ORNL/TM-2024/3259

Revising Apetrei's bounding volume hierarchy construction algorithm to allow stackless traversal

Andrey Prokopenko
Damien Lebrun-Grandié

**Approved for public release.
Distribution is unlimited.**

February 2, 2024

ORNL IS MANAGED BY UT-BATTELLE LLC FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2024/3259

Computational Sciences and Engineering Division

REVISING APETREI'S BOUNDING VOLUME HIERARCHY CONSTRUCTION ALGORITHM TO ALLOW STACKLESS TRAVERSAL

Andrey Prokopenko
Damien Lebrun-Grandié

February 2, 2024

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831
managed by
UT-BATTELLE LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

ABSTRACT

Stackless traversal is a technique to speed up range queries by avoiding usage of a stack during the tree traversal. One way to achieve that is to transform a given binary tree to store a left child and a skip-connection (also called an escape index). In general, this operation requires an additional tree traversal during the tree construction. For some tree structures, however, it is possible to achieve the same result at a reduced cost. We propose one such algorithm for a GPU hierarchy construction algorithm proposed by Karras in Karras 2012. Furthermore, we show that our algorithm also works with the improved algorithm proposed by Apetrei in Apetrei 2014, despite a different ordering of the internal nodes. We achieve that by modifying Apetrei's algorithm to restore the original Karras' ordering of the internal nodes. Using the modified algorithm, we show how to construct a hierarchy suitable for a stackless traversal in a single bottom-up pass.

1. INTRODUCTION

Tree structures, such as bounding volume hierarchy (BVH), octrees and kd-trees, are used to accelerate the search for close geometric objects. Such trees are used in many applications, including computer graphics (ray tracing, collision detection), molecular dynamics, geographic information systems, cosmology, and others.

The emergence of GPU accelerators spurred efforts to develop highly parallel versions of the tree algorithms. Reducing thread execution divergence (executing different code) and data divergence (reading or writing disparate locations in memory) is highly desirable in parallel implementations, particularly for accelerators with thousands of threads (such as GPUs). The construction phase, in particular, is especially challenging on GPUs. An idea of parallelization of BVH construction by using a space-filling curve (called linear BVH, or LBVH) was first proposed in Lauterbach et al. 2009, with further improvements in Pantaleoni and Luebke 2010; Garanzha, Pantaleoni, and McAllister 2011. The first fully parallel algorithm allowing construction of all internal nodes concurrently was introduced in Karras 2012, and further improved in Apetrei 2014. The latter algorithm is considered to be the fastest BVH construction algorithm on GPUs. Both Karras' and Apetrei's algorithms are widely used Howard et al. 2019; Lebrun-Grandié et al. 2020, and may serve as an intermediate step for constructing a higher quality BVH Karras and Aila 2013; Domingues and Pedrini 2015.

Search indexes have to support different types of search queries. The *range* search finds all objects that intersect with a query object. Examples of the range search include finding all objects within a certain distance and finding all triangles in the scene that a ray intersects with.

Range search is typically implemented using a stack to keep track of the nodes to traverse. However, usage of stacks is undesirable on GPUs as it may lead to lower occupancy due to higher memory demands per thread. To avoid it, researchers developed stackless traversal, a technique to avoid explicitly managing a stack of node pointers for each thread during the traversal. The approach in Torres, Martín, and Gavilanes 2009 introduced an idea of a *skip connection* (also called *escape index*), which is a node index where the traversal should proceed if the intersection test with the current node is not satisfied, or if the node is a leaf node.

Stackless traversal requires a modification of a hierarchy, replacing right children with skip connections. Typically, this requires an extra tree traversal pass during the construction. In this work, we show that Karras' internal node numbering allows a short calculation of the skip-connections.

