation according to NVIDIA [39, 40]. For unsorted lookups, this effect is less pronounced, as all methods are limited by memory throughput. In this case, the improvement for **RX** of 3.25× is on par with that of **B+** (3.17×), followed by **HT** (2.39×). **SA** improved by 4.24×, but still performs noticeably worse than all other baselines, especially on older hardware. Since this trend of increasing the number of raytracing cores will likely continue in future generations (such as the upcoming *Grace Hopper* generation), **RX** might be able to outperform the traditional variants eventually.

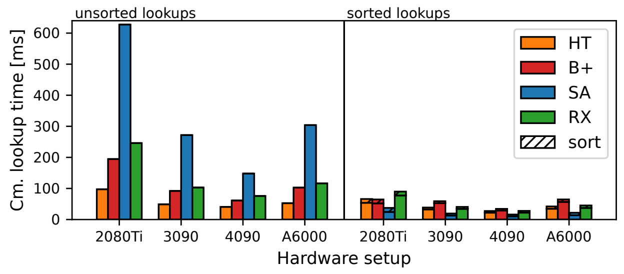


**Figure 18: Impact of hardware architecture on lookup times.**

## 5 RELATED WORK

Obviously, NVIDIA RTX was originally designed to accelerate the rendering of complex lighting effects, e.g., transparency, reflection, refraction, and drop shadows, in real-time. However, apart from this paper, there exist other works outside of rendering that utilized hardware acceleration on RTX-enabled GPUs in creative ways.

For example, one line of work uses raytracing to perform *point containment tests* [29, 36, 50, 55]. A point containment test determines whether a point lies inside the boundary of a polygon (in 2D) or a polyhedron (in 3D). One way to implement such a test is by finding the closest intersection with the boundary, which can be solved efficiently on RTX-enabled GPUs. Another area of application is *time-of-flight imaging* [35, 47, 53]. Time-of-flight sensors compute distances to surrounding objects by measuring the difference between transmission and reception of electromagnetic or acoustic pulses. Specialized software can then recreate a three-dimensional scene from a sufficient number of measurements. To test these systems, researchers can simulate the propagation of pulses produced by such a sensor with hardware-accelerated raytracing in a virtual environment. Also, the problem of performing an efficient *radius search* [18, 59] has been accelerated using RTX. A radius search locates all points within a fixed radius from a specified point. One can improve this search with a bounding volume hierarchy to quickly exclude far-away points, which can be delegated to the RTX cores. Further areas of application include physical simulations for *graph rendering* [56] and *particle movement* [8, 9, 51].

Note that raytracing cores are not limited to consumer and workstation GPUs, but also integrated into some data-center GPUs, including the NVIDIA T4, A2, A10, A16, A40, and L40, some of which can be accessed easily through various cloud providers. Future data-center GPUs might even incorporate more specialized versions of these cores, if data management applications start exploiting them.

Independent from NVIDIA RTX, in recent years, there is a trend of turning traditional index structures to *GPU-resident index structures*. Typically, this implies optimizations to the memory layout and access patterns, and changes to the memory allocation strategy. Occasionally, a GPU index requires profound algorithmic changes to alleviate contention issues caused by the enormous number of active threads. Note that some existing GPU indexes only allow lookups on the GPU, while construction and updates can only be done via the CPU. GPU hashtables [2–4, 24, 25, 32, 57] provide key-value mappings which yield top-of-the-line performance for point lookups, but require various degrees of over-allocation to perform efficiently, and cannot answer range lookups. If one only needs to test membership, Bloom filters [16, 23, 25] and quotient filters [20] require less space than hashtables, but membership tests can produce false positives. Radix trees [1] and comparison-based trees [5–7, 28] also manage key-value mappings, but additionally support range lookups. Trees generally perform worse than hashtables since tree traversal entails multiple cache misses, and some trees demand sophisticated in-GPU memory managers to dynamically allocate new nodes. While our comparison includes a state-of-the-art comparison-based tree, no code for the radix tree was freely available at the time of writing. If a column is limited to a small amount of discrete values, a GPU bitmap index [49] can also offer fast point and range lookups, while keeping a minimal memory footprint. Since we specifically devised **RX** to support all possible 64-bit values in Section 3, a fair comparison with bitmap indexes is not possible. Spatial queries require more specialized solutions, such as GPU R-Trees [44, 54] or GPU permutation indexes [33]. Since R-Trees also employ bounding volumes, it would have been compelling to compare a software R-Tree to our hardware-accelerated BVH approach. Unfortunately, at the time of writing, no R-Tree implementation was openly available. Finally, learned indexes [58] achieve high performance on GPUs, since they heavily depend on linear algebra operations, which have been extensively optimized on GPUs, and some GPUs even offer specialized accelerators for these operations. Regrettably, the corresponding implementation cannot be found online. Apart from indexing on GPUs, there is has been interesting work proposing entirely or partially GPU-resident DBMS architectures [10, 11, 17]. Also, other DBMS operations like joins [19, 30, 34, 43, 45] or grouping and aggregation [27, 48, 52] have been migrated successfully to GPUs.

