





Optimizing LBVH-Construction and Hierarchy-Traversal to accelerate k NN Queries on Point Clouds using the GPU

J. Jakob  and M. Guthe 

Visual Computing, University of Bayreuth, Germany
{johannes.jakob, michael.guthe}@uni-bayreuth.de

Abstract

Processing point clouds often requires information about the point neighbourhood in order to extract, calculate and determine characteristics. We continue the tradition of developing increasingly faster neighbourhood query algorithms and present a highly efficient algorithm for solving the exact neighbourhood problem in point clouds using the GPU. Both, the required data structures and the k NN query, are calculated entirely on the GPU. This enables real-time performance for large queries in extremely large point clouds. Our experiments show a more than threefold acceleration, compared to state-of-the-art GPU based methods including all memory transfers. In terms of pure query performance, we achieve over 10^5 answered neighbourhood queries per millisecond for 16 nearest neighbours on common graphics hardware.

Keywords: nearest neighbour, radius queries, bounding volume hierarchies, bvh optimization, acceleration structures

ACM CCS: • Theory of computation → Nearest neighbour algorithms; Massively parallel algorithms

1. Introduction

A multitude of algorithms in various domains rely on finding the k nearest neighbours (k NN) of a query in a data set, including databases, machine learning, computer vision, computational geometry, robotics, computer graphics or physical simulations. The size of data sets has increased enormously in recent years. Improved scanning technologies have led to the ability to scan huge environments or objects in extremely high resolution. Since such point clouds rarely have more information than an additional vertex normal, various algorithms have to be applied to the scan data, e.g. registration of several data sets, calculation or smoothing of surface normals, removal of residual noise and outliers from the scanning process, sub-sampling, segmentation, reconstruction and feature matching. These almost always require finding the k nearest neighbours for each data set point, which is a very expensive and time-consuming step and a challenge for real-time capability. To cope with a large amount of data in as little time as possible, the trend is to use GPUs to accelerate such costly steps, as they have established themselves as a reliable co-processor of the CPU for highly data-parallel applications.

Our contribution is a performance improvement over previous solutions running on the GPU to enable real-time performance for

exact k NN queries even in dynamic scenarios. To achieve this, we create and optimize a spatial acceleration data structure for point-clouds and use a SIMD register-memory based priority queue for k NN search. Both, construction of the data structure and the query are executed entirely on graphics hardware, thus, avoiding expensive memory-copy bottlenecks. Our experiments with synthetic as well as real-world data sets show that our approach is scalable and on average about 3.3 times faster compared to state-of-the-art methods.

2. Problem Statement

Nearest neighbour search describes the similarity search in data sets. Consider the metric space \mathcal{S} , a data set $X = (x_1, \dots, x_n) \subseteq \mathcal{S}$, a query set $Q = (q_1, \dots, q_m) \subseteq \mathcal{S}$ and a distance metric $m : \mathcal{S}^2 \rightarrow \mathbb{R}$. The k -nearest neighbour search then is the task of finding the k closest (w.r.t. m) data points to each $q \in Q$ from the data set X .

High-dimensional metric spaces, memory and/or runtime restrictions and the number of query points as well as regular data set changes are challenging for any algorithm. In the last decades, several approaches have been proposed with proven upper bounds of computational complexity [Ben75, AMN*98], that generally

seek to reduce the number of distance computations using different types of trees. However, for many methods based on k nearest neighbours the computation time needed to find these for all input points, still remains the bottleneck. For this reason, many algorithms are specially designed for certain application areas.

In this paper, we consider applications that require the k nearest neighbours (k NN) or k approximate neighbours (k ANN) where S is a low-dimensional euclidean space ($d = 3$) and m is the L_2 -norm. This occurs especially in the context of real-time processing of 3D data. Examples are simulations, e.g. SPH [GJM77], point-cloud registration [BM92, PCS15] and processing or real-time photon-mapping [PDC*03, MLM13], which must find the k NN for at least each individual data point in the data set. In various cases this set changes constantly (e.g. registration, simulations, photon-mapping), which also requires any index structures to be updated or rebuilt each time and forbids any expensive off line preprocessing.

Our main contributions addressing these problems in this paper are:

- A highly parallel bottom up LBVH optimization step, specialized on point-cloud data.
- A stackless BVH traversal algorithm that can find all exact k NN using a register based priority queue.

3. Related Work

Since the literature for finding the k NN is vast, we limit ourselves to GPU based k NN techniques. The methods developed so far can be divided into two main classes: Brute force approaches and selection based procedures.

3.1. Brute-force methods

There are many techniques to solve the k NN problem via a brute-force approach using the GPU. Generally, for each query these methods initiate a calculation of the distance to all data points in the data set. The two most important approaches for calculating the distance matrix are a direct implementation with a self-developed kernel [GDB08, ZHWG08, KZ09, LLWJ09, LWLJ09, LLWJ10, BGTP10, KH10, BGT*11, KH12, SPS12, ARBM12, DKMD13], and the use of an already well optimized matrix multiplication routine [BDHK06, GDB08, SPS12, DKMD13, LA15, THE*15, JDJ17, KD18], e.g. using *cuBLAS*. User-defined direct implementations are typically optimized by tiling, which divides the distance matrix into several submatrices (tiles) of equal size. The used tile size is optimized in such a way, that a group of query and reference points can be stored in fast shared memory and reused by threads within the same compute block. The calculated distances must then be sorted and the nearest k extracted.

1) *Squared Distance matrix*: Bustos **et al.** [BDHK06] were one of the first using GPUs for NN computation. They computed the squared distance matrix using custom shaders and performed multiple texture reductions to obtain the final nearest neighbour. By using the latest GPGPU architecture, it was possible for Garcia

et al. [GDB08] to assign one thread per query for distance sorting after the brute force step. Instead of using one thread per query, Sismantis **et al.** [SPS12] used a parallelized truncated bitonic sort per query, while Dashti **et al.** [DKMD13] relied on radix sort from Nvidias thrust library. Li **et al.** [LA15] decided to integrate a truncated merge sort directly into the matrix multiplication routine, which discards candidates as it becomes clear that they cannot belong to the top k . A k -selection by Tang **et al.** [THE*15] was accelerated by using a merge queue, a buffered search, and hierarchical partition to better support the SIMD architecture of GPUs. Johnson **et al.** [JDJ17] were able to handle data sets that are too large for current GPU main memory by relying on *cuBLAS* and a specialized k -selection algorithm. A multi GPU approach was proposed by Klusek **et al.** [KD18], which also computes the squared distance matrix with a subsequent sort operation to extract the k NN, but offer a better data distribution among the available GPUs.