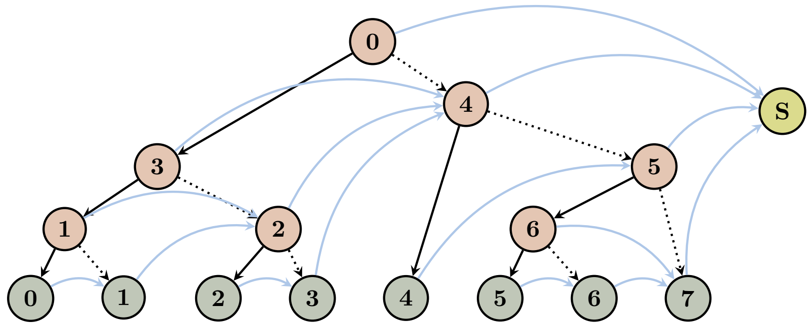


Figure 1. An example of a hierarchy with skip-connections. The right children (dotted lines) are replaced with skip-connections (blue curves). The skip-connections on the right-most path point to the sentinel node S.

While Karras 2012 and Apetrei 2014 result in an identical hierarchy structure, they produce different ordering of the internal nodes. Compared to the Karras' ordering, in which siblings have subsequent indices, the siblings in the Apetrei ordering may have an arbitrary large gap between them. The unfortunate side effect of this new ordering is that the skip-connections can no longer be easily set.

In the paper, we propose an elegant modification to the Apetrei's algorithm which restores the original Karras node ordering. The proposed fix requires only a slight algorithm change. It retains the performance gains in the hierarchy construction, and allows construction of a hierarchy with the skip connections in a single bottom-up pass.

To summarize, our key contributions are:

- We demonstrate a straightforward way to determine skip-connections in Karras' algorithm.
- We show how Apetrei's algorithm can be modified to restore Karras' internal node ordering.
- We provide a single bottom-up traversal algorithm to generate a hierarchy with skip-connections.
- We include pseudo-codes for both construction and traversal algorithms to allow readers to easily implement them in their code.

The remainder of the paper is organized as follows. Section 2 provides an overview of the stackless traversal. In Section 3, we provide an overview of both Karras' and Apetrei's algorithms and highlight their differences. We describe our modification to the Apetrei's algorithm and provide the resulting single bottom-up construction method in Section 4. Finally, we mention our implementation in Section 5.

2. STACKLESS TRAVERSAL

Stackless traversal is a technique to avoid explicitly managing a stack of node pointers for each thread. The approach in Torres, Martín, and Gavilanes 2009 introduced *skip-connections* (also called *escape index*), an index of a node where the traversal should proceed if the intersection test with the current node is not satisfied, or if the node is a leaf node. In Figure 1, the right children of the internal nodes are removed (denoted by dotted lines), and skip-connections (blue curves) are introduced. For the nodes on the right-most path, the skip-connections point to the artificial terminal node called *sentinel*.

A critical observation is that each skip-connections points to the right child of the last internal node that a given node is in the left subtree of. The only exceptions to this rule are the nodes on the right-most path (including the root node), which all point to the sentinel.

Algorithm 1 demonstrates the stackless traversal for a range search. If an encountered node does not satisfy the predicate, its subtree is avoided by immediately using skip-connection. Otherwise, either the left child is explored next (for internal nodes), or a positive match is processed (leaf node).

Algorithm 1 Stackless tree traversal algorithm using skip-connections for a predicate *query*. Each node N stores a skip-connection N_{skip}. In addition, each internal node stores a left child N_{left}. ♦ denotes the sentinel node.

```
N ← I_0                                        ▷ Start from the root node
repeat
    if query is satisfied on N then
        if N is a leaf node then
            Store the result or perform an operation
            N ← N_skip
        else
            N ← N_left
    else
        N ← N_skip
until N = ♦
```

3. A TALE OF TWO ORDERINGS

3.1 KARRAS' ALGORITHM

Given n primitives, the construction algorithm proposed in Karras 2012 is done in several steps:

1. calculate Morton indices for the primitives;
2. sort Morton indices;
3. generate hierarchy structure;
4. compute the bounding boxes of the internal nodes.

The first two steps produce sorted Morton indices $\mathcal{M} = \{m_i\}_{i=0}^{n-1}$ of the provided geometric objects. In step 3, the algorithm constructs a binary radix tree as a hierarchical representation of the common prefixes of a given set of keys (Morton indices in this case). The constructed hierarchy has n leaf and $n-1$ internal nodes.

Let $\delta(i,j)$ function be the longest common prefix between keys m_i and m_j for $0 \leq i < j < n$, and $+\infty$ for all other indices. We also define $\delta^*(i) = \delta(i, i+1)$ for convenience.