Finally, apart from OptiX, DirectX [46] and Vulkan [21] provide specialized APIs to specifically target hardware-accelerated raytracing, which both support our proposed indexing scheme.

## 6 LESSONS LEARNED & CONCLUSION

We presented **RX** and showed that database indexing can indeed be expressed as a raytracing problem to utilize the built-in hardware acceleration of RTX GPUs. We analyzed five design dimensions and empirically evaluated that by splitting 64-bit keys into three parts (3D Mode) and by using these parts as vertex coordinates for a compacted triangle BVH, we achieve a good trade-off between space utilization and lookup performance. Further, we discovered that under point lookups, **RX** can compete with traditional comparison-based indexes. In high-miss and high-skew scenarios, **RX** even outperforms the traditional indexes, including the hashtable. **RX** also works well when lookups are submitted in smaller batches, and when the lookup distribution is skewed. However, we also identified that **RX** is currently not competitive with traditional indexes in terms of build time, memory footprint, and support for updates. Thus, **RX** should be used in a read-only fashion. Finally, we have seen that **RX** improves faster than the baselines over multiple hardware generations. If this trend continues, **RX** might be able to outperform the baselines on future RTX generations.

## ACKNOWLEDGMENTS

This work is supported by NVIDIA's Academic Hardware Grant, which provided us with the A6000 GPU.

## REFERENCES

[1] Md. Maksudul Alam, Srikanth B. Yoginath, and Kalyan S. Perumalla. 2016. Performance of Point and Range Queries for In-memory Databases Using Radix Trees

on GPUs. In *18th IEEE International Conference on High Performance Computing and Communications; 14th IEEE International Conference on Smart City; 2nd IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2016, Sydney, Australia, December 12-14, 2016*, Jinjun Chen and Laurence T. Yang (Eds.). IEEE Computer Society, 1493–1500. https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0212

[2] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. 2009. Real-time parallel hashing on the GPU. *ACM Trans. Graph.* 28, 5 (2009), 154. https://doi.org/10.1145/1618452.1618500

[3] Dan A. Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. 2012. Chapter 4 - Building an Efficient Hash Table on the GPU. In *GPU Computing Gems Jade Edition*, Wen mei W. Hwu (Ed.). Morgan Kaufmann, Boston, 39–53. https://doi.org/10.1016/B978-0-12-385963-1.00004-6

[4] Saman Ashkiani, Martin Farach-Colton, and John D. Owens. 2018. A Dynamic Hash Table for the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018*. IEEE Computer Society, 419–429. https://doi.org/10.1109/IPDPS.2018.00052

[5] Saman Ashkiani, Shengren Li, Martin Farach-Colton, Nina Amenta, and John D. Owens. 2018. GPU LSM: A Dynamic Dictionary Data Structure for the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018*. IEEE Computer Society, 430–440. https://doi.org/10.1109/IPDPS.2018.00053

[6] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martin Farach-Colton, and John D. Owens. 2019. Engineering a high-performance GPU B-Tree. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 145–157. https://doi.org/10.1145/3293883.3295706

[7] Muhammad A. Awad, Serban D. Porumbescu, and John D. Owens. 2022. A GPU Multiversion B-Tree. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT 2022, Chicago, Illinois, October 8-12, 2022*, Andreas Klöckner and José Moreira (Eds.). ACM, 481–493. https://doi.org/10.1145/3559009.3569681

[8] Pascal R. Bähr, Bruno Lang, Peer Ueberholz, Marton Ady, and Roberto Kersevan. 2022. Development of a hardware-accelerated simulation kernel for ultra-high vacuum with Nvidia RTX GPUs. *Int. J. High Perform. Comput. Appl.* 36, 2 (2022), 141–152. https://doi.org/10.1177/10943420211056654

[9] Blyth, Simon. 2020. Meeting the challenge of JUNO simulation with Opticks: GPU optical photon acceleration via NVIDIA OptiX. *EPJ Web Conf.* 245 (2020), 11003. https://doi.org/10.1051/epjconf/202024511003

[10] Nils Boeschen and Carsten Binnig. 2022. GaccO - A GPU-accelerated OLTP DBMS. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1003–1016. https://doi.org/10.1145/3514221.3517876

[11] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.* 12, 5 (2019), 544–556. https://doi.org/10.14778/3303753.3303760

[12] NVIDIA Corporation. 2018. NVIDIA Turing Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/ Accessed: February 27, 2023.