2) *custom distance kernel*: Garcia [GDB08], Zhao [ZHWG08] and Kuang **et al.** [KZ09] use a custom matrix multiplication, paired with an insertion sort or radix sort algorithm respectively to find the k NN, leveraging the speed of modern sort libraries. Many approaches split the distance computation of each query into blocks. Liang **et al.** [LLWJ09, LWLJ09] find the local k NN within each block by simultaneously testing each distance against all others. A single thread per query then merges the lists using heap selection. Liang, Kato and Barrientos [LLWJ10, BGTP10, KH10, BGT*11, KH12] proposed different heap-based approaches for the k NN selection process, using one thread with a max heap per query or a heap-based reduction over multiple blocks [BGT*11]. Truncated sort was introduced by Sismanis **et al.** [SPS12]. Elements are removed from the vector when it is clear that they cannot belong to the set of smallest k . They describe several algorithms and show that their truncated bitonic sort has excellent performance on the GPU. Arefin **et al.** [ARBM12] maintain an unsorted array of size k for each query and a pointer to the largest element in the array. A single thread maintains this structure on each level with a linear scan. Dashti **et al.** [DKMD13] also use a radix-sort approach but on the whole distance matrix. Query-distances of the candidates are first sorted collectively and then sorted by search index to separate the results for each query. Then only the top k elements have to be extracted.

All brute-force approaches have in common, that they are only suitable for small data- and query-sets due to high memory requirements and only perform competitively on high dimensional data.

3.2. Selection based procedures

Selection based k NN methods are more diverse and can be further divided into either hashing, graph-based or spatial subdivision strategies bringing also some asymptotically more efficient algorithms to the GPU.

1) *Hashing based approaches*: Most of the used hashing-based algorithms are variants of locality-sensitive hashing (LSH) [PLM10, PM11, PM12, LŽ15]. LSH uses several hash functions of the same type, which are location-sensitive. This enables neighbouring points to be more likely to fall into the same hash bucket than points that are far apart from each other. During the query stage,

the search point is hashed with all hash-functions and then a linear search is performed in the set of data points with which the query matches. Pan **et al.** [PM11] additionally use a parallel RP-tree or random projections [PM12] to pre-partition their data sets into several groups, so that items similar to each other are clustered together. Subsequently Bi-Level LSH is executed on each partition tree and the k NN are then extracted via short-list search. Lukac **et al.** improve the efficiency of existing LSH approaches by using Multi-Probe LSH [LŽ15], that also scans the nearby hash buckets of the one into which a query point is hashed. Therefore, a smaller number of hash tables is required to achieve a certain accuracy. Choosing the hash function affects performance and must be done carefully. For k ANN a deterministic skip-list data structure is used to hold the k ANN neighbours indices and distances. Wiechollek **et al.** [WWSHL16] introduced a two level product quantization tree (built upon a combination of inverted multi-index and hierarchical PQ) for high dimensional data sets. They combine their method with a new re-ranking algorithm based on closest-line projections and a bin ordering heuristic, which results in good performance for very high dimensional data sets. However, most hashing based approaches only provide approximate nearest neighbours.

2) *k*NN-Graph based algorithms: Fast k NN queries can be achieved by using a k NN-Graph as acceleration data structure. Fu **et al.** [FC16] use a hierarchic divide-and-conquer algorithm to construct an initial k NN graph using *eight* randomized truncated k-d trees for graph-initialization. A nearest neighbour descent is then applied to refine the graph. They report fast query and build times, but only produce approximate results.

3) *Spatial partitioning methods*: Spatial partitioning methods solve the k NN problem by constructing a spatial index. The index generation can be roughly categorized into those that partition the data and those that partition the space. The former try to cluster data on the basis of their spatial proximity as, i.e. Li **et al.** [LSP*12], which create multiple lists of the data set with shifted points. These are then sorted using a space filling curve. The k ANN can be thus found by considering only the k preceding and k succeeding points in each shifted list. Hachisuka **et al.** [HJ10] use only a single hash list with exactly one data point per hash entry and can therefore only provide approximate k NN results. Space partitioning based approaches on the other hand make use of grids [PDC*03, LTF*09, SBMN16], octrees [GGG08] and k-d trees [ZHWG08, QMN09, ML09]. The k NN are searched by backtracking [QMN09] or heap-based priority search in the underlying tree-structure. Grid based methods [MB17] have the advantage of a very good and fast radius search, but can degenerate to a full search for exact k NN. This in turn is the strength of hierarchical techniques like octrees or k-d trees. However, all these methods are limited to low dimensional problems but provide unbeatable performance compared to the previous approaches in these cases.

4. Design and Implementation

In the following, we discuss design and implementation of our approach. As already stated, spatial subdivision methods are very suitable for low-dimensional data and allow asymptotically efficient queries. Our proposed method is therefore based on a k-d tree.

Algorithm 1. LBVH Construction

```

1: procedure CONSTRUCTTREE
2: for each leaf with global index  $i \in [0, n)$  in parallel
3: COMPUTELEAFBOUNDSleaves[i]
4:  $curr \leftarrow$  FINDPARENT  $i - 1, i, \text{leaves}[i].id$ 
5: while (ATOMICXORworker_flag[curr], 1) do
6: MERGECHILDRENAABBcurr
7:  $left \leftarrow$  rangeOfKeys[curr].x
8:  $right \leftarrow$  rangeOfKeys[curr].y
9:  $curr \leftarrow$  FINDPARENT left, right, curr
10: THREADFENCE

```

Pseudocode for choosing the correct parent node. n is number of points in data set. Pointer, bounding boxes and per node primitive count are computed and set in the findParent(.) function. For implementation details of methods and needed data-structures we refer to [Ape14].

