Preprint of the version accepted at IEEE ICDE 2025

More Bang For Your Buck(et): Fast and Space-efficient Hardware-accelerated Coarse-granular Indexing on GPUs

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

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

Rosina Kharal
University of Waterloo
Waterloo, Canada
rosina.kharal@uwaterloo.ca

Trevor Brown
University of Waterloo
Waterloo, Canada
trevor.brown@uwaterloo.ca

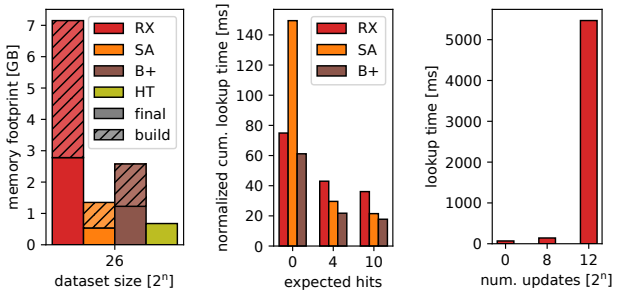
***Abstract*—In recent work, it has been shown that NVIDIA's raytracing cores on RTX video cards can be exploited to realize hardware-accelerated lookups for GPU-resident database indexes. This is done by materializing all keys as triangles in a 3D scene. Lookups are performed by firing rays into the scene and utilizing the built-in index structure to detect collisions with triangles in a hardware-accelerated fashion. While this approach, called RTIndeX (or RX for short), is indeed promising, it currently suffers from three limitations: (1) significant memory overhead per key, (2) slow range lookups, and (3) poor updateability. In this work, we show that all three problems can be tackled by a single design change: Generalizing RX to become a *coarse-granular* index cgRX, which no longer indexes individual keys, but key buckets. We show that representing buckets in 3D space such that the lookup of a key is performed both correctly and efficiently is highly nontrivial and requires a careful orchestration of positioning triangles and firing rays in a specific sequence. Our experimental evaluation shows that cgRX offers the most bang for the buck(et) by providing a up to $6.9\times$ higher ratio of throughput to memory footprint than comparable baselines (that support range lookups). At the same time, cgRX improves the range-lookup performance over RX by up to $15\times$ and offers practical updatability that is up to $5.6\times$ faster than rebuilding from scratch.**

***Index Terms*—Database indexing, GPUs, raytracing**

I. Introduction

Utilizing hardware accelerators in creative ways to speed up database operations has become increasingly popular over the last few years. A good example for this trend is **RT**Inde**X** [1] (or **RX** for short), which is a hardware-accelerated indexing mechanism that exploits the raytracing cores present on modern NVIDIA RTX video cards. The core idea of **RX** is as follows: To index a set of key-rowID pairs, each key is represented by a corresponding triangle in a 3D scene. Then, each triangle is associated with a corresponding rowID. To perform a lookup, a ray is fired through the area in the 3D scene where the triangle representing the key is expected. If it collides with a triangle, the lookup is a hit, and the corresponding rowID is retrieved.

To perform collision detection between rays and triangles efficiently, the video card (GPU) can create and utilize a special tree-based index structure in hardware, called a *bounding volume hierarchy* (BVH), which indexes all triangles in the 3D scene. As shown in [1], **RX** has two advantages over traditional software-based GPU-resident index structures such as [2]–[4]: (a) The construction and querying of the BVH is *built-in*. Consequently, no complex manual hand-crafting of a highly parallel GPU-resident index structure is required. (b) Both the BVH traversal as well as the ray-intersection tests with the candidate triangles are *hardware-accelerated* by the dedicated raytracing cores, speeding up the lookup process over software-based implementations in several cases [1]. However, unfortunately, the approach also currently faces a set of limitations.



(a) Memory footprint (b) Range lookups (c) Updates

Fig. 1: Limitations of **RX**: Memory overhead, range lookups, and lookup performance after updates [1].

First of all, the memory overhead per key of **RX** is high since a single 8B integer key is represented as a triangle described by nine 4B floats, resulting in 36B per key. Consequently, 78% of the key representation is actually overhead. In the left plot in Figure 1, which has been generated from the results of [1], we can see that the traditional index structures have a significantly smaller memory footprint than **RX**, which is mainly due to having less overhead per key. As memory is scarce on GPUs, this can be a limitation for many applications. Second, range lookups are currently a weakness of **RX**. In the middle plot in Figure 1, we can see that for all tested range sizes, the **B+**-tree outperforms **RX**. The reason for this is that a range lookup in **RX** requires a large number of ray intersection tests with candidate triangles if its selectiv-

ity is low. In contrast, a **B+**-tree simply performs a *single* tree traversal for the lower bound key and then sequentially scans the leaf level. Third, **RX** is very sensitive to updates. Interestingly, the problem is not the cost of performing the updates, but a severe drop in lookup performance *after* the updates have been applied. The right plot in Figure 1 shows this by performing a batch of lookups after applying a varying number of updates: The more updates have been applied, the more the lookup performance deteriorates, up to a slowdown of $78\times$ over no updates. The reason for this is that the BVH update procedure only scales the existing bounding volumes to reflect the updates instead of restructuring the BVH. This can heavily increase the number of intersection tests that must be carried out during lookups.

A. Challenges of Hardware-accel. Coarse-granular Indexing

Interestingly, the aforementioned limitations can be addressed by making a single design change: Instead of creating a *fine-granular index* **RX**, which maps each individual key/triangle to its rowID, we propose to generalize the concept to a *coarse-granular index* **cgRX**, which indexes groups of keys instead, but still retains hardware acceleration. Precisely, each group is represented by a single key/triangle, which then maps to a separately stored bucket of key-rowID pairs. This design change drastically reduces the memory footprint of the structure — for example, for a bucket size of eight, the memory *overhead* decreases from 78% in **RX** to only 36% in **cgRX**, since we only need one additional 36B triangle for every bucket of eight 8B keys. As the number of triangles correlates with the size of the generated BVH, **cgRX** also constructs a significantly smaller BVH than **RX**, reducing the traversal time. Of course, additional cost must be factored in for post-filtering a retrieved bucket for the key(s) of interest. Still, by adjusting the granularity at which keys are grouped, we can balance the cost of traversing the BVH, the cost of searching the bucket, and the overall memory footprint depending on the requirements of the environment.

Note that while the basic principle of coarse-granular indexing is straightforward, mapping it to hardware-accelerated raytracing, which we propose in this work in form of **cgRX**, is highly non-trivial. As we no longer materialize *all* keys in the scene, but only a *single* representative key for each bucket, the central challenge is to ensure that the lookup procedure still detects all hits and misses correctly. As we will see, depending on the situation, this requires firing a sequence of up to five rays from specific positions in specific directions. Since at the same time, the performance overhead of a lookup must remain small, we will apply a set of delicate optimizations to both the triangle arrangement as well as the ray firing procedure.

B. Contributions and Structure of the Paper

In summary, we make the following contributions: **(1)** After recapping **RX** in Section II, we present **cgRX**, our new hardware-accelerated coarse-granular index for NVIDIA RTX GPUs in Section III, which supports 64-bit keys as well as point and range lookups. We first discuss the construction and lookup procedure of a **naive representation**, which speeds up the navigation through the 3D scene by introducing additional marker triangles. **(2)** Based on that, we present an **optimized representation**, which avoids the materialization of additional marker triangles altogether. Instead, we turn a subset of representatives into implicit markers. This is done by (a) moving certain representatives and (b) introducing auxiliary representatives in the scene. The evaluation shows that the optimized representation improves both lookup performance and memory footprint for very sparse key sets. **(3)** We present an extension called **cgRXu** to support efficient batch-wise **updates** (inserts and deletes) in Section IV. Insertions and deletions are handled by organizing buckets as a linked list of physical nodes, which are attached (or detached) on demand. By this, updates to the BVH and hence the extreme deterioration of the lookup performance is avoided. **(4)** As **cgRX** provides a set of **configuration parameters**, we analyze their impact for a variety of key distributions in Section V. Precisely, we analyze the impact of (a) the optimizations, (b) the key mapping into 3D space, as well as (c) the bucket size. We also test the robustness of the choice by evaluating 4560 different indexing scenarios. **(5)** Using the best configuration(s), we perform an extensive **experimental evaluation** against a set of state-of-the-art baselines in Section VI. The evaluation analyzes (a) the throughput to memory footprint ratio for point lookups, (b) the range lookup performance, (c) the impact of batching, (d) the impact of the hit rate, (e) the impact of lookup skew, and (f) the update performance.

II. Background

We start by discussing the working principle of the fine-granular index **RX** [1], where we discuss both its construction and lookup procedure. As both **cgRX** and **RX** use NVIDIA's OptiX computing API [5], [6] to program the raytracing pipeline, the following implementation details will also be relevant for the presentation of **cgRX** in Section III.