[13] NVIDIA Corporation. 2019. Dynamic Updates. https://raytracing-docs.nvidia.com/optix7/guide/index.html#acceleration_structures#dynamic-updates Accessed: February 27, 2023.

[14] NVIDIA Corporation. 2022. CUB. https://nvlabs.github.io/cub/ Accessed on February 27th, 2023.

[15] NVIDIA Corporation. 2023. NVIDIA OptiX. https://developer.nvidia.com/rtx/ray-tracing/optix Accessed: February 27, 2023.

[16] Lauro B. Costa, Samer Al-Kiswany, and Matei Ripeanu. 2009. GPU support for batch oriented workloads. In *28th International Performance Computing and Communications Conference, IPCCC 2009, 14-16 December 2009, Phoenix, Arizona, USA*. IEEE Computer Society, 231–238. https://doi.org/10.1109/PCCC.2009.5403809

[17] Harish Doraiswamy and Juliana Freire. 2022. SPADE: GPU-Powered Spatial Database Engine for Commodity Hardware. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2669–2681. https://doi.org/10.1109/ICDE53745.2022.00245

[18] I. Evangelou, G. Papaioannou, K. Vardis, and A. A. Vasilakis. 2021. Fast Radius Search Exploiting Ray Tracing Frameworks. *Journal of Computer Graphics Techniques (JCGT)* 10, 1 (5 February 2021), 25–48. http://jcgt.org/published/0010/01/02/

[19] Hao Gao and Nikolai Sakharnykh. 2021. Scaling Joins to a Thousand GPUs. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2021, Copenhagen, Denmark, August 16, 2021*, Rajesh Bordawekar and Tirthankar Lahiri (Eds.). 55–64. http://www.adms-conf.org/2021-camera-ready/gao_adms21.pdf

[20] Afton Geil, Martin Farach-Colton, and John D. Owens. 2018. Quotient Filters: Approximate Membership Queries on the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018*. IEEE Computer Society, 451–462. https://doi.org/10.1109/IPDPS.2018.00055

[21] Khronos Group. 2020. Ray Tracing in Vulkan. https://www.khronos.org/blog/ray-tracing-in-vulkan Accessed: February 27, 2023.

[22] Owens Research Group. 2021. MVGpuBTree: Multi-Value GPU B-Tree. https://github.com/owensgroup/MVGpuBTree Accessed: February 27, 2023.

[23] Masatoshi Hayashikawa, Koji Nakano, Yasuaki Ito, and Ryota Yasudo. 2019. Folded Bloom Filter for High Bandwidth Memory, with GPU Implementations. In *2019 Seventh International Symposium on Computing and Networking, CANDAR 2019, Nagasaki, Japan, November 25-28, 2019*. IEEE, 18–27. https://doi.org/10.1109/CANDAR.2019.00011

[24] Daniel Jünger, Christian Hundt, and Bertil Schmidt. 2018. WarpDrive: Massively Parallel Hashing on Multi-GPU Nodes. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018*. IEEE Computer Society, 441–450. https://doi.org/10.1109/IPDPS.2018.00054

[25] Daniel Jünger, Robin Kobus, André Müller, Christian Hundt, Kai Xu, Weiguo Liu, and Bertil Schmidt. 2020. WarpCore: A Library for fast Hash Tables on GPUs. In *27th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2020, Pune, India, December 16-19, 2020*. IEEE, 11–20. https://doi.org/10.1109/HiPC50609.2020.00015

[26] Daniel Jünger. 2022. warpcore. https://github.com/sleeepyjack/warpcore Accessed: February 27, 2023.