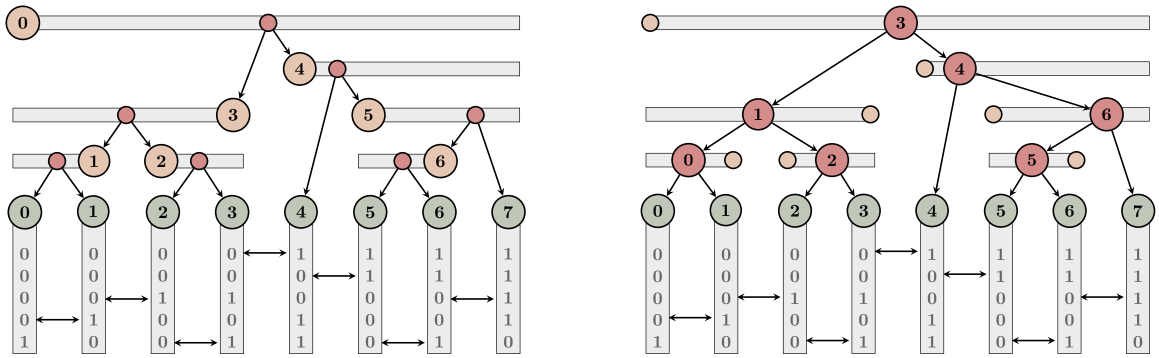
The Karras' idea is that each internal node covers a linear range of keys, and partitions its keys according to the highest differing bit in its range. The split position γ for an internal node covering the range $[i,j]$, $0 \leq i < j < n$, must satisfy $\delta(\gamma, \gamma+1) = \delta(i,j)$. The ranges of the children of this internal node are then $[i, \gamma]$ and $[\gamma+1, j]$.

To perform step 3 completely in parallel, the algorithm assigns internal node indices to correspond to the split position in their parent. The children of an internal node with a split γ are assigned indices γ and $\gamma+1$ in either the internal node array $\mathcal{I} = \{I_k\}_{k=0}^{n-2}$, or the leaf node array $\mathcal{L} = \{L_k\}_{k=0}^{n-1}$, assuming they are stored separately. This way, an internal node information can be fully ascertained by its range and its split. Karras' algorithm determines these values using linear and binary search through $\mathcal{M}$ using δ function. The root node is assigned index 0. For more details, see Karras 2012.

In the Karras layout, the index of each internal node coincides with one of the bounds of its range. Specifically, if an internal node with a range $[i,j]$ is a left child of its parent, its index is j; otherwise, if it is a right child, it is i. It can also be seen that it coincides with the range bound that has a smaller out of two values $\delta^*(i-1)$ and $\delta^*(j)$ (the root node index is always that of its left range bound). We will use this property to modify the Apetrei's algorithm.

An example of a constructed hierarchy for a set of Morton indices is shown in Figure 2a ($n = 8$). The internal nodes $\mathcal{I}$ are shown in orange, leaf nodes $\mathcal{L}$ are in green, and the split position for each internal node in red. The ranges for the internal nodes are denoted by gray boxes. For instance, internal node I_3 covers the range $[0,3]$ and has the split position $\gamma = 1$, thus having two children I_1 and I_2 (both internal nodes). Similarly, I_4 covers the range $[4,7]$ with a split $\gamma = 4$ and children L_4 and I_5 (one leaf and one internal node).

One can observe, that for the Karras' node ordering, the index of the target node of a skip connection is going to be the right range of the nodes spanned by its parent, incremented by one. Algorithm 2 shows the calculation of the skip-connection target for a given node N. If the right boundary range corresponds to the rightmost leaf, this indicates that the node is on the right-most side, so that its skip-connection should point to the sentinel node. For all other nodes, the skip-connection will point to the node whose index the right range boundary incremented by one. While the index is straightforward to determine, an additional calculation (line 5) is required to figure out whether it belongs to a leaf or an internal node.



(a) Karras **(b) Apetrei**

Figure 2. The ordering of the internal nodes in Karras' and Apetrei's algorithms.