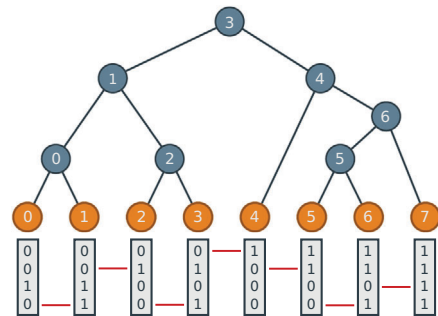


Figure 1: Example LBVH according to [Ape14]. Leaf nodes are numbered from 0 to 7 and internal nodes from 0 to 6. Leaves were previously sorted based on their Morton code. The range of Morton keys covered by each node is indicated by a horizontal red bar.

4.1. LBVH construction

For the initial spatial acceleration structure we use the fast LBVH construction of Apetrei **et al.** [Ape14], which is based on ordering primitives along a space-filling curve. Compared to previous methods, this bottom-up construction algorithm is able to generate both tree-hierarchy and enclosing bounding boxes in one single and simple kernel launch as shown in Algorithm 2.

We first compute a Morton code for each item in the data set and sort all points accordingly using a parallel radix-sort. Subsequently, after creating the leaf nodes, an initial LBVH is built in a single bottom-up traversal by choosing the parent and simultaneously computing the bounding box at each step. The resulting tree is shown in Figure 1.

Our implementation of the kernel proposed by Apetrei **et al.** differs only in that we store explicit parent pointers per node, the sum of all points in the current subtree (used during optimization), and force an explicit synchronization of the global memory write accesses as outlined in line 10 of Algorithm 2. This is required as Nvidia GPUs use a weakly-ordered memory model. The order in which a thread writes data to global (or shared) memory is not necessarily the order in which the data written by another thread is observed. Depending

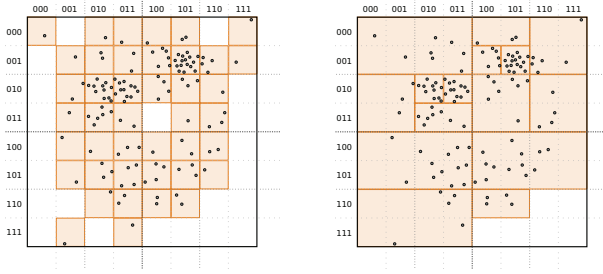


Figure 2: Example hierarchy merge. On the left side leaf nodes at full Morton resolution are shown. On the right we fused some leaf nodes (based on the underlying z -curve) to achieve a more homogeneous point density per node.

on the graphics card used (memory read/write ordering is different in different architectures), this leads to nondeterministic behaviour and invalid hierarchies since the pointers and keys set by the `FINDPARENT` method in line 9 are read directly in the next iteration during bottom up traversal. For implementation details of the listed methods, we refer to [Ape14].

4.2. Tree optimization

By using Morton codes for fast hierarchy generation, however, the space is discretized into a grid. Depending on the data set size and Morton code resolution, the point density and distribution within the data set, it may occur that many points share the same Morton key and are thus placed in the same leaf node, while the majority of the remaining leaves contain only one or very few data points. In terms of data-parallel processing this effect is adverse as it causes diverging threads during a tree traversal and thus a performance drop.

In order to alleviate this problem, we try to create a more shallow hierarchy with leaf nodes of ideally equal data density, as outlined in Figure 2. This is achieved by a second bottom-up traversal of the initial LBVH. At each inner node we decide whether to reduce it into a leaf node (depending on the data density of the left and right children) and continue upwards, or to do nothing and stop the traversal. Merged nodes, that are no longer needed are flagged accordingly. To prevent race conditions between two threads coming from a left and right subtree, only one thread is allowed to collapse and continue (see line 6 in Algorithm 3). The procedure is outlined in Algorithm 3.

This way we incrementally fuse spatially related parts of the data set without destroying the underlying tree and at the same time can reuse the pre-calculated bounding volumes, which results in extremely fast processing. A visual example of this procedure applied to the tree in Figure 1 is shown in Figure 3. During the bottom-up traversal a heuristic φ decides whether an internal node becomes a leaf node. The necessary pointer adjustments are then performed by the `MAKELEAF(...)`-method. In the following both are described in detail:

Collapse heuristic. The heuristic $\varphi(\text{node})$ decides whether the current node becomes a leaf node or not. We use a very simple point

Algorithm 2. Tree optimization kernel

```

1: procedure OPTIMIZE TREE bvh
2: for each leaf  $l$  in constructed bvh in parallel
3:  $curr \leftarrow bvh.GETPARENTl$ 
4: while do true traverse hierarchy bottom up
5:  $parent \leftarrow bvh.GETPARENTl$ 
6: if ( $\varphi curr$  and ( $thread_{ID} \leq curr$ ))
7:  $bvh.MAKELEAFcurr, l$ 
8:  $is\_valid[curr] \leftarrow false$ 
9:  $curr \leftarrow parent$ 
10: else if ( $\varphi curr$  and ( $thread_{ID} > curr$ )) then
11:  $is\_valid[l] \leftarrow false$ 
12: return
13: else
14: return

```

Pseudocode of our optimization kernel. As each leaf/node can be identified with a global index, we just flag only the deleted ones. This simplifies computing new memory positions for all nodes in the subsequent compaction step.