[27] Tomas Karnagel, René Müller, and Guy M. Lohman. 2015. Optimizing GPU-accelerated Group-By and Aggregation. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015, Kohala Coast, Hawaii, USA, August 31, 2015*, Rajesh Bordawekar, Tirthankar Lahiri, Bugra Gedik, and Christian A. Lang (Eds.). 13–24. http://www.adms-conf.org/2015/gpu-optimizer-camera-ready.pdf

[28] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, 339–350. https://doi.org/10.1145/1807167.1807206

[29] Moritz Laass. 2021. Point in Polygon Tests Using Hardware Accelerated Ray Tracing. In *SIGSPATIAL '21: 29th International Conference on Advances in Geographic Information Systems, Virtual Event / Beijing, China, November 2-5, 2021*, Xiaofeng Meng, Fusheng Wang, Chang-Tien Lu, Yan Huang, Shashi Shekhar, and Xing Xie (Eds.). ACM, 666–667. https://doi.org/10.1145/3474717.3486796

[30] Zhuohang Lai, Xibo Sun, Qiong Luo, and Xiaolong Xie. 2022. Accelerating multi-way joins on the GPU. *VLDB J.* 31, 3 (2022), 529–553. https://doi.org/10.1007/s00778-021-00708-y

[31] Charles L. Lawson and Richard J. Hanson. 1995. Solving least squares problems. *Classics in applied mathematics 15, SIAM* (1995), 1–337.

[32] Yuchen Li, Qiwei Zhu, Zheng Lyu, Zhongdong Huang, and Jianling Sun. 2021. DyCuckoo: Dynamic Hash Tables on GPUs. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 744–755. https://doi.org/10.1109/ICDE51399.2021.00070

[33] Mariela Lopresti, Fabiana Piccoli, and Nora Reyes. 2021. GPU Permutation Index: Good Trade-Off Between Efficiency and Results Quality. In *Computer Science - CACIC 2021 - 27th Argentine Congress, CACIC 2021, Salta, Argentina, October 4-8, 2021, Revised Selected Papers (Communications in Computer and Information Science)*, Patricia Pesado and Gustavo Gil (Eds.), Vol. 1584. Springer, 183–200. https://doi.org/10.1007/978-3-031-05903-2_13

[34] Vasilis Mageirakos, Riccardo Mancini, Srinivas Karthik, Bikash Chandra, and Anastasia Ailamaki. 2022. Efficient GPU-accelerated Join Optimization for Complex Queries. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 3190–3193. https://doi.org/10.1109/ICDE53745.2022.00295

[35] Mogamat Yaaseen Martin, Simon Lucas Winberg, Mohammed Yunus Abdul Gaffar, and David MacLeod. 2022. The Design and Implementation of a Ray-tracing Algorithm for Signal-level Pulsed Radar Simulation Using the NVIDIA® OptiX Engine. *J. Commun.* 17, 9 (2022), 761–768. https://doi.org/10.12720/jcm.17.9.761-768

[36] Nate Morrical, Ingo Wald, Will Usher, and Valerio Pascucci. 2022. Accelerating Unstructured Mesh Point Location With RT Cores. *IEEE Trans. Vis. Comput. Graph.* 28, 8 (2022), 2852–2866. https://doi.org/10.1109/TVCG.2020.3042930

[37] NVIDIA. 2023. Nsight Compute. https://developer.nvidia.com/nsight-compute Accessed: Jul 9, 2023.

[38] NVIDIA. 2023. Nsight Systems. https://developer.nvidia.com/nsight-systems Accessed: Jul 9, 2023.

[39] NVIDIA. 2023. NVIDIA ADA GPU ARCHITECTURE. https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-v2.1.pdf Accessed: July 13, 2023.

[40] NVIDIA. 2023. NVIDIA AMPERE GA102 GPU ARCHITECTURE. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf Accessed: July 13, 2023.

[41] NVIDIA. 2023. Statement from NVIDIA about inspecting the BVH. https://forums.developer.nvidia.com/t/visualize-optix-generated-bvh-in-nsight-compute/253837 Accessed: May 24, 2023.

[42] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David P. Luebke, David K. McAllister, Morgan McGuire, R. Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: a general purpose ray tracing engine. *ACM Trans. Graph.* 29, 4 (2010), 66:1–66:13. https://doi.org/10.1145/1778765.1778803

[43] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1413–1425. https://doi.org/10.1145/3448016.3457254

[44] Sushil K. Prasad, Michael McDermott, Xi He, and Satish Puri. 2015. GPU-based Parallel R-tree Construction and Querying. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPS 2015, Hyderabad, India, May 25-29, 2015.* IEEE Computer Society, 618–627. https://doi.org/10.1109/IPDPSW.2015.127

[45] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient Join Algorithms For Large Database Tables in a Multi-GPU Environment. *Proc. VLDB Endow.* 14, 4 (2020), 708–720. https://doi.org/10.14778/3436905.3436927

[46] Microsoft DirectX Team. 2018. Announcing Microsoft DirectX Raytracing! https://devblogs.microsoft.com/directx/announcing-microsoft-directx-raytracing/ Accessed: February 27, 2023.