A. Construction of *RX*

Given a set of key-rowID pairs to index, the construction happens in two steps: In the first step, the keys are transformed into a set of corresponding triangles in the 3D scene, where for each key k, **RX** creates a single isolated triangle. Practically, this is done by writing the positions of the three corner points of each triangle one after the other into a so-called *vertex buffer*. The position of each triangle in the scene is computed using a *key mapping*. [1] observed that this key mapping cannot be arbitrary, but is limited to 23 bits in each dimension to ensure correct floating-point arithmetic. Consequently, **RX** uses a mapping where the 23 least significant bits of each key k are treated as the x coordinate, the next 23 bits as the y coordinate, and the 18 most significant bits as the z coordinate, denoted as $k \mapsto (k_{22:0}, k_{45:23}, k_{63:46})$. Geometrically speaking, this key mapping arranges all triangles into *rows* and *planes*. Note that to ease visualization, in the following examples, we will use a simpler key mapping where the three last bits of the key determine the x coordinate, the next two bits

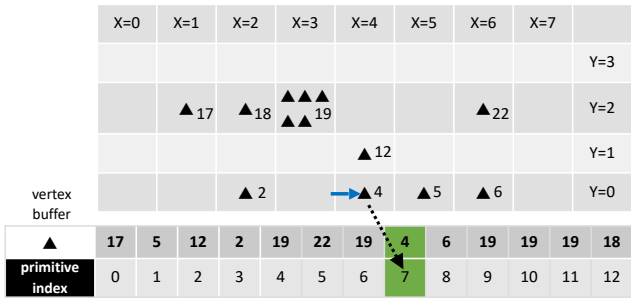


Fig. 2: Example key set and associated triangle representation for **RX** [1], followed by a lookup of key 4 returning rowID 7.

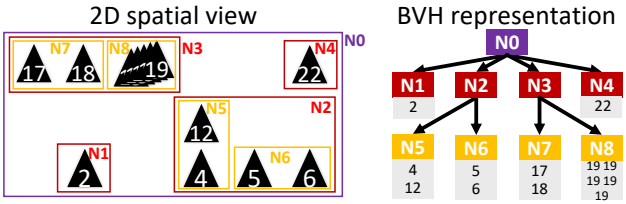


Fig. 3: Possible BVH for the scene in Figure 2

determine the y coordinate, and the remainder determines the z coordinate, i.e., $k \mapsto (k_{2:0}, k_{4:3}, k_{63:5})$. Apart from mapping the keys to triangles, each key k also must be associated with its rowID r. This is done implicitly by materializing the triangle of k at position r in the vertex buffer. This position is called the *primitive index*, and it can be queried later on. Figure 2 shows how the generated triangles look for an example key set of 13 keys, as well as the state of the vertex buffer. For example, key 4 is mapped to the triangle $\blacktriangle_4$ at position $x = 4$, $y = 0$, and $z = 0$. This triangle is materialized in the vertex buffer at slot 7. This effectively associates key 4 with rowID 7.

In the second step, the aforementioned BVH is built on top of the 3D scene. A BVH is a tree-like index structure in which small, disjoint groups of triangles form the leaves. Each group is then enclosed by a 3D cuboid, a so-called bounding volume. These bounding volumes are then iteratively grouped and enclosed by larger bounding volumes until only the root bounding volume remains. To construct the BVH, the vertex buffer is passed to `optixAccelBuild()`, which then indexes all individual triangles in the buffer without any further involvement of the programmer. Figure 3 visualizes how the generated BVH could look like (in 2D) for the example key set. In the example, the triangle representing key 4 is enclosed by the small bounding volume N5 first, next by the larger volume N2, and finally by the root volume N0.

B. Lookups in *RX*

Using the generated BVH, NVIDIA's OptiX is able to quickly find intersections between triangles and rays via hardware acceleration. Therefore, **RX** maps each lookup operation to a corresponding ray-triangle intersection problem. A ray is defined by its point of origin o and a three-dimensional direction vector d. To perform a point lookup of key k, one

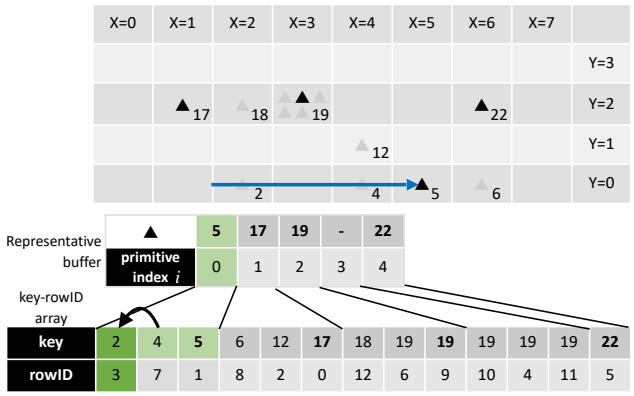


Fig. 4: Example key set and associated triangle representation, followed by a lookup of key 2 which returns rowID 3. The representative $\blacktriangle_5$ of bucket 0 is located in the *same* row as the searched key 2. Row markers are hidden for simplicity.

first computes the 3D position p associated with k using the key mapping, and then selects the ray parameters o and d so that the ray passes through p. If a triangle exists at position p, the ray will intersect this triangle and retrieve the associated rowID. To prevent a ray from extending beyond a single triangle and producing false positives, OptiX provides an option to limit a ray to a specified length. Similarly, a range lookup $[l, u]$ can be performed by firing one or multiple rays in parallel to the x-axis, starting at the position associated with the lower bound l, and limiting the ray to not extend beyond the upper bound u. This way, only the triangles that are located between the given bounds will be hit, and we eventually obtain all rowIDs that are relevant to the lookup. As is typical for GPU-resident indexes, **RX** implements batch lookups to improve GPU utilization, where each lookup is performed by a single thread.

Note that the execution time of a ray traversal depends on the shape of the BVH, and ultimately on the key distribution. Average case analyses for ray traversal times following several common BVH construction algorithms are surveyed in [7].

III. Coarse-granular RX

Let us now discuss the specifics of **cgRX**. Using a running example, we will present two different 3D scene representations that **cgRX** can generate and query: A **naive representation**, and an **optimized representation** which requires fewer rays to be cast in certain situations while *also* having a lower memory footprint.

Just like **RX**, **cgRX** uses a key mapping to uniquely represent each key as a triangle on an integer grid, which is exemplified in Figure 4. However, in contrast to **RX**, not *all* keys are actually materialized as triangles. An array of keys and rowIDs is sorted and logically partitioned into equally-sized buckets (of size 3), and only the last key in each bucket is inserted as a *bucket representative*, shown as a black triangle $\blacktriangle_k$. All remaining keys are *not* materialized in the scene. We still visualize them as gray triangles $\blacktriangle_k$. As a

consequence of this design, the number of triangles we need to store in the vertex buffer is greatly reduced over **RX** which, in turn, reduces the size of the BVH. At the same time, looking up a key k becomes more complex, since there is no guarantee that there will be a triangle at the position p associated with k. Instead, we need to search for the next bucket representative, which is always larger than or equal to k. So, the bucket representative either has to (1) be in the same row, but have a larger or equal x coordinate than p, or (2) be on the same plane, but have a larger y coordinate than p, or (3) be on a different plane and have a larger z coordinate than p.

Case (1) is visualized in Figure 4 by showing the lookup of key 2, where the triangle $\blacktriangle_2$ does not actually exist in the scene: To locate the bucket representative, we cast a single ray along the positive x-axis, starting our ray slightly left of $\blacktriangle_2$. The ray intersects $\blacktriangle_5$, which is the first triangle in the vertex buffer, and is therefore associated with primitive index 0 and hence bucket 0. We then search bucket 0 to find that key 2 occurs at rowID 3 in the original table (column).

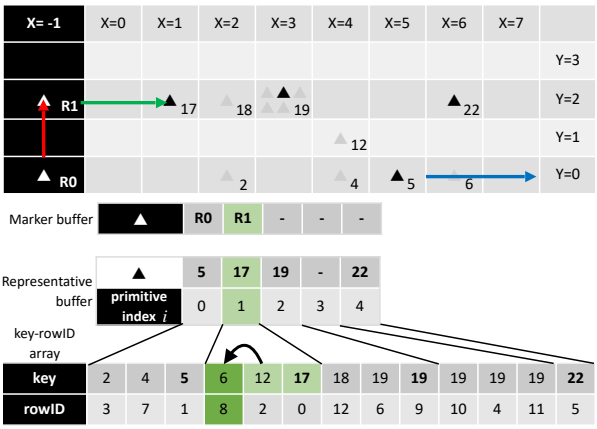


Fig. 5: Lookup of key 6 which returns rowID 8. The representative $\blacktriangle_{17}$ of the corresponding bucket 1 is located in a *different* row as the searched key 6. This row is quickly identified using marker $\triangle_{R1}$.

Case (2), depicted in Figure 5, is more complex: When attempting to find key 6, the first ray fails to hit a representative in the same row. Thus, the next representative must be the leftmost triangle in the next populated row. To allow efficient discovery of the next populated row, we introduce *row markers* $\triangle_R$ into the scene: If a row contains at least one representative, we add a triangle at $x = -1$ in the same row. With the help of row markers, we can easily locate the next populated row by casting a second ray along the y-axis starting in the subsequent row at $x = -1$.

After intersecting $\triangle_{R1}$ at $y = 2$, we cast a third ray from $y = 2, x = 0$ to find the first representative of this row, which hits triangle $\blacktriangle_{17}$ associated with bucket 1. Note that in the figure, the $y = 2$ row contains multiple triangles that could be intersected by the ray. However, we are only interested in the leftmost intersection, i.e., the one closest to the ray origin. This *closest-hit* discovery is a fundamental operation in computer graphics, and therefore, natively supported by OptiX.

Case (3) is similar to case (2). It is shown in Figure 6, where a lookup of key 22 is performed on an extended key set. If the second ray fails to hit a row marker, there are no more populated rows on this plane. Therefore, we now need to find the next populated plane. Similar to what we did with rows, we also mark every populated plane with a *plane marker* $\triangle_P$ at $x = -1$ and $y = -1$. This allows us to cast a ray along the z-axis to discover $\triangle_{P1}$, followed by two more rays: One ray cast along the y-axis locates the next populated row via $\triangle_{R2}$, and the final ray along the x-axis intersects triangle $\blacktriangle_{93}$ associated with bucket 4.