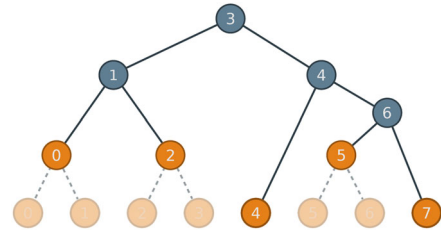


Figure 3: Example tree layout after optimization of tree depicted in Figure 1. From bottom up, subtrees were collapsed into a single new leaf node.

count limit: we compare the sum of the number of stored points in the left and right node with a user specified threshold and return true if this node should become a leaf node:

$$\varphi(v) = \begin{cases} true & \text{if } \sum_{AABB(v)} \leq \Theta \\ false & \text{else} \end{cases}$$

with $\sum = \# \text{points in current volume (AABB)}$
 $\Theta = \text{threshold points in volume.}$

This is trivial as we computed and stored the number of total contained points in each node during initial hierarchy buildup.

MakeLeaf. The key to an efficient hierarchy adjustment is in two aspects: First, during hierarchy construction we temporarily store the number of contained points in each internal node by adding the primitive number of left and right children during bottom up traversal. Second, the data set items are sorted according to their Morton code in memory. An internal node is thus easily turned into a leaf node by simply replacing it with the leftmost leaf node of its subtree.

Since we start the bottom up traversal at the tree leaves, each thread needs to remember the leaf-id it came from. The `MAKELEAF(...)` method then just adjusts pointers, the bounding box and the primitive count, as outlined in Algorithm 4 and visualized on

Algorithm 3. make internal to leaf node

```

1: procedure MAKELEAF $curr, leaf$ 
2:  $parent \leftarrow parent[curr]$ 
3:  $parent[leaf] \leftarrow parent$ 
4:  $aabb[leaf] \leftarrow aabb[curr]$ 
5:  $primitive\_cnt[leaf] \leftarrow subtree\_size[curr]$ 
6: if ( $left[parent] = curr$ ) then
7:  $left[parent] \leftarrow leaf$ 
8: else
9:  $right[parent] \leftarrow leaf$ 

```

Pseudo code of the function which merges an internal node into a leaf by reusing the leftmost leaf node of the subtree.

the right. In this example, the depicted leaf with number 5 will contain all stored primitives of the leaves 5 and 6 of the initial tree.

4.3. Tree compaction

Due to the fact that entire subtrees collapse into a single leaf node, the memory layout of the LBVH fragments. This impacts query performance, which is why we compact it in a final phase to allow better coalescing memory accesses. As usually only a few parts of the tree are removed and data ordering in memory does not necessarily have to be preserved, we use a highly modified parallel in-place compaction [DMG10]. This allows us to avoid a complete duplicate of the hierarchy and also maximizes memory throughput. It would also be possible to perform a simple compaction by copying the tree to a new memory location and adjusting the pointers on the fly. However, this includes nearly doubling the required memory for the hierarchy and requires expensive de-/allocations. We opted for the in-place method to achieve the fastest possible compaction and a low memory footprint.

4.4. Neighbourhood query

The main problem with hierarchical traversal-methods on a GPU is that only the most coherent part of the operations near the tree root are accelerated (everything cached, nearly no thread divergencies). Consequently, a multitude of highly incoherent per-thread workloads have to be processed.

In the following, two key aspects, which lead to increased query performance when combined, are discussed in more detail: *traversal scheme* and *register based priority queue*. We will also briefly explain how to modify our approach to allow a radius and an approximate k nearest neighbour search.

4.4.1. Backtracking traversal

Usually, on the CPU side heap-based traversal strategies are preferred, which use the quadratic distance of a bounding volume to the query point as priority key. On a GPU, however, this heap would have to be kept in global or shared device memory. Due to the necessary data-dependent heap updates this leads to random and irregular memory movements. The resulting latencies, warp divergencies and/or bank conflicts then cause the SIMD units to not

Algorithm 4. Register Priority Queue

```

#define min(x,y) (x<y?x:y)
#define max(x,y) (x<y?y:x)
#define CAS(x,y) { auto tmp=min(x,y); y=max(x,y);
                    x = tmp; }

template < typename KEY, int SIZE >
struct StaticInsertionSortPQ
{
    KEY _k[SIZE];
    int _size = 0;

    ...

    void push(KEY const key) {
        ++_size;
        _k[0] = key;
        sort();
    }

    void sort() {
        #pragma unroll
        for (int i(0); i<(SIZE-1); ++i)
            CAS(_k[i], _k[i+1]);
    }

    ...
}

```

being saturated. For this reason, we opted for a backtracking approach [HDW*11]. The main difference between a backtracking or heap-based traversal is that more nodes need to be visited, compared to using a latency-heavy binary heap of nodes not yet visited. In principal, this is the same as a depth-first search while using the information of the current found nearest neighbours to decide whether a subtree needs to be traversed or not. For further implementation details, the reader is referred to the supplemental material.

4.4.2. Register based priority queue

To keep track of the currently found nearest neighbours usually a second (maximum-) heap is used. For small k , a CPU can usually keep the entire heap in L1 cache, which enables extremely low latency and high bandwidth. However, as mentioned above, heaps generally do not show good data parallelism on GPUs.

We were inspired by the idea of utilizing registers for sorting network primitives on the GPU as proposed by Johnson **et al.** [JDJ17]. Our approach differs in that we use a simple array in device register memory for our currently found neighbours and keep it sorted using insertion sort. For different k NN sizes we use a compile-time unrolled insertion sort as shown in Algorithm 5. Consequently the compiler can create the code directly and we do not have to provide complex sorting networks for every possible array size and also emit less instructions. Each time, before inserting a new point, we test whether its distance is smaller than the current largest in the heap and insert it only if this is the case. Therefore it can safely be overwritten.

With this approach we benefit from vector parallelism, extremely low memory latency and easy implementation. To enable the CUDA compiler into keeping a sorted list in register only, everything must

be known at compile time and enough device registers must be available. During runtime we then choose the appropriate kernel.

4.4.3. Radius search

Closely related to the k NN search is the radius search, which returns all nearest neighbours within a specified search radius. Our approach can be easily modified to use this radius as the abort criterion during traversal. The only change to the backtracking algorithm is, that elements which are smaller than or equal to the current search radius are always inserted. The traversal is aborted if no subtree within the search radius is available. This is only limited by the fact, that the size of the k NN heap must be set in advance and remains fixed during the query.