[47] Peter Thoman, Markus Wippler, Robert Hranitzky, and Thomas Fahringer. 2020. RTX-RSim: Accelerated Vulkan Room Response Simulation for Time-of-Flight Imaging. In *Proceedings of the International Workshop on OpenCL* (Munich, Germany) *(IWOCL '20)*. Association for Computing Machinery, New York, NY, USA, Article 17, 11 pages. https://doi.org/10.1145/3388333.3388662

[48] Diego G. Tomé, Tim Gubner, Mark Raasveldt, Eyal Rozenberg, and Peter A. Boncz. 2018. Optimizing Group-By and Aggregation using GPU-CPU Co-Processing. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2018, Rio de Janeiro, Brazil, August 27, 2018*, Rajesh Bordawekar and Tirthankar Lahiri (Eds.). 1–10. http://www.adms-conf.org/2018-camera-ready/tome_groupby.pdf

[49] Brandon Tran, Brennan Schaffner, Joseph M. Myre, Jason Sawin, and David Chiu. 2021. Exploring Means to Enhance the Efficiency of GPU Bitmap Index Query Processing. *Data Sci. Eng.* 6, 2 (2021), 209–228. https://doi.org/10.1007/s41019-020-00148-8

[50] Ingo Wald, Will Usher, Nathan Morrical, Laura Lediaev, and Valerio Pascucci. 2019. RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location. In *High-Performance Graphics 2019 - Short Papers, Strasbourg, France, July 8-10, 2019*, Markus Steinberger and Theresa Foley (Eds.). Eurographics Association, 7–13. https://doi.org/10.2312/hpg.20191189

[51] Bin Wang, Ingo Wald, Nate Morrical, Will Usher, Lin Mu, Karsten E. Thompson, and Richard Hughes. 2022. An GPU-accelerated particle tracking method for Eulerian-Lagrangian simulations using hardware ray tracing cores. *Comput. Phys. Commun.* 271 (2022), 108221. https://doi.org/10.1016/j.cpc.2021.108221

[52] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. 2014. Concurrent Analytical Query Processing with GPUs. *Proc. VLDB Endow.* 7, 11 (2014), 1011–1022. https://doi.org/10.14778/2732967.2732976

[53] Qiang Wang, Bo Peng, Ziyuan Cao, Xing Huang, and Jingfeng Jiang. 2020. A Real-time Ultrasound Simulator Using Monte-Carlo Path Tracing in Conjunction with Optix Engine. In *2020 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2020, Toronto, ON, Canada, October 11-14, 2020.* IEEE, 3661–3666. https://doi.org/10.1109/SMC42975.2020.9283057

[54] Simin You, Jianting Zhang, and Le Gruenwald. 2013. Parallel spatial query processing on GPUs using R-trees. In *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, BigSpatial@SIGSPATIAL 2013, Nov 4th, 2013, Orlando, FL, USA*, Varun Chandola and Ranga Raju Vatsavai (Eds.). ACM, 23–31. https://doi.org/10.1145/2534921.2534949

[55] Stefan Zellmann, Daniel Seifried, Nate Morrical, Ingo Wald, Will Usher, Jamie A. P. Law-Smith, Stefanie Walch-Gassner, and André Hinkenjann. 2022. Point Containment Queries on Ray-Tracing Cores for AMR Flow Visualization. *Comput. Sci. Eng.* 24, 2 (2022), 40–51. https://doi.org/10.1109/MCSE.2022.3153677

[56] Stefan Zellmann, Martin Weier, and Ingo Wald. 2020. Accelerating Force-Directed Graph Drawing with RT Cores. In *31st IEEE Visualization Conference, IEEE VIS 2020 - Short Papers, Virtual Event, USA, October 25-30, 2020.* IEEE, 96–100. https://doi.org/10.1109/VIS47514.2020.00026

[57] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-KV: A Case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores. *Proc. VLDB Endow.* 8, 11 (2015), 1226–1237. https://doi.org/10.14778/2809974.2809984

[58] Xun Zhong, Yong Zhang, Yu Chen, Chao Li, and Chunxiao Xing. 2022. Learned Index on GPU. In *38th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2022, Kuala Lumpur, Malaysia, May 9, 2022.* IEEE, 117–122. https://doi.org/10.1109/ICDEW55742.2022.00024

[59] Yuhao Zhu. 2022. RTNN: accelerating neighbor search using hardware ray tracing. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, Jaejin Lee, Kunal Agrawal, and Michael F. Spear (Eds.). ACM, 76–89. https://doi.org/10.1145/3503221.3508409