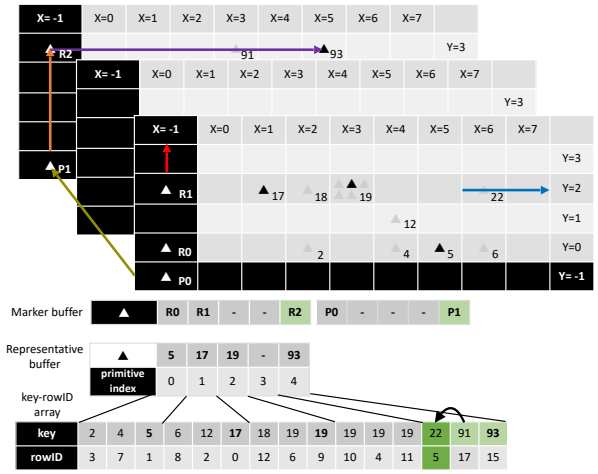


Fig. 6: Lookup of key 22 when the key set is spread across multiple planes. The example shows the worst case where five rays are required to perform the lookup.

Let us also discuss how to **handle duplicates**: If the same key occurs multiple times, it can happen that the duplicates of the key span over multiple buckets. This is the case for key 19 in Figure 6, which occurs five times in total and consequently spans over the buckets 2 and 3. To handle this situation, we create a representative only for the first of the two buckets. A lookup for 19 will then find this first representative and consequently jump to the start of bucket 2 in the sorted key-rowID array. Scanning the bucket retrieves the first duplicate of 19. Scanning further will identify all remaining duplicates of 19 across buckets. The scan stops as soon as the first key larger than 19 is found, namely 22. This ensures that *all* duplicates are visited. Similarly, we only generate a marker for the first representative in the row/plane to avoid duplicate markers.

A. Implementation Details

Construction. In Algorithm 1, we formalize the construction procedure as pseudo-code. We use the notation $k.x$ to refer to the bits of k that are mapped to the x-coordinate. These bit operations are required several times: In lines 2 and 3, we check if all representatives lie on the same plane or even in

the same row, in which case we can skip the allocation and generation of plane/row markers. The algorithm then loops over all buckets and creates a representative for each bucket, if necessary. In lines 13 and 15, we check whether the previous key belongs to a different row/plane, in which case the current key is the first of the current row/plane. The mkTri(x, y, z) function creates a small triangle that is centered around the point (x, y, z). Note that for simplicity, the pseudo-code does not show the specific handling of the first iteration, where there exists no previous representative. Of course, our actual implementation handles this case correctly.

Algorithm 1: Construction of the naive representation

```
   Input: keys, bucketSize
   Output: reps, markers
1  minRep ← keys[bucketSize - 1], maxRep ← keys[len(keys) - 1]
2  multiLine ← minRep.yz != maxRep.yz
3  multiPlane ← minRep.z != maxRep.z
4  numBuckets ← ceil(len(keys) / bucketSize)
5  allocate reps[numBuckets]
6  allocate markers[(multiLine + multiPlane) · numBuckets]
7  for bucketID ← 0 to numBuckets - 1 do in parallel
8  |  repIdx ← min((bucketID + 1) · bucketSize, len(keys)) - 1
9  |  rep ← keys[repIdx]
10 |  prevRep ← keys[repIdx - bucketSize]
11 |  if rep != prevRep then
12 |  |  reps[bucketID] = mkTri(rep.x, rep.y, rep.z)
13 |  if multiLine and rep.yz != prevRep.yz then
14 |  |  markers[bucketID] = mkTri(-1, rep.y, rep.z)
15 |  if multiPlane and rep.z != prevRep.z then
16 |  |  markers[bucketID + numBuckets] = mkTri(-1, -1, rep.z)
17 return reps, markers
```

Lookups. Algorithm 2 shows the code responsible for obtaining the bucketID for a given key. The xCast(x, y, z) function internally delegates to OptiX to cast a ray with direction $(1, 0, 0)$ originating at (x, y, z). Its return value stores whether a triangle was intersected by the ray and, if so, exposes the primitive index as well as the coordinates of the intersection point. yCast and zCast are defined analogously. The first y-axis ray and the z-axis ray have to start in the next row/plane, so we offset their origins by 1 in the appropriate directions. Note that Algorithm 2 can also be used to answer range lookups of the form $[l, u]$: We utilize the raytracing approach to find the representative for l, which leads us to the first bucket containing a value larger than or equal to l. From there, we linearly scan the key-rowID array until we hit the first key larger than u. If the lower bound l is larger than the largest key, we can safely report an empty result. **cgRX** always performs this scan by invoking a separate CUDA kernel to spawn a group of 16 threads per lookup, which allows loading (and potentially aggregating) neighboring entries very efficiently.

Markers. Whenever all representatives share the same plane or row, we can skip the generation of some or all markers. We do so by retrieving the smallest representative and the largest representative of the dataset and checking whether they span across multiple rows (line 2) and multiple planes (line 3) by inspecting their coordinates. During construction, we then add row/plane markers only if they are actually required (lines 13-16). The effect of this optimization is shown in Figures 4 and 5, which only contain row markers. Even so, in the worst-case, the use of markers can still inflate the size of the BVH by 3x, which is a problem that we will address in the optimized representation.

Algorithm 2: Point lookup in the naive representation

```
   Input: key
   Output: bucketID or MISS
1  minRep ← keys[bucketSize - 1], maxRep ← keys[len(keys) - 1]
2  if k < minRep then return 0
3  if k > maxRep then return MISS
4  sameRowHit ← xCast(key.x, key.y, key.z)
5  if sameRowHit then return sameRowHit.primitiveIndex
6  nextRowHit ← yCast(-1, key.y + 1, key.z)
7  if nextRowHit then
8  |  sameRowHit ← xCast(0, nextRowHit.y, key.z)
9  |  return sameRowHit.primitiveIndex
10 nextPlaneHit ← zCast(-1, -1, key.z + 1)
11 nextRowHit ← yCast(-1, 0, nextPlaneHit.z)
12 sameRowHit ← xCast(0, nextRowHit.y, nextPlaneHit.z)
13 return sameRowHit.primitiveIndex
```

Bucket Search. To search a bucket, **cgRX** supports both linear search and binary search on buckets in column layout as well as in row layout. However, our experimental evaluation showed that both for small buckets of 4 entries and very large buckets of 65,536 entries, binary search on a row layout performs best, so we use this combination for the remainder of the paper.

B. Optimized Representation

Depending on the key distribution, it is possible that a single bucket spans multiple rows or even multiple planes. This is the reason why we have to potentially fire multiple rays to locate a representative. As firing more rays makes the lookup more expensive, in the following, we propose an optimized representation which addresses this problem while potentially decreasing the memory footprint. The high-level idea is based on two modifications which we are allowed to perform in the scene without harming correctness: (1) Let r be a representative and let k be the next key after r. Observe that we can replace r by another representative r' as long as $r < r' < k$, even if r' is not a key itself. In other words, we can *move* a representative as long as it does not collide with the next key. (2) Let r and r' be two adjacent representatives with $r < r'$. Then we are allowed to *insert* a new representative r'' between them so that $r < r'' < r'$. Since r'' falls into the bucket represented by r', we need to associate r'' with the same bucketID as r'. Using these rules, we are able to ensure that each populated row ends with a representative in the last slot (at x_{max}), either by moving an existing representative there (1), or by inserting a new one (2). Consequently, when starting a lookup in a populated row, we never have to fire more than a single ray as we will always find a representative there, therefore requiring less rays along the y-axis. Analogously, we can place a representative in the last slot of each populated plane to reduce the number of rays along the z-axis. At the same time, these newly inserted representatives

can also serve as row/plane markers since they are always located at $x = x_{max}$ or $y = y_{max}$, as long as we change the respective offsets for yCast and zCast in Algorithm 2.

Algorithm 3: Construction of the optimized representation

```
   Input: keys, bucketSize
   Output: reps
1  minRep ← keys[bucketSize - 1], maxRep ← keys[len(keys) - 1]
2  multiLine ← minRep.yz != maxRep.yz
3  multiPlane ← minRep.z != maxRep.z
4  numB ← ceil(len(keys) / bucketSize)
5  allocate reps[(1 + multiLine + multiPlane) · numB]
6  for bucketID ← 0 to numB - 1 do in parallel
7      repIdx ← min((bucketID + 1) · bucketSize, len(keys)) - 1
8      rep ← keys[repIdx]
9      nextKey ← keys[repIdx + 1]
10     movable ← nextKey.yz != rep.yz
11     prevRep ← keys[repIdx - bucketSize]
12     nextRep ← keys[repIdx + bucketSize]
13     needsRep ← rep != prevRep or (movable and rep.x != x_max)
14     needsRowMark ← !movable and rep.yz != nextRep.yz
15     needsPlaneMark ← rep.y != y_max and rep.z != nextRep.z
16     if needsRep then
17         x ← if movable then x_max else rep.x
18         doFlip ← movable and prevRep.yz != rep.yz
19         reps[bucketID] ← mkTri(x, rep.y, rep.z, doFlip)
20     if multiLine and needsRowMark then
21         reps[bucketID + numB] ← mkTri(x_max, rep.y, rep.z)
22     if multiPlane and needsPlaneMark then
23         reps[bucketID + 2 · numB] ← mkTri(x_max, y_max, rep.z)
24 return reps
```

Construction. Algorithm 3 shows the pseudo-code for constructing this optimized representation. The code copies the single-row/single-plane optimization from the naive variant (lines 2 and 3), but instead of allocating a separate marker buffer, it reserves additional space in the representative buffer (line 5), as the optimized variant does not differentiate between representatives and markers.