4.4.4. Approximate k NN

If an exact k NN search is not necessary and to further accelerate our approach, our method can be easily extended to support an approximate k nearest neighbour search. This can be achieved by setting a limit for the number of visited nodes during the query phase. If this threshold is exceeded, the current traversal is aborted.

4.5. Algorithm parameters

Our approach therefore only depends on two selectable parameters, which influence hierarchy quality and thus LBVH construction and query runtime. In the following, we discuss their effects in detail:

MMorton code resolution The Morton code resolution (number of quantization bits per dimension) affects the initial bin size of the LBVH leaves. The higher the resolution, the fewer points will fall into the same bin, which increases size and build time of the initial LBVH and more subtrees have to be merged in the subsequent optimization phase. But with very inhomogeneous distributed point clouds or large-scale 3D scans with fixed scanning positions, it often occurs that an extremely high number of points collide into the same Morton cell, causing extremely unbalanced workloads.

Per Node point-threshold The selected point threshold during the optimization step influences the final hierarchy and thus the query- and optimization runtime. A small threshold per leaf node leads to more visited tree-nodes and therefore to longer traversal times and thread divergencies. A too large threshold results in a major part of the runtime during traversal being spent on pointless sorting of found points, which are later discarded anyway.

5. Results

To ensure a performance analysis that is as extensive as possible, we used both synthetic and real-world data sets from different areas of computer vision and graphics in the following tests. The selected data-sets, shown in Figure 4, provide a broad range of distribution patterns: from uniformly to extremely irregularly distributed (e.g. terrestrial laser scan with extreme densities in close proximity, as well as widely scattered individual measurements).



Figure 4: Visualization of our test data set. The shown point clouds (about 70) have different characteristics and are either real 3D scans (airborne, hand-held and stationary) with noise and outliers or synthetically obtained by sampling polygon meshes.

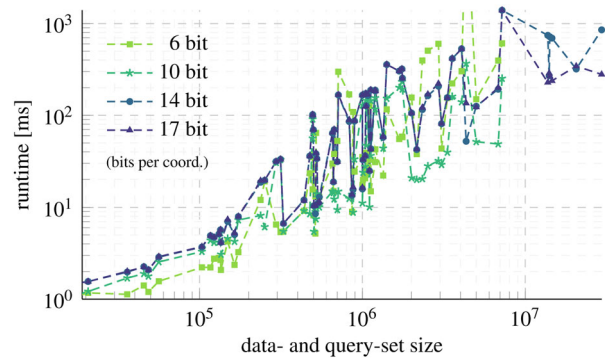


Figure 5: Runtime plot showing construction of initial LBVH. [Ape14] without any optimizations followed by a k NN query (with $k = 16$) for each point in the input data set without memory transfer times. Shown are different Morton code resolutions in bits per dimension. For large models ($> 5 \cdot 10^6$) a Morton code with less or equal to 32 bits in total was not sufficient, which is why no measurement results are available here. The prominent runtime peaks are caused by uneven distributed data-sets.

All measurements were performed on an AMD Ryzen™ 7 2700X CPU @ 3.7 GHz, 32 GB RAM with a NVIDIA GeForce™ GTX 2080TI, running under Linux 5.4.14 with NVIDIA driver version 440.44. We implemented and compiled our hierarchy construction and traversal algorithm with CUDA 10.2.

5.1. Hierarchy optimization analysis

In the following we discuss the parameter selection of our approach in detail and show that all presented optimizations contribute to the performance of our approach.

Morton code resolution: To achieve a good initial LBVH quality we tested different quantization resolutions and measured algorithm runtime, as shown in Figure 5. As expected, low resolutions are suitable for small point clouds as this reduces sorting overhead and creates a smaller hierarchy. For larger point clouds or severely inhomogeneous distributed ones, higher quantization resolutions are absolutely necessary. Due to massive differences between the point

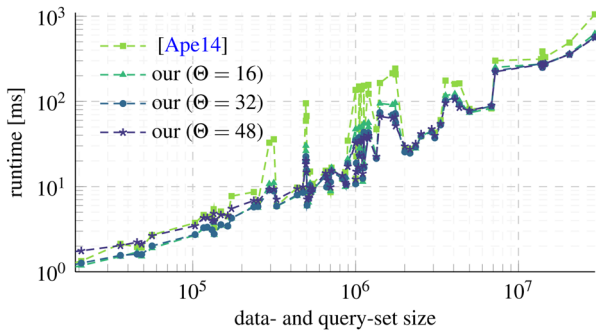


Figure 6: Runtime plot showing hierarchy construction (without compaction) and k NN query (with $k = 16$) for each point in the input data set without memory transfer times. Shown are runtimes achieved with LBVH by Apetrei *et al.* [Ape14] vs. our optimization step enabled using two different thresholds. Our proposed optimization step delivers faster and more consistent runtimes throughout the entire benchmark data set.

clouds in terms of their properties and characteristics, there are significant runtime differences visible. For simplicity, we chose a quantization resolution of 10 bits per dimension for all point clouds. If they are larger than $5 * 10^6$, we use 17 bits. This results in relatively balanced and favourable initial hierarchies, which can cope with very inhomogeneous data and can be optimized in the next step. Due to word size restrictions, a 64 bit key was used in these cases. **Per node point threshold:** To show the improvement of our optimization step, we compared the query performance using the initial hierarchy of Apetrei *et al.* [Ape14] to ours using different thresholds. Figure 6 shows the total runtime of hierarchy construction, with and without the optimization step and k NN query (with $k = 16$) for each point in the input data set without memory transfer times. This way it can be easily estimated whether the optimization overhead is justified. Furthermore the plot depicts, that the initial hierarchy does not scale well for point clouds with different characteristics. Especially with highly inhomogeneous data sets, there are abrupt runtime outliers. Our proposed optimization improves exactly these negative aspects and delivers faster results throughout the whole benchmark data set. We use a threshold value of 32, since higher values did not lead to any significant improvement. **Hierarchy memory compaction** As outlined in Section 4.3 we compact the memory layout after the optimization step using an in-place compaction to prevent memory fragmentation. In the following benchmark, we compare the query performance after the hierarchy optimization with a threshold of 32 using no compaction, a default out of place compaction and our used in-place compaction. Figure 7 shows the total runtime of hierarchy construction, optimization, compaction and k NN query (with $k = 16$) for each point in the input data set without memory transfer times for the three different profiles.