Algorithm 2 Skip-connections in Karras' node ordering for a node N. ⧫ denotes the sentinel node.

---

if $\text{range}_{right} = n - 1$ **then**
  $N_{skip} \leftarrow$ ⧫
else
  $r \leftarrow \text{range}_{right} + 1$
  if $\delta^*(r-1) < \delta^*(r)$ **then**
    $N_{skip} \leftarrow L_r$
  else
    $N_{skip} \leftarrow I_r$

3.2 APETREI'S ALGORITHM

The algorithm modification proposed in Apetrei 2014 merged steps 3 and 4 of the Karras' algorithm into a single bottom-up step. In contrast with Karras algorithm, where each internal node constructs its range and determines a split independently, in Apetrei's approach the ranges get merged in the bottom-up traversal starting from the leaves, resulting in a faster algorithm, while also being easier to implement and requiring fewer lines of code. The internal nodes are indexed using the split positions γ rather than one of the ends of the corresponding range. An interesting side effect of such ordering is that the root node is no longer guaranteed to be I_0, and its index now needs to be stored in the hierarchy.

An example of a resulting hierarchy and internal node ordering is shown on Figure 2b. As the algorithm uses splits for internal node indices, the gap between indices of two siblings may become large. For example, the children of I_0 have a gap of 3.

The pseudocode for the Apetrei's algorithm is shown in Algorithm 3. Compared to the original paper, the presented version shows the complete procedure, except for the construction of the bounding boxes. The ATOMICCAS function performs the atomic Compare-And-Swap operation. It compares the contents of a memory location (first argument) with a given value (second argument). If they are the same, it overwrites the contents of that location with a new given value (third argument). This is done as a single atomic operation. It returns the value read from the memory location (*not* the value written to it). By comparing the return value of ATOMICCAS with the initialization value of the store array, it can be determined whether that location has already been modified. If it was, it indicates that the current thread is the second thread up and may proceed

Algorithm 3 Apetrei's algorithm. For simplicity, the construction of the bounding boxes is omitted. $\diamondsuit$ denotes an invalid entry value (e.g., -1).

Initialize all entries in store to $\diamondsuit$
for all leaf node with index $i \in [0, n-1]$ **in parallel do**
  $\text{range}_{left} \leftarrow i$, $\delta_{left} \leftarrow \delta^*(\text{range}_{left} - 1)$
  $\text{range}_{right} \leftarrow i$, $\delta_{right} \leftarrow \delta^*(\text{range}_{right})$
  repeat
    if $\delta_{right} < \delta_{left}$ **then** ▷ Left child of its parent
      $p \leftarrow \text{range}_{right}$ ▷ Apetrei index p
      if $\text{range}_{left} = \text{range}_{right}$ **then** $I_{p,left} \leftarrow L_i$ **else** $I_{p,left} \leftarrow I_i$
      $\text{range}_{right} \leftarrow \text{ATOMICCAS}(\text{store}_p, \diamondsuit, \text{range}_{left})$
      if $\text{range}_{right} = \diamondsuit$ **then return**
      $\delta_{right} \leftarrow \delta^*(\text{range}_{right})$ ▷ Recompute outdated value
    else ▷ Right child of its parent p
      $p \leftarrow \text{range}_{left} - 1$
      if $\text{range}_{left} = \text{range}_{right}$ **then** $I_{p,right} \leftarrow L_i$ **else** $I_{p,right} \leftarrow I_i$
      $\text{range}_{left} \leftarrow \text{ATOMICCAS}(\text{store}_p, \diamondsuit, \text{range}_{right})$
      if $\text{range}_{left} = \diamondsuit$ **then return**
      $\delta_{left} \leftarrow \delta^*(\text{range}_{left} - 1)$ ▷ Recompute outdated value
    $i \leftarrow p$
  until $r_i^{right} = r_i^{left} + n - 1$

further. Otherwise, as the first thread up, the thread exits the procedure.