For each bucket, we need to check several conditions before placing the triangles. Following the observations listed above, a representative can be moved to the end of the row if the next key is not in the same row (line 10). This creates a special case for handling duplicate representatives: In the naive variant, we skipped insertion of all but the first representative to ensure that no two representatives exist at the same coordinates. In this variant, it is still possible to also insert the last representative of a duplicate group *if* it can be moved away from its initial position to the end of the row (line 13). If a representative is the last in its row and it cannot be moved, we have to explicitly insert a new representative at the end of the row (line 14). For bucket b, this triangle is placed in slot b + numBuckets of the vertex buffer. Similarly, the bucket with the last representative on a plane generates an additional plane marker at $x = x_{max}$, $y = y_{max}$ in slot $b + 2 \cdot$ numBuckets. If this last representative happens to be located in the last row of a plane ($y = y_{max}$), we can skip its generation, since it coincides with the row marker (line 15).

In line 18, we perform a further optimization called *triangle flipping*. It applies whenever a representative can be moved to the end of the row while also being the only representative in this row. In this case, any ray being fired in the corresponding row will *always* hit this representative. Therefore, we do not need to fire this ray at all. To inform the lookup procedure of this fact, we "flip" the triangle by inverting the order in which the corner points are stored in the buffer (the winding order) from clockwise to counter-clockwise. This way, any ray fired along the y-axis will recognize the hit as a *back-side hit* as opposed to a *front-side hit*, and can react accordingly.

Figure 7 shows the optimized representation for our example key set. In comparison to the naive representation in Figure 4, we can see three differences: One representative is newly inserted as a marker ($▲_7$) while another representative is moved ($▲_{23}$ replaces the representative $▲_{22}$ to its left) to serve as another marker. In exchange, no explicit markers are present anymore at $x = -1$. Let us revisit the special

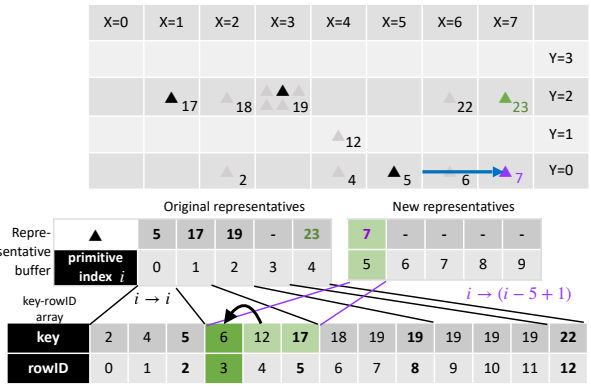


Fig. 7: Visualization of the optimized representation.

cases from Algorithm 3. For the first bucket with bucketID 0, the representative $▲_5$ must be materialized, as it is not a duplicate. However, we cannot materialize it at the end of the row, as its next key $▲_6$ is located in the same row. Still, as $▲_5$ is the last representative of the row, it is responsible for creating a new representative $▲_7$ which serves as the row marker. Key 19 in row $y = 2$ appears multiple times, but since none of its instances are the last key in the row, there is no difference in how the representatives are placed compared to the naive algorithm. Instead, we move representative $▲_{22}$, which happens to be the last key in the row, to the end of the row, producing $▲_{23}$. Finally, we do not insert any plane markers, since all keys reside on a single plane. If we had to generate a plane marker, it would be located in the very last slot of the plane at $x = 7$ and $y = 3$. A lookup of key 6 (hit) is now answered by firing a *single* ray (hitting $▲_7$), instead of *three* rays. Note that the primitive index $i = 5$ of $▲_7$ is greater than the number of buckets, since $▲_7$ is a newly inserted representative that has been stored after all regular representatives in the vertex buffer. Consequently, the primitive index has to be re-mapped to the corresponding bucketID by means of

$$i \mapsto \begin{cases} i - 2 \cdot \text{numBuckets} + 1 & \text{if } i \geq 2 \cdot \text{numBuckets} \\ i - \text{numBuckets} + 1 & \text{if } i \geq \text{numBuckets} \\ i & \text{otherwise} \end{cases}$$

which is cheap and easy to compute on a GPU.

IV. Handling Updates

The presentation thus far has focused on a sorted array-based representation, but inserting into a globally sorted array

requires shifting keys and rebuilding the BVH. Both are prohibitively expensive. So, to facilitate efficient updates, we propose **cgRXu**, a *node-based* variant of this representation.

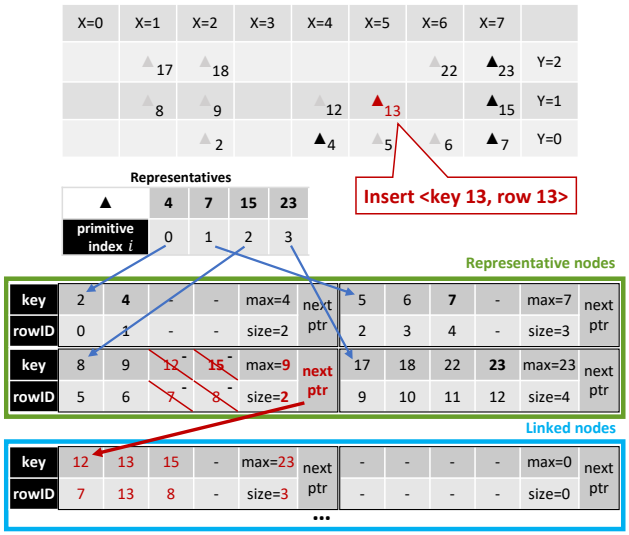


Fig. 8: Node-based representation for updates. Memory is partitioned into a green region where the data structure is initially built using contiguous nodes, and a blue region that is used to extend buckets when nodes are split.

The high level idea is to implement each *bucket* as a *linked list* of *nodes*. Nodes have a fixed size N, a tuneable parameter that we analyze in our experiments. Each node contains sorted *keys* and corresponding *rowID*s, a *next* pointer, a *maxKey* and a current *size*. For each bucket, an initial *representative node* is created, and subsequent insertions into a bucket will cause this node to be *split*, resulting in the creation of a new node, and the movement of half of the keys into that new node. This is illustrated in Figure 8, which depicts a state some time after the initial construction of the index, after some subsequent insertions have happened, and a key-rowID pair $\langle 13, 13 \rangle$ is inserted. Nodes in a bucket can be split multiple times, and all nodes corresponding to a bucket are linked together using their *next* pointers, starting with the representative node. The keys in a bucket are maintained in sorted order across all nodes. This way, a point lookup terminating at a representative node that has been split can simply follow the *next* pointers to locate the correct node with the relevant key, *without requiring the BVH or buckets to be updated*.

Rather than allocating each node individually, it is more efficient to allocate a large slab of memory, and manually partition it into nodes. Once this region has been used entirely, we enlarge it by allocating additional memory. We divide this large allocation into two subregions, one for representative nodes (*representative node region*), and the other for allocating new nodes to be appended to linked lists (*linked node region*). Note that the next pointer of nodes can only be directed into the linked node region, since representative nodes are always the heads of their respective linked lists.

Initial construction. Given a sorted array of key-rowID pairs for initial bulk loading, we first divide them into buckets of size $N/2$ (half the node size—a tunable parameter). Note that this partitions the keys in a *distribution-adaptive* way, by evenly dividing the *keys* being bulk loaded across buckets, rather than evenly dividing the *key range*. In other words, every $N/2$-th key in the input array becomes the *maxKey* of a node. A special *overflow bucket* with maxKey ∞ is added to handle keys larger than any key in the initial bulk load.

We then reserve space for one node in the representative node region for each bucket, and fill the node with its corresponding key-rowID pairs in parallel. Note that each representative triangle effectively points directly to the representative node for its bucket. More precisely, since nodes are stored contiguously in the representative node region, the triangle's primitive index can be multiplied by the size of a node, and added to the base address of the representative node region, to obtain the starting address of the representative node.

Lookups. The raytracing procedure to locate a bucket is unchanged, except for the node address calculation explained above. We then traverse the bucket chain starting at the representative node to find the last node where maxKey is greater than the key we are looking for. If there are many duplicates of a single key, these duplicates may span multiple nodes, and they will all be found by this search procedure.

Insertion and deletion. Similar to lookups, keys to be inserted or deleted are collected in a batch that is then sent to the GPU in an array. The keys are then sorted. Any key that is both to be inserted and deleted in a batch can simply be eliminated from the batch. The actual insertion or deletion is handled by a CUDA kernel that dedicates *one thread to each bucket* in the current representation. The thread responsible for bucket i does two binary searches on the batch's sorted keys to identify the sequence of keys it is responsible for, and traverses the nodes corresponding to bucket i, deleting and inserting these keys as appropriate. An advantage of allocating one thread per bucket is that there are no concurrency issues associated with updating a bucket (for instance, neither atomic read-modify-write instructions nor locks are needed).

Deletions are processed first, as by doing so, space may be created to facilitate insertions without splitting. For each key, the thread first locates the appropriate node in the list by comparing with the *maxKey* of each node. Then, it performs binary search to locate the appropriate index within the appropriate node. Deletion of a key results in keys to the right being shifted to the left. Insertion of a key results in shifting to the right. As explained above, insertion into a full node splits the node, changing its next pointer to point to a new node, and moving half of the keys into the new node. The new node receives the old node's *maxKey*, and the old node's largest key after the split becomes its new *maxKey*. If the new node is being inserted in the middle of a list, its next pointer is set to point to the following node.