The graph clearly shows that the successive memory compaction step provides an increased global device-memory bandwidth during the query (and compaction) and accelerates the total runtime noticeably. In most cases, apart from a few negligible exceptions, in-place compaction is the fastest option. This allows for larger models and eliminates the need for further expensive memory de-/allocations.

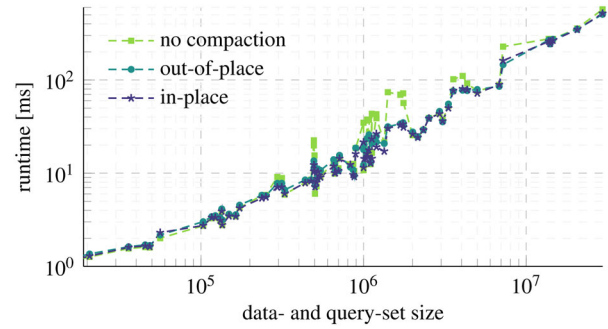


Figure 7: Runtime plot showing hierarchy construction, optimization ($\Theta = 32$), and k NN query (with $k = 16$) for each point in the input data set without memory transfer times. In-place compaction is in nearly all cases the fastest method due to increased memory-bandwidth.

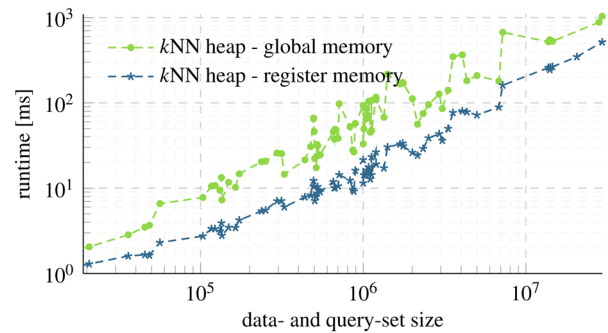


Figure 8: This plot shows the overall runtime using two different heap datastructures for the currently found k NN. Shown is total runtime (hierarchy construction, optimization and k NN query with $k = 16$) without memory transfer times. Using our register based approach delivers. By using our approach a significantly shorter runtime is achieved.

5.2. Traversal optimization analysis

In this subsection, we discuss the performance gains we achieved through our register-based heap approach, compared to a binary-heap that resides in global memory and our decision to use a backtracking traversal scheme. For this purpose, we also implemented a binary heap that resides in global device memory.

Register based k NN-Heap To demonstrate the advantage of sorting and storing the currently found k NN in register memory instead of global device memory, we compared the runtime of both approaches. The results are shown in Figure 8. As before, we compare the total runtime: the sum of hierarchy construction, optimization and a full k NN query ($k = 16$) for each point without any memory transfer times. The massive advantage of using device register memory for neighbour search only is evident. **Traversal-scheme** Heap-based traversal strategies are typically used on CPUs, since they expand and examine fewer nodes overall. This generally leads to a faster runtime compared to backtracking approaches. On the GPU, however, due to its possible size, this node heap must be stored and sorted in slow global device memory, which leads to random mem-

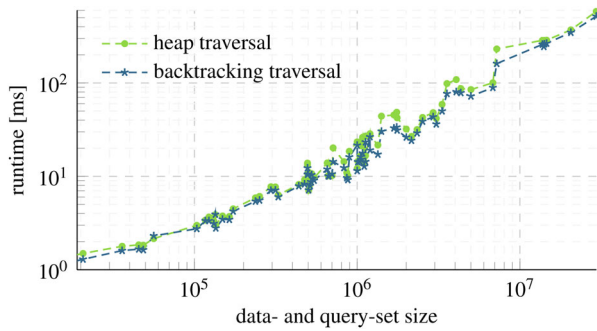


Figure 9: Runtime plot showing different traversal strategies. Shown is total runtime (hierarchy construction, optimization and kNN query with $k = 16$ without memory transfer times). The backtracking strategy is faster in nearly all cases using our proposed optimized hierarchy and register heap.

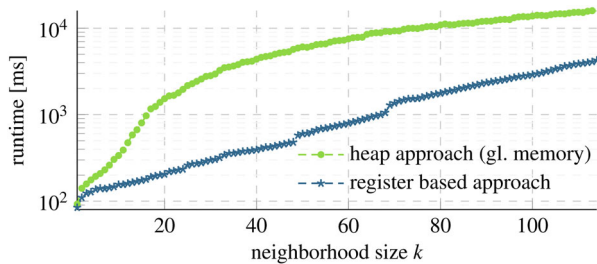


Figure 10: Algorithm runtime comparison of our register based approach vs. a binary-heap in global memory with increasing neighbourhood size. The used point cloud is a very inhomogeneously distributed large scale city scan (Zagreb001) from the Robotics 3D Scan Repository. To make the large queries possible, the query-vector was split into up to four parts, such that point cloud, query and results could fit entirely on the GPU. Measurements do not include memory transfer times.

ory accesses, low bandwidth and thus longer runtime. We therefore use a backtracking approach. Figure 9 shows that the overhead of additional expanded nodes can be compensated for by the lower memory latency. The decision for a backtracking approach thus delivers a faster result in almost all cases.

5.3. Register limit analysis

Two main components limit the scalability of our approach:

- The available number of registers is limited (amount differs from GPU to GPU). If more registers are requested than available the compiler spills addresses to global and slow GPU main memory.
- The number of compare-and-swap (CAS) operations increases linearly with the size of the requested neighbourhood (k), which generates a significant compute overhead with large k .