The Algorithm 3 includes several additional optimizations not present in Apetrei 2014. First, the algorithm keeps track of the results of δ^* for the ranges, updating them only when necessary. This results in fewer memory loads of the Morton indices array which exhibit a random access pattern[1]. Second, the temporary storage serves dual purpose, both as a flag for allowing only one thread up, as well as for temporary storage of the opposite range, reducing memory allocation. We also note that, as recommended in Apetrei 2014, the δ^* function is switched from computing the common prefix (as in the Karras' algorithm) to a simpler XOR evaluation. If the Morton codes are identical, we follow the Karras idea of augmenting the key with a bit representation of its index.

1. We found that computing δ^* values as part of the bottom-up procedure is faster than pre-computing and storing them beforehand.

4. MODIFIED APETREI'S ALGORITHM WITH SKIP-CONNECTIONS

We modify the Apetrei's algorithm to restore the Karras ordering of the internal nodes and installation of skip-connections. The modified version is presented in Algorithm 4. Let us highlight the differences with Algorithm 3.

First, while the Apetrei index p of the parent node is still being calculated and used for referencing the temporary storage, the Karras index q is now used to reference the location of the parent node. The index computation on line 20 of Algorithm 4 uses the property we mentioned in Section 3.1. Specifically, that the Karras index coincides with its range boundary that has the smaller value of δ^*. Thus, by comparing the δ^* values of the parent range, we are able to figure out its Karras index.

Second, a parent node is now updated by a single thread instead of two. Specifically, lines 8 and 14 in Algorithm 3 are replaced by lines 21–26 in Algorithm 4. This is due to the fact that the determination of the parent index, q, now requires the knowledge of its full range, which is only available to the second thread up. On the flip side, a parent node can now determine the indices of its children, as knowing a split position γ (which is exactly the Apetrei index p in this case), the indices of the children are γ and $\gamma + 1$ in the internal or the leaf node array.

Third, given the knowledge of the Karras index q and the full range of the parent as part of the Apetrei's algorithm, we can now easily calculate the target for each of the skip-connections as was explained in Section 3.1. Lines 5–8 (lines 22–26) integrate Algorithm 2 into the procedure for leaf (internal) nodes, respectively.

Finally, the index i on line 27 is updated with the Karras' index instead of Apetrei's, and the loop termination condition is simplified, as I_0 is always the root in the Karras ordering.

Algorithm 4 Modified Apetrei's algorithm with skip installation. For simplicity, the construction of the bounding boxes is omitted. $\diamondsuit$ denotes an invalid entry value (e.g., -1). $\blacklozenge$ denotes the sentinel node.

```
Initialize all entries in store to $\diamondsuit$
for all leaf node with index $i \in [0, n-1]$ in parallel do
    $\text{range}_{left} \leftarrow i, \quad \delta_{left} \leftarrow \delta^*(i-1)$
    $\text{range}_{right} \leftarrow i, \quad \delta_{right} \leftarrow \delta^*(i)$
    if $i = n-1$ then
        $L_{i,skip} \leftarrow \blacklozenge$
    else
        if $\delta_{right} < \delta^*(i+1)$ then $L_{i,skip} \leftarrow L_{i+1}$ else $L_{i,skip} \leftarrow I_{i+1}$
    repeat
        if $\delta_{right} < \delta_{left}$ then                          ▷ Left child of its parent
            $p \leftarrow \text{range}_{right}$                                ▷ Apetrei index $p$
            $\text{range}_{right} \leftarrow$ ATOMICCAS($\text{store}_p, \diamondsuit, \text{range}_{left}$)
            if $\text{range}_{right} = \diamondsuit$ then return
            $\delta_{right} \leftarrow \delta^*(\text{range}_{right})$                 ▷ Recompute outdated value
        else                                                          ▷ Right child of its parent
            $p \leftarrow \text{range}_{left} - 1$
            $\text{range}_{left} \leftarrow$ ATOMICCAS($\text{store}_p, \diamondsuit, \text{range}_{right}$)
            if $\text{range}_{left} = \diamondsuit$ then return
            $\delta_{left} \leftarrow \delta^*(\text{range}_{left})$                   ▷ Recompute outdated value
        if $\delta_{right} < \delta_{left}$ then $q \leftarrow \text{range}_{right}$ else $q \leftarrow \text{range}_{left}$   ▷ Karras index $q$
        if $\text{range}_{left} = q$ then $I_{q,left} \leftarrow L_i$ else $I_{q,left} \leftarrow I_i$
        if $\text{range}_{right} = n-1$ then
            $I_{q,skip} \leftarrow \blacklozenge$
        else
            $r \leftarrow \text{range}_{right} + 1$
            if $\delta_{right} < \delta^*(r)$ then $I_{q,skip} \leftarrow L_r$ else $I_{q,skip} \leftarrow I_r$
        $i \leftarrow q$
    until i = 0
```

5. IMPLEMENTATION

The described algorithm is implemented as part of the ArborX library Lebrun-Grandié et al. 2020. The code is available at https://github.com/arborx/ArborX.

ACKNOWLEDGEMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

6. REFERENCES

Apetrei, C. 2014. "Fast and Simple Agglomerative LBVH Construction." In *Computer Graphics and Visual Computing (CGVC),* edited by Rita Borgo and Wen Tang. The Eurographics Association. ISBN: 978-3-905674-70-5. https://doi.org/10.2312/cgvc.20141206.

Domingues, L. R., and H. Pedrini. 2015. "Bounding Volume Hierarchy Optimization Through Agglomerative Treelet Restructuring." In *Proceedings of the 7th Conference on High-Performance Graphics,* 13–20. HPG '15. New York, NY, USA: ACM. ISBN: 978-1-4503-3707-6, accessed October 16, 2019. https://doi.org/10.1145/2790060.2790065. http://doi.acm.org/10.1145/2790060.2790065.

Garanzha, K., J. Pantaleoni, and D. McAllister. 2011. "Simpler and faster HLBVH with work queues." In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics,* 59–64.

Howard, M. P., A. Statt, T. M. Truskett, and A. Z. Panagiotopoulos. 2019. "Quantized bounding volume hierarchies for neighbor search in molecular simulations on graphics processing units" [in en]. *Computational Materials Science* 164 (June): 139–146. ISSN: 0927-0256, accessed April 28, 2020. https://doi.org/10.1016/j.commatsci.2019.04.004. https://orproxy.lib.utk.edu:2054/science/article/pii/S092702561930206X.

Karras, T. 2012. "Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees." In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics,* 33–37. EGGH-HPG'12. Goslar Germany, Germany: Eurographics Association. ISBN: 978-3-905674-41-5, accessed October 16, 2019. https://doi.org/10.2312/EGGH/HPG12/033-037. https://doi.org/10.2312/EGGH/HPG12/033-037.

Karras, T., and T. Aila. 2013. "Fast Parallel Construction of High-quality Bounding Volume Hierarchies." In *Proceedings of the 5th High-Performance Graphics Conference,* 89–99. HPG '13. New York, NY, USA: ACM. ISBN: 978-1-4503-2135-8, accessed October 16, 2019. https://doi.org/10.1145/2492045.2492055. http://doi.acm.org/10.1145/2492045.2492055.

Lauterbach, C., M. Garland, S. Sengupta, D. Luebke, and D. Manocha. 2009. "Fast BVH Construction on GPUs" [in en]. *Computer Graphics Forum* 28, no. 2 (April): 375–384. ISSN: 1467-8659, accessed June 9, 2020. https://doi.org/10.1111/j.1467-8659.2009.01377.x. https://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2009.01377.x.

Lebrun-Grandié, D., A. Prokopenko, B. Turcksin, and S. R. Slattery. 2020. "ArborX: A Performance Portable Geometric Search Library." *ACM Transactions on Mathematical Software* 47, no. 1 (December): 2:1–2:15. ISSN: 0098-3500. https://doi.org/10.1145/3412558. https://doi.org/10.1145/3412558.

Pantaleoni, J., and D. Luebke. 2010. "HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry." In *Proceedings of the Conference on High Performance Graphics,* 87–95.

Torres, Roberto, Pedro J. Martín, and Antonio Gavilanes. 2009. "Ray casting using a roped BVH with CUDA." In *Proceedings of the 25th Spring Conference on Computer Graphics,* 95–102. SCCG '09. New York, NY, USA: Association for Computing Machinery, April. ISBN: 978-1-4503-0769-7, accessed August 10, 2020. https://doi.org/10.1145/1980462.1980483. https://doi.org/10.1145/1980462.1980483.