V. Parameter Configuration

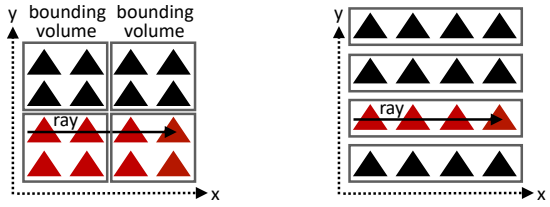
In the following, we will experimentally analyze the impact of all configuration parameters of **cgRX**. Then, we will use the best configuration(s) in Section VI.

We perform all of the following experiments on 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, as consistently verified during the work on this project. The CPU is an AMD ThreadRipper 3990X.

Unless specified otherwise, we generate a key set of 2^{26} keys consisting of 32-bit or 64-bit unsigned integers. For some fixed integer d, the first part of the key set consists of all keys from 0 to $d-1$ to reflect a dense key arrangement, and the second part is picked uniformly and randomly from the remaining value range to reflect a sparse key arrangement. In the experiments, we vary the percentage of keys that are picked uniformly from 0% to 100%, which we simply refer to as the *uniformity* of the key set. We always shuffle the key sequence, and the final position in the shuffled sequence determines a key's rowID. Lookups are drawn randomly from the key set, and we perform 2^{27} lookups by default. The rowIDs obtained through the lookup are aggregated per-lookup, and then written to a separate result buffer to test for correctness.

A. Key Mapping and 3D Representation

First, we compare the naive with the optimized representation of **cgRX** in terms of point-lookup performance to see whether one variant performs consistently better than the other. However, before that, we have to discuss the impact of the key mapping. We identified that for the default key mapping used in [1], namely $k \mapsto (k_{22:0}, k_{45:23}, k_{63:46})$, the performance of both variants was not competitive for sparser key distributions. The reason for this is that the proprietary BVH construction algorithm cannot choose a reasonable bounding volume layout due to the data being largely uniform. This renders the first (and non-avoidable) x-axis ray highly expensive. In Figure 9a, we conceptually visualize such a disadvantageous clustering. In the example, the x-axis ray has to perform costly intersection tests with an unnecessarily large number of triangles since it also has to check for intersections in neighboring rows. Ideally, the bounding volumes would primarily extend along the x-axis, such that only the triangles in the current row have to be checked for intersection, as shown in Figure 9b. To incentivize such a grouping, we simply adjust the key mapping slightly by multiplying both the y-coordinate $k_{45:23}$ and the z-coordinate $k_{63:46}$ with a large, carefully chosen constant, resulting in the mapping $k \mapsto (k_{22:0}, 2^{15} \cdot k_{45:23}, 2^{25} \cdot k_{63:46})$. *Consequently, in all upcoming experiments, we only use the scaled mapping.*



(a) No scaling. (b) Scaling the y-axis.

Fig. 9: The impact of scaling on the BVH structure (shown for 2D). Red triangles must be tested for intersection.

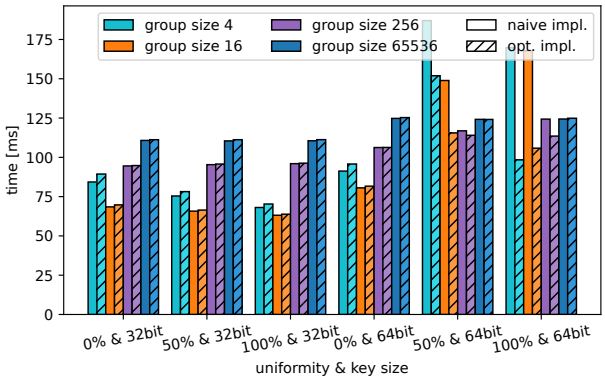


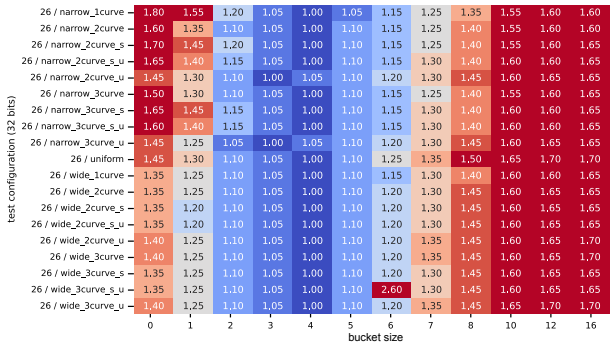
Fig. 10: Naive vs optimized representation for scaled key mapping $k \mapsto (k_{22:0}, 2^{15} \cdot k_{45:23}, 2^{25} \cdot k_{63:46})$.

Coming back to the initial comparison of the naive and optimized representations, we can observe in Figure 10 that for 32-bit key sets, both variants perform equally well. From a geometric perspective, 32-bit keys are always arranged on a single plane. The amount of rays for lookups is therefore limited to three, with most lookups requiring only one ray. Therefore, the optimized scene representation does not yield significant improvements. However, for the 64-bit key sets with a high uniformity (and hence, high sparsity), optimizing the representatives significantly shortens the lookup procedure and improves the performance. Inspecting the number of individually fired rays for each variant revealed that for smaller buckets, the optimized representation avoids firing the second x-axis ray in most cases, because the previous y-axis ray hit a flipped representative. Apart from performance, we can also confirm that the optimized representation reduces the memory footprint over the naive representation for sparse key sets: For example, for 64-bit keys and a bucket size of 4, the optimized representation saves 16% and 28% memory over the naive one for a uniformity of 50% and 100%, respectively. *Hence, we will use only the optimized representation in the following.*

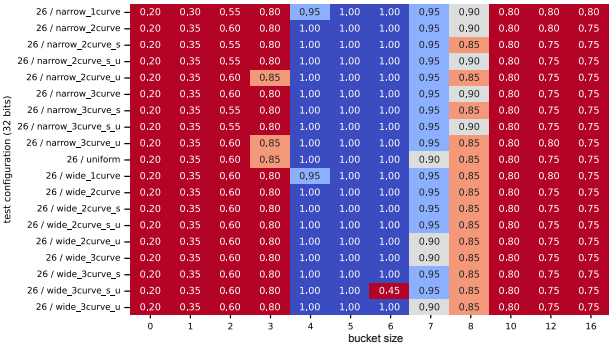
B. Bucket Size

Next, we identify the best bucket size for **cgRX**. As our index aims at providing high space efficiency while achieving competitive lookup performance, intuitively, we want to choose the largest bucket size that still offers a good throughput. To quantify how well our index balances this trade-off, we introduce a metric called *throughput per memory footprint*. We take the throughput as entries looked up per second and divide it by the memory footprint of the structure in bytes.

By this, we essentially measure how an index structure "buys" throughput performance by consuming additional memory (for **cgRX**, this means adding representatives in the scene).



(a) Point-lookup time.



(b) Throughput per memory footprint.

Fig. 11: Robustness of bucket size across 19 key distributions.

To analyze which bucket size performs best, we evaluate twelve bucket size configurations against nineteen different key distributions, varying from uniform to highly skewed and mixtures of both. Further, we test both 32-bit and 64-bit keys, five different key set sizes (2^{24} to 2^{28}), and two GPU generations (RTX 4090 and A6000), resulting in 4560 test combinations in total. The results show that across the board, a universal decision on the best performing bucket size can be made. Due to space restrictions, in Figure 11, we can show only the results for 32-bit keys and a key set size of 2^{26} on the RTX 4090. For each configuration, we show the relative performance in comparison to the best performing bucket size, where Figure 11a shows the pure point lookup-time, while Figure 11b shows the throughput per memory footprint metric. We can see that for the latter, a bucket size of $2^5 = 32$ performs best in the vast majority of cases. Even when inspecting only the lookup performance, a bucket size of 32 is among the best choices. However, we can also see that for a significantly larger bucket size of $2^8 = 256$, the throughput per memory footprint is still 85% of the best option. Consequently, we also consider this configuration further as a space efficient alternative.

Size recommendation. Based on our findings, **cgRX** uses a bucket size of **32** by default for the remaining experiments, which optimizes the throughput per memory footprint ratio. Also, we show the results for a bucket size of **256** as a space efficient and performance-competitive alternative.

VI. Experimental Evaluation

In the following, we evaluate how assorted configurations of **cgRX** perform against a set of competitive baselines. Under the update workload, we evaluate both **cgRX** and **cgRXu**. We include the same baselines as in [1]: **HT**: A GPU-resident open addressing hash table [4], [8], which performs cooperative probing. The target load factor is set to the recommended 80% (40% for updates). **B+**: A GPU-resident B+tree [9], [10] performing cooperative 16-thread tree traversal and only supporting 32-bit keys. **SA**: A GPU-resident Sorted Array [1] which uses binary search for lookups. Note that all methods that require the sorting of the input key-rowID array (**cgRX**, **B+**, and **SA**) use CUB's `DeviceRadixSort` [11] for this purpose, and the **cost for sorting is always included** in the reported times.