Therefore we compared our register-based method with a heap-based approach using global memory. Figure 10 shows the query runtimes of both methods with increasing k . As benchmark point

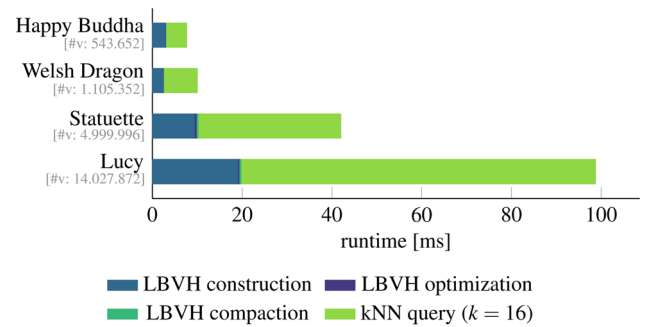


Figure 11: Algorithm runtime break down. Shown are all parts of LBVH construction and kNN-Query (with $k = 16$) on four different sized pointclouds without memory transfer times. Our proposed LBVH optimization and compaction step occupy are lightning fast and only needs a tiny fraction of the overall runtime.

cloud we used a very inhomogeneously distributed large scale city scan. It can be clearly seen that our algorithm runtime approaches more and more the heap-based method with increasing neighbourhood size. However, the runtime difference remains significant. This is due to the fact that each insert operation of the heap leads to many global memory accesses and incoherent thread loads. The small runtime bumps, occurring at $k = 24, 49$ and 69 also happen with other data-sets. This indicates a significantly different emitted kernel code. We cannot provide measurements beyond a neighbourhood size of 114, as the kernels did not provide correct results on our graphics card with a neighbourhood larger than that, because of local memory corruptions. This indicates that the compiler cannot allocate enough memory and/or register addresses and represents a hard upper limit of our approach.

5.4. Runtime break down

In Figure 11a runtime break down of our approach without any memory transfer times is shown. As usual a kNN-query with $k = 16$ (for each point in the input cloud) was used on four different point clouds from the benchmark set. Our proposed LBVH optimization takes only a fraction of the total runtime. For a data set the size of 14 million points (Lucy from the Stanford 3D Repository), the LBVH can be constructed and optimized in less than 18 milliseconds. A kNN query with a neighbourhood search size of $k = 16$ for each of the 14 million points thus takes 79 milliseconds on average. This corresponds to a query performance of over 10^5 answered neighbourhood queries per millisecond. Downloading the results of this query alone would take longer than the kernel needs for the calculation. For a runtime break down including all memory transfer times, we refer the reader to the supplemental material.

5.5. Kernel Performance Analysis

The main performance limiting factors of k nearest neighbour calculation using trees on a GPU are SIMD efficiency and latency caused by memory access and instruction dependencies. We profiled our query kernel on Apetrei's [Ape14] as well as our proposed BVH, using different combinations of traversal strategies and a global as

Table 1: Latency and cache efficiency analysis of the query-kernel using a real-world data set (3D laser scan). Shown are different query runs ($k = 16$) with different combinations of BVH, traversal-scheme and k NN-heap. (Obtained with NVIDIA Nsight Compute).

Method	IPC	L1	L1 + L2	fetch	mem.	dep.
[Ape14] + hp.trav. + glb.heap	0.15	47.9%	75.3%	0.6%	87.6%	1.5%
our BVH + hp.trav. + glb.heap	0.23	63.7%	90.2%	0.9%	86.3%	2.3%
our BVH + hp.trav. + reg.heap	1.80	70.9%	93.4%	5.0%	62.5%	7.9%
our BVH + backtr. + reg.heap	2.82	70.3%	99.0%	8.5%	31.9%	12.8%

well as the proposed register-based k NN-heap. Table 1 shows the number of executed instructions per clock cycle, cache efficiency, and the warp-stall reasons.

The analysis clearly shows that each of our optimization decisions contributes to the overall performance improvement, since the issued instructions per clock cycle (IPC) increase significantly. This is especially evident when switching from the global memory to the register-based k NN-heap. Using the optimized BVH the cache efficiency increases massively. In combination with the register-heap and a backtracking strategy the warp-stall reasons shift from memory- to instruction-dependency. Random memory access and memory latency are still very present but significantly decreased. This is only a logical consequence of the way data is now processed. It is mainly fetched from registers and stored again after the instruction has been executed. Our analysis also shows, that the kernel usage and memory bandwidth changes from 5% and 40%, respectively, for Apetreis BVH with heap-traversal and a global k NN-heap to about 70% and 35% using our proposed approach. The massive increase of compute-usage at almost the same memory bandwidth is our main source of performance improvement.

5.6. Comparison

We compared our approach to different existing and publicly available methods and also included one *approximate* algorithm (GPULSH) to show the high query performance of our exact approach. In the following, we give a brief description of the used reference algorithms and the used parameters or adjustments:

FAISS is a library for efficient similarity search and clustering of high dimensional dense vectors, that also delivers GPU implementations as drop-in replacements for their CPU equivalents [JDJ17]. Because we want exact results, and no data set training, we build and use the *IndexFlatL2* entirely on the GPU that only performs brute-force L2 distance search.

KNNCUDA by Garcia *et al.* [GDB08] is a k NN brute force technique for high-dimensional feature-data, based on distance matrix calculation and an optimized insertion sort method. The *knm_cuda_global*-kernel was used, as it was the fastest of the three available implementations. Also, due to the massive GPU memory requirements, we had to split the query vector into several parts depending on its size, so that the entire query could be processed.

ExactCUDAKNN proposed by Klusek *et al.* [KD18], represents another brute force approach, computing the squared distance matrix with a subsequent sort operation to extract the k NN.

BFKNN is the fourth brute force method to compute the k NNs on a GPU [LA15]. Similar to KNNCUDA, we had to split the query vector into several parts to be able to fit everything into GPU main memory.

GPUFLANN [ML09] is a nearest neighbour library for high- and low-dimensional data also featuring a GPU implementation that was written by Andreas Mützel. Is based on a top-down constructed k-d tree and a heap-based traversal.

GPULSH presents a GPU-based locality sensitive hashing (LSH) algorithm, to perform an *approximate* k NN search in high dimensional spaces. The used data-structure of Pan *et al.* [PM11] avoids expensive operations like sort and attempts to reduce the search space by partitioning it into several groups using LSH.