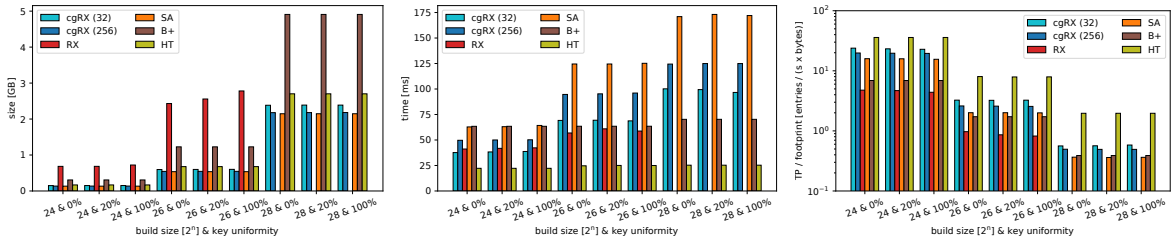
Method	Point	Range	Mem	64-bit	Bulk-load	Updates
HT [4], [8]	✓	×	med	✓	×	✓
B+ [9], [10]	✓	✓	med	×	✓	✓
SA [1]	✓	✓	low	✓	✓	rebuild
RX [1]	✓	✓	high	✓	✓	rebuild
RTScan (RTc1) [12]	×	✓	high	limited	on CPU	rebuild
cgRX	✓	✓	low	✓	✓	rebuild
cgRX-u	✓	✓	low	✓	✓	✓

TABLE I: Overview of all tested indexes.

Of course, we also compare against the original fine-granular **RX**. Additionally, we compare the range-lookup performance against **RTScan (RTc1)** [12], another recently published raytracing based indexing method which has been specifically designed for that purpose. Instead of concurrently executing a large number of (single-threaded) lookups, **RTScan (RTc1)** parallelizes a single range lookup by firing a large number of rays at different positions concurrently, where the number of concurrently fired rays depends on the size of the range. To ensure a fair comparison for the case where a single range lookup does not fully utilize the available resources, we extended the original implementation to support executing a batch of 32 range lookups concurrently. Note that **RTScan (RTc1)** does not support point lookups out of the box and hence, we cannot include it in the corresponding experiments. Table I summarizes the core features of all competitors.

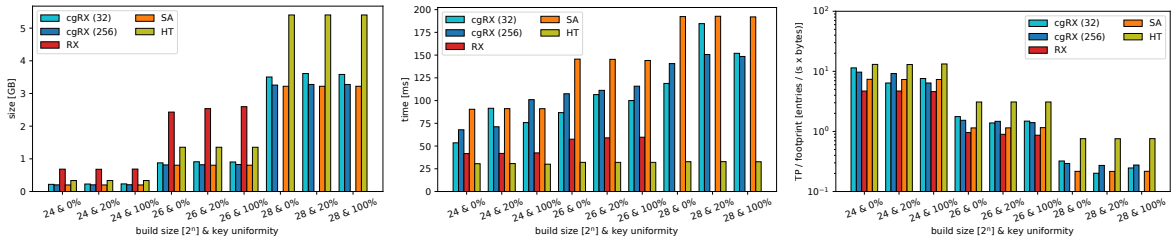
A. Memory Footprint and Point-lookup Performance

One of our central motivations for **cgRX** was to reduce the memory footprint of the original **RX** approach *while* providing good performance. To find out whether **cgRX** achieves this goal, Figure 12a shows the permanent memory footprint of all methods on 32-bit key sets of different sizes (2^{24}, 2^{26}, and 2^{28}) with varying uniformity (0%, 20%, and 100%). For **cgRX**, we evaluate bucket sizes 32 and 256. While for 2^{24} keys, all indexes have a negligible memory footprint, for 2^{26} keys,



(a) Memory footprint. (b) Accumulated point-lookup time. (c) Throughput per memory footprint.

Fig. 12: Comparison of memory footprint and point-lookup performance for key range $[0, 2^{32} - 1]$.



(a) Memory footprint. (b) Accumulated point-lookup time. (c) Throughput per memory footprint.

Fig. 13: Comparison of memory footprint and point-lookup performance for key range $[0, 2^{64} - 1]$.

RX has by far the highest footprint between 2.5GiB (0% uniformity) and 2.9GiB (100% uniformity). In comparison, even for a rather small bucket size of 32, **cgRX** shows a significantly lower memory footprint of only around 0.7GiB, which is already less than the footprint of **B+** at around 1.1GiB. For a bucket size of 256, **cgRX** even approaches the space-optimal **SA**. For 2^{28} keys, **RX** runs out of memory, while **cgRX** still beats **HT** and stays on par with **SA**. **B+** consumes almost twice the amount of memory as the best **cgRX** configuration.

In the Figures 12b and 12c, we bring the point-lookup performance into the picture. While Figure 12b showing the accumulated point-lookup time alone reveals that **cgRX** has a point-lookup performance somewhere between the remaining range-lookup supporting structures, Figure 12c sets both memory footprint and point-lookup performance into perspective by showing the previously introduced throughput per memory footprint metric. We can see that **cgRX** clearly performs best among all indexes supporting range lookups. For 2^{24} keys, the best configuration of **cgRX** has a throughput per memory footprint that is $5\times$ higher than for **RX**, $3.5\times$ higher than for **B+**, and $1.5\times$ higher than for **SA**. At 2^{26} keys, the throughput per memory footprint is still $3.4\times$ higher than for **RX**, $1.9\times$ higher than for **B+**, and $1.6\times$ higher than for **SA**. Only **HT** outperforms **cgRX** by $2.5\times$. For 2^{28} keys, where **RX** runs out of memory, the improvement over **B+** and **SA** is still around $1.5\times$. Note that we observe a similar trend for 64-bit keys in Figure 13 where, unfortunately, we cannot include **B+** as it lacks the support for wide keys. Overall, we can see that **cgRX** provides the best "bang for the buck" of all general-purpose GPU-resident indexes, a property which is especially important in the presence of scarce GPU memory.

B. *Range-lookup Performance*

Next, we analyze the range-lookup performance. We use a 32-bit key set of 2^{26} keys with a uniformity of 0% (dense) and vary the number of expected hits per range lookup from 2^0, which resembles a point lookup, to 2^{24}. We fire a batch of 2^{16} range lookups and report the normalized cumulative lookup time, which is the total time of all range lookups of the batch divided by the total number of retrieved entries, on a logarithmic scale. We also include the range-lookup specific baselines **RTScan (RTc1)** as well as a **FullScan** which just scans the entire array and filters for the range of interest. **HT** does not support range lookups and is therefore not included.

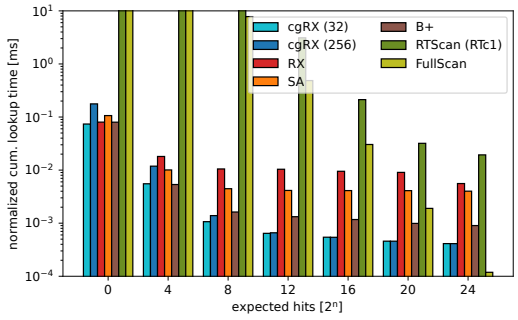


Fig. 14: Range lookups on a dense 32-bit key range.

Figure 14 shows the results (cut off at 10ms). We can see that for range lookups, **cgRX** (32) outperforms the direct

competitor **RX** for all tested bucket sizes. The reason for this is that **cgRX** must perform only one point lookup per range lookup, followed by a simple scan. In contrast, **RX** must detect all qualifying entries in the collision detection of its tracing procedure, which is prohibitively expensive. For selectivities between 2^8 and 2^{20} hits per range lookup, **cgRX** is the best performing method. For example, for 2^8, it is more than $9.9\times$ faster than **RX**, $4.2\times$ faster than **SA**, and $1.5\times$ faster than **B+**. As **cgRX** uses a cooperative scan just like **B+**, the performance of both methods becomes almost the same for lower selectivities. However, **cgRX** has the advantage that all data is stored consecutively, while **B+** scans individual leaf nodes. Surprisingly, **RTScan (RTc1)** is even four orders of magnitude slower than **cgRX**, as it is not able to properly parallelize the batch of lookups. In fact, **RTScan (RTc1)** is even slower than **FullScan**, showing that it is currently not suited for answering batched range lookups, despite batched lookups being a common workload on GPUs.

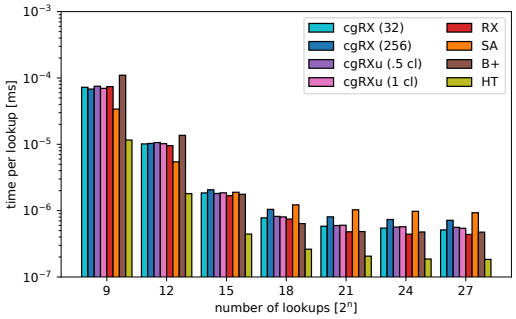


Fig. 15: Varying the number of lookups fired in a batch.

C. Varying the Batch Size

So far, we have used a batch size of 2^{27} point lookups and 2^{16} range lookups. Depending on the application, smaller or larger batches might also arrive, thus we now vary the number of point lookups fired in a batch from 2^9 to 2^{27}. For completeness, we also include **cgRXu** here. Figure 15 reports for each configuration the time per lookup, i.e., the time it took to answer all queries divided by the number of queries, on a logarithmic scale. First of all, we can observe that the performance of all methods deteriorates with a decrease in batch size — for very small batches of 2^9 and 2^{12} point lookups, the GPU becomes severely under-utilized. For batch sizes of 2^{15} to 2^{27}, the performance remains rather stable for all indexes. This shows that **cgRX** is not more susceptible to smaller batch sizes than the traditional baselines — in contrast, for a batch size of 2^{15}, **cgRX** draws even with **RX** and **B+** while having a significantly smaller memory footprint.

D. Varying the Hit Ratio

So far, all point lookups resulted in hits. To see the impact of misses, we now fire a certain amount of point lookups that do *not* hit an indexed key and report the accumulated point-lookup time. We differentiate between misses that lie within the value range of the indexed data, and ones that lie outside of that range. The key set consists of 32-bit keys with uniformity 100%. Figure 16 shows the results. We

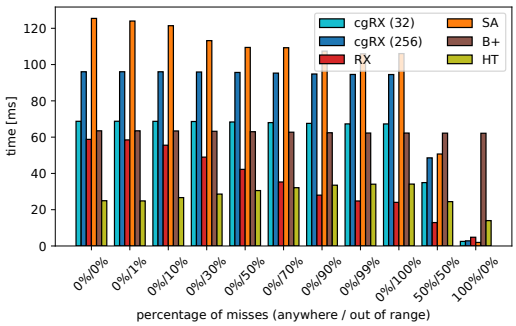


Fig. 16: Varying the hit ratio.

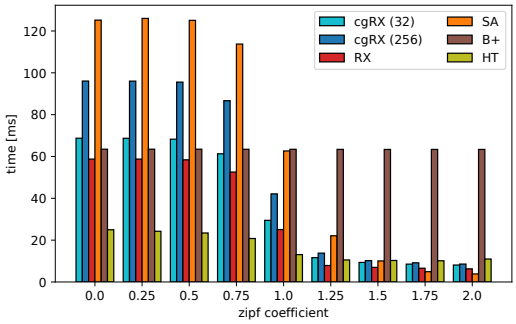


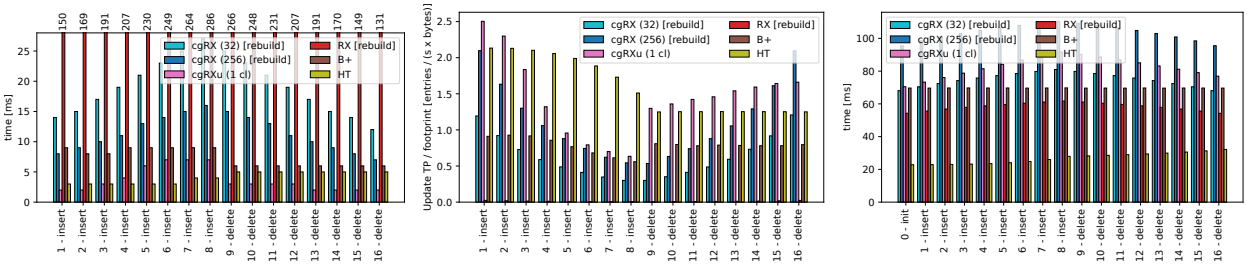
Fig. 17: Varying the skew of lookups.

observe that while **RX** strongly benefits from misses, this is not the case for **cgRX**. The reason for this is that **RX** is able to abort the BVH traversal as soon as it detects that a key is not covered by any bounding volume. This is not possible for **cgRX** which always finds a representative if the target key is within the value range. Consequently, the miss is detected rather late during the bucket search process. This means that **cgRX** should be primarily used in hit-only or hit-mostly lookup scenarios. The last two bars show the effect of out-of-range misses: Here, the search is trivial, and **cgRX** can answer all queries quickly.

E. Varying the Lookup Skew

In all previous experiments, we picked the lookup keys uniformly from the key range. In the following, we test the effect of skewed lookups, which follow a Zipf distribution. We include the uniform distribution seen so far by using a Zipf coefficient of 0.0, and evaluate different levels of skew by varying the coefficient from 0.25 (low skew) to 5.0 (extreme skew). Again, we report the accumulated point-lookup time.

In Figure 17, we can observe that skew is generally beneficial from a performance perspective, as it increases the chance of cache hits and therefore reduces memory accesses. **B+** is an outlier here, where the lookup time is apparently unaffected by skew. NVIDIA's kernel profiler shows that the execution is bottlenecked by the so-called address divergence unit, which handles block synchronization and divergent branches.



(a) Applying batches of updates. (b) Update throughput per mem. footprint. (c) Accumulated point-lookup time.

Fig. 18: Updating **cgRXu**, **B+**, and **HT** compared against rebuilding **cgRX** and **RX** from scratch.

F. Updates

Finally, let us investigate how well our node-based update mechanism in **cgRXu** performs. We compare it against the alternative of rebuilding **cgRX** from scratch for every update batch. Further, we include the option of rebuilding **RX**, which was also the only practical way of applying updates in [1], as well as the update mechanisms of **B+** and **HT**.

Initially, we bulk-load all variants with 2^{26} keys, with 100% uniformity. Then, we fire eight waves of equally-sized insertion batches, where each batch is followed by a lookup batch of size 2^{27}. We configure these waves of insertions such that in total, the number of entries is increased by $2.2\times$. Afterwards, we perform eight corresponding waves of deletions, again interleaved with lookups. We configure **cgRXu** with a node size corresponding to a 128B cache line, initially filled to 50%.

Figure 18a shows the time to apply update waves for all variants, while Figure 18b shows the ratio of update throughput to the structure's current memory footprint. Figure 18c shows the time to perform lookup batches after each update wave has been applied. We can see in 18a that **cgRXu** reduces the cost of applying updates significantly by up to $5.6\times$ in comparison to the impractical full rebuilds in **cgRX**. Compared to other baselines (**B+** and **HT**), **cgRXu** indicates competitive update performance and even outperforms most baselines in many instances. Also, the cost of updating **cgRXu** increases at a slower rate than the cost of fully rebuilding. Rebuilding **RX** is not an option at all, as it is an order of magnitude more expensive than the baselines. At the same time, we can observe in 18b that introducing a linked list of nodes to represent buckets in **cgRXu** results in improved update throughput per memory footprint, even though nodes might only be partially occupied. However, we can also see in 18c that for update-heavy workloads **B+** and **HT** still perform lookups better than **cgRX** and **cgRXu**.

VII. Related Work

Apart from the original **RX** [1] and the recently published **RTScan (RTc1)** [12], which are both baselines in this work, there exists a line of work from other areas that exploits hardware-accelerated raytracing. These include point containment tests [13]–[16], time-of-flight imaging [17]–[19], radius or nearest neighbour search [20]–[22], graph rendering [23] and tracking of particle movement in applied physics [24]–[26]. These applications re-write their original problem into a ray tracing task, then leverage RT-core hardware acceleration for efficient processing. RayJoin [27] is a recent application leveraging RT cores to accelerate spatial join queries. In contrast to traditional relational joins, these spatial joins require fast evaluation of line segment intersections and point-in-polygon tests. Arkade [22] builds on prior research in k-nearest neighbor search using ray tracing, expanding support to include additional distance metrics such as Manhattan and angular (cosine) distances. While these works also exploit hardware-accelerated ray tracing creatively, they unfortunately do not qualify as indexing baselines for this work. In terms of indexes, many CPU data structures have been adapted to become GPU-resident in recent years. GPU-resident indexes include hash tables [8], [28]–[33], from which we picked our baseline **HT**, but also bloom filters [8], [34], [35] and quotient filters [36], which are suitable for set containment tests and trade memory footprint with false-positive accuracy. Further, radix trees [37] and comparison-based trees [9], [38]–[40] also exist for GPUs and also provide range lookup support. While our evaluation includes a state-of-the-art comparison-based tree **B+**, unfortunately, no public implementation of the radix trees is available at the time of writing. There also exist GPU-resident spatial indexes such as R-Trees [41], [42], GPU permutation indexes [43], and a GPU-resident learned index [44]. While these would also make great baselines for our comparisons, their codebase is not publicly accessible.

VIII. Conclusion

We presented **cgRX**, a coarse-granular GPU-resident index which exploits hardware acceleration via RT cores and overcomes the main limitations of its fine-granular predecessor **RX**, namely high memory footprint, poor range-lookup performance, and bad updateability. We have shown that **cgRX** provides the most bang for the buck by offering an up to $5\times$ higher throughput in relation to the memory footprint compared to other state-of-the-art GPU-resident indexes that support both point and range lookups. At the same time, **cgRX** improves the range-lookup performance more than $15\times$ over **RX**. Our updatable variant **cgRXu** improves the update performance by up to $5.6\times$ over rebuilding from scratch.

REFERENCES

[1] J. Henneberg and F. Schuhknecht, "RTIndeX: Exploiting hardware-accelerated GPU raytracing for database indexing," *Proc. VLDB Endow.*, vol. 16, no. 13, pp. 4268–4281, 2023. [Online]. Available: https://www.vldb.org/pvldb/vol16/p4268-schuhknecht.pdf

[2] M. A. Awad, S. Ashkiani, R. Johnson, M. Farach-Colton, and J. D. Owens, "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*, J. K. Hollingsworth and I. Keidar, Eds. ACM, 2019, pp. 145–157. [Online]. Available: https://doi.org/10.1145/3293883.3295706

[3] M. A. Awad, S. D. Porumbescu, and J. D. Owens, "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*, A. Klöckner and J. Moreira, Eds. ACM, 2022, pp. 481–493. [Online]. Available: https://doi.org/10.1145/3559009.3569681

[4] D. Jünger, R. Kobus, A. Müller, C. Hundt, K. Xu, W. Liu, and B. Schmidt, "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, 2020, pp. 11–20. [Online]. Available: https://doi.org/10.1109/HiPC50609.2020.00015

[5] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. P. Luebke, D. K. McAllister, M. McGuire, R. K. Morley, A. Robison, and M. Stich, "Optix: a general purpose ray tracing engine," *ACM Trans. Graph.*, vol. 29, no. 4, pp. 66:1–66:13, 2010. [Online]. Available: https://doi.org/10.1145/1778765.1778803