All methods were tested with the same benchmark data set, shown in Figure 4. To ensure a fair comparison, the runtime was averaged over 10 iterations including a preceding device warm-up (to disable a possible GPU power save state). For all methods the total algorithm runtimes were measured, including memory transfer times, but *without* IO-timings, which required adjustments for some methods.

Additionally, we validated our results with a brute force approach on the CPU. Our implementation yields identical results except in cases in which adjacent distances are equal and the order is undefined. That is, the returned indices are sorted in the increasing order of the corresponding distances. This also applies to the radius search.

5.6.1. k NN-Search runtime comparison

In the following, we compare the runtimes for the k NN search of our approach for different neighbourhood sizes over a wide range of highly diverse point clouds with the aforementioned reference algorithms. For each single point in each point cloud the k nearest neighbours are searched. Figure 12 shows the achieved runtimes. All brute-force algorithms are in some cases more than one order of magnitude slower than the approximate approach (GPULSH), as well as both spatial subdividing methods, which include GPU-FLANN and our approach which provides the fastest results. Our method is mostly independent of data set characteristics and thus point distribution, respectively point density. We achieve an almost linear runtime behaviour depending on the input data set. Compared to the fastest reference algorithm *GPUFLANN* we achieve an average speedup of about 3.3. For a detailed speedup analysis we refer the reader to the supplemental material.

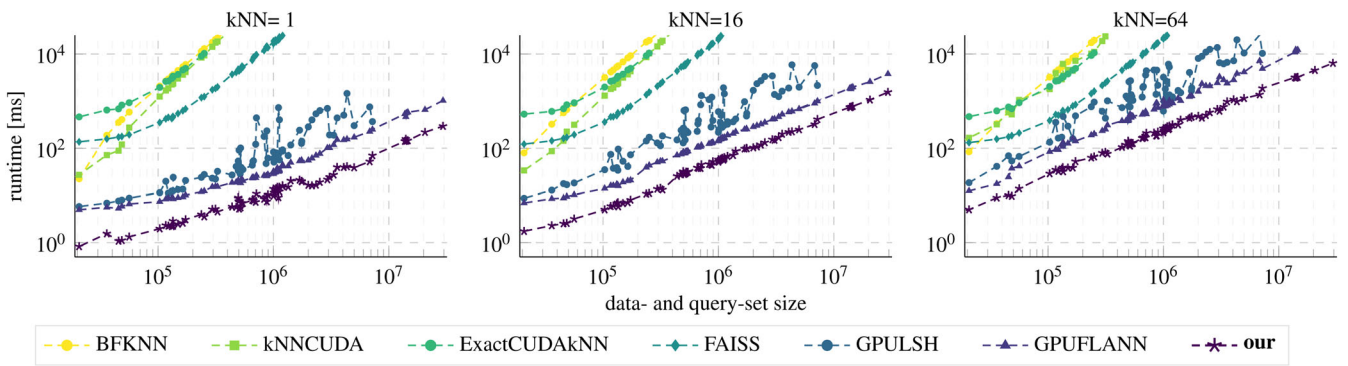


Figure 12: Algorithm runtimes of all state-of-the-art reference algorithms in comparison to our approach including memory transfer. Each algorithm was configured to calculate the $k = 1, 16,$ and 64 exact nearest neighbours of each point in the input data set. Except for GPULSH, which only provides approximate results! GPUFLANN and GPULSH were not able to process large point clouds with large queries, leading to less measuring points. Our approach is on average 3.3 times faster than the fastest reference algorithm: GPUFLANN.

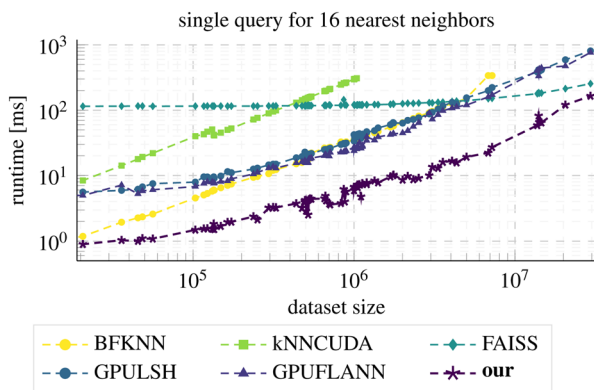


Figure 13: Algorithm runtimes (including memory transfer) of a single query ($k = 16$) in each point cloud. Even in this (very brute-force advantageous) scenario, our method is on average four times faster than all reference algorithms. Some algorithms could not provide any or incomplete measurements due to internal errors or memory limitations.

5.6.2. Query performance

Many parallelized brute-force methods are well suited to quickly answer a few k NN queries in a huge data set. Figure 13 shows the advantage of our method over existing ones even for answering just a single k NN query.

Despite the required LBVH construction and optimization beforehand, our approach delivers a significant speedup of 4.3 on average. *ExactCUDAkNN* was not able to answer a single query and the reference algorithms *kNNCUDA* and *BFKNN* could not provide valid results for all data sets due to internal errors or memory restrictions.

5.6.3. Radius-search runtime comparison

Another very important query for point cloud driven algorithms is to get all closest neighbours of a point within a given radius. We

therefore measured the total algorithm runtime to find all nearest neighbours within a search-radius of 0.5%, 1%, and 5% of the current point cloud bounding box extent. As only *GPUFLANN* and our implementation offer the possibility to launch a *radius-search*, we configured both algorithms for a maximum neighbourhood count of 64 for a fair comparison. In Figure 14 the achieved runtimes are plotted. Again, our approach consistently delivers a faster runtime over the whole benchmark set and achieves an average speedup of about 3.3, similar to the k NN query. For a detailed speedup analysis we refer the reader to the supplemental material once more.

6. Further Tests

In our attempts to further improve our approach, we have tested and implemented additional methods, which have not led to any acceleration. Two important methods are discussed in more detail below.

6.1. Hilbert codes

As already stated in