[6] N. Corporation, "Nvidia optix," 2023, accessed: February 27, 2023. [Online]. Available: https://developer.nvidia.com/rtx/ray-tracing/optix

[7] D. Meister and J. Bittner, "Performance comparison of bounding volume hierarchies for GPU ray tracing," vol. 11, no. 3, 2022. [Online]. Available: https://jcgt.org/published/0011/04/01/paper.pdf

[8] D. Jünger, R. Kobus, A. Müller, C. Hundt, K. Xu, W. Liu, and B. Schmidt, "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, 2020, pp. 11–20. [Online]. Available: https://doi.org/10.1109/HiPC50609.2020.00015

[9] M. A. Awad, S. D. Porumbescu, and J. D. Owens, "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*, A. Klöckner and J. Moreira, Eds. ACM, 2022, pp. 481–493. [Online]. Available: https://doi.org/10.1145/3559009.3569681

[10] O. R. Group, "Mvgpubtree: Multi-value gpu b-tree," 2021, accessed: February 27, 2023. [Online]. Available: https://github.com/owensgroup/MVGpuBTree

[11] N. Corporation, "Cub," 2022, accessed on February 27th, 2023. [Online]. Available: https://nvlabs.github.io/cub/

[12] Y. Lv, K. Zhang, Z. Wang, X. Zhang, R. Lee, Z. He, Y. Jing, and X. S. Wang, "Rtscan: Efficient scan with ray tracing cores," *Proc. VLDB Endow.*, vol. 17, no. 6, pp. 1460–1472, 2024. [Online]. Available: https://www.vldb.org/pvldb/vol17/p1460-lv.pdf

[13] S. Zellmann, D. Seifried, N. Morrical, I. Wald, W. Usher, J. A. P. Law-Smith, S. Walch-Gassner, and A. Hinkenjann, "Point containment queries on ray-tracing cores for AMR flow visualization," *Comput. Sci. Eng.*, vol. 24, no. 2, pp. 40–51, 2022. [Online]. Available: https://doi.org/10.1109/MCSE.2022.3153677

[14] N. Morrical, I. Wald, W. Usher, and V. Pascucci, "Accelerating unstructured mesh point location with RT cores," *IEEE Trans. Vis. Comput. Graph.*, vol. 28, no. 8, pp. 2852–2866, 2022. [Online]. Available: https://doi.org/10.1109/TVCG.2020.3042930

[15] I. Wald, W. Usher, N. Morrical, L. Lediaev, and V. Pascucci, "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*, M. Steinberger and T. Foley, Eds. Eurographics Association, 2019, pp. 7–13. [Online]. Available: https://doi.org/10.2312/hpg.20191189

[16] M. Laass, "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*, X. Meng, F. Wang, C. Lu, Y. Huang, S. Shekhar, and X. Xie, Eds. ACM, 2021, pp. 666–667. [Online]. Available: https://doi.org/10.1145/3474717.3486796

[17] Q. Wang, B. Peng, Z. Cao, X. Huang, and J. Jiang, "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, 2020, pp. 3661–3666. [Online]. Available: https://doi.org/10.1109/SMC42975.2020.9283057

[18] M. Y. Martin, S. L. Winberg, M. Y. A. Gaffar, and D. MacLeod, "The design and implementation of a ray-tracing algorithm for signal-level pulsed radar simulation using the nvidia® optix engine," *J. Commun.*, vol. 17, no. 9, pp. 761–768, 2022. [Online]. Available: https://doi.org/10.12720/jcm.17.9.761-768

[19] P. Thoman, M. Wippler, R. Hranitzky, and T. Fahringer, "Rtx-rsim: Accelerated vulkan room response simulation for time-of-flight imaging," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3388333.3388662

[20] I. Evangelou, G. Papaioannou, K. Vardis, and A. A. Vasilakis, "Fast radius search exploiting ray tracing frameworks," *Journal of Computer Graphics Techniques (JCGT)*, vol. 10, no. 1, pp. 25–48, February 2021. [Online]. Available: http://jcgt.org/published/0010/01/02/

[21] Y. Zhu, "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*, J. Lee, K. Agrawal, and M. F. Spear, Eds. ACM, 2022, pp. 76–89. [Online]. Available: https://doi.org/10.1145/3503221.3508409

[22] D. K. Mandarapu, V. Nagarajan, A. Pelenitsyn, and M. Kulkarni, "Arkade: k-nearest neighbor search with non-euclidean distances using gpu ray tracing," in *Proceedings of the 38th ACM International Conference on Supercomputing*, 2024, pp. 14–25.

[23] S. Zellmann, M. Weier, and I. Wald, "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, 2020, pp. 96–100. [Online]. Available: https://doi.org/10.1109/VIS47514.2020.00026

[24] Blyth, Simon, "Meeting the challenge of juno simulation with opticks: Gpu optical photon acceleration via nvidia optix," *EPJ Web Conf.*, vol. 245, p. 11003, 2020. [Online]. Available: https://doi.org/10.1051/epjconf/202024511003

[25] B. Wang, I. Wald, N. Morrical, W. Usher, L. Mu, K. E. Thompson, and R. Hughes, "An gpu-accelerated particle tracking method for eulerian-lagrangian simulations using hardware ray tracing cores," *Comput. Phys. Commun.*, vol. 271, p. 108221, 2022. [Online]. Available: https://doi.org/10.1016/j.cpc.2021.108221

[26] P. R. Bähr, B. Lang, P. Ueberholz, M. Ady, and R. Kersevan, "Development of a hardware-accelerated simulation kernel for ultra-high vacuum with nvidia RTX gpus," *Int. J. High Perform. Comput. Appl.*, vol. 36, no. 2, pp. 141–152, 2022. [Online]. Available: https://doi.org/10.1177/10943420211056654

[27] L. Geng, R. Lee, and X. Zhang, "Rayjoin: Fast and precise spatial join," in *Proceedings of the 38th ACM International Conference on Supercomputing*, 2024, pp. 124–136.

[28] D. Jünger, C. Hundt, and B. Schmidt, "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, 2018, pp. 441–450. [Online]. Available: https://doi.org/10.1109/IPDPS.2018.00054

[29] S. Ashkiani, M. Farach-Colton, and J. D. Owens, "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, 2018, pp. 419–429. [Online]. Available: https://doi.org/10.1109/IPDPS.2018.00052

[30] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, "Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1226–1237, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1226-zhang.pdf

[31] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-time parallel hashing on the GPU," *ACM Trans. Graph.*, vol. 28, no. 5, p. 154, 2009. [Online]. Available: https://doi.org/10.1145/1618452.1618500

[32] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Chapter 4 - building an efficient hash

table on the gpu," in *GPU Computing Gems Jade Edition*, ser. Applications of GPU Computing Series, W. mei W. Hwu, Ed. Boston: Morgan Kaufmann, 2012, pp. 39–53. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123859631000046

[33] Y. Li, Q. Zhu, Z. Lyu, Z. Huang, and J. Sun, "Dycuckoo: Dynamic hash tables on gpus," in *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2021, pp. 744–755. [Online]. Available: https://doi.org/10.1109/ICDE51399.2021.00070

[34] L. B. Costa, S. Al-Kiswany, and M. Ripeanu, "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, 2009, pp. 231–238. [Online]. Available: https://doi.org/10.1109/PCCC.2009.5403809

[35] M. Hayashikawa, K. Nakano, Y. Ito, and R. Yasudo, "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, 2019, pp. 18–27. [Online]. Available: https://doi.org/10.1109/CANDAR.2019.00011

[36] A. Geil, M. Farach-Colton, and J. D. Owens, "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, 2018, pp. 451–462. [Online]. Available: https://doi.org/10.1109/IPDPS.2018.00055

[37] M. M. Alam, S. B. Yoginath, and K. S. Perumalla, "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*, J. Chen and L. T. Yang, Eds. IEEE Computer Society, 2016, pp. 1493–1500. [Online]. Available: https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0212

[38] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "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*, A. K. Elmagarmid and D. Agrawal, Eds. ACM, 2010, pp. 339–350. [Online]. Available: https://doi.org/10.1145/1807167.1807206

[39] S. Ashkiani, S. Li, M. Farach-Colton, N. Amenta, and J. D. Owens, "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, 2018, pp. 430–440. [Online]. Available: https://doi.org/10.1109/IPDPS.2018.00053

[40] M. A. Awad, S. Ashkiani, R. Johnson, M. Farach-Colton, and J. D. Owens, "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*, J. K. Hollingsworth and I. Keidar, Eds. ACM, 2019, pp. 145–157. [Online]. Available: https://doi.org/10.1145/3293883.3295706

[41] S. You, J. Zhang, and L. Gruenwald, "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*, V. Chandola and R. R. Vatsavai, Eds. ACM, 2013, pp. 23–31. [Online]. Available: https://doi.org/10.1145/2534921.2534949

[42] S. K. Prasad, M. McDermott, X. He, and S. Puri, "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, 2015, pp. 618–627. [Online]. Available: https://doi.org/10.1109/IPDPSW.2015.127

[43] M. Lopresti, F. Piccoli, and N. Reyes, "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*, ser. Communications in Computer and Information Science, P. Pesado and G. Gil, Eds., vol. 1584. Springer, 2021, pp. 183–200. [Online]. Available: https://doi.org/10.1007/978-3-031-05903-2_13

[44] X. Zhong, Y. Zhang, Y. Chen, C. Li, and C. Xing, "Learned index on GPU," in *38th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2022, Kuala Lumpur, Malaysia, May 9, 2022*. IEEE, 2022, pp. 117–122. [Online]. Available: https://doi.org/10.1109/ICDEW55742.2022.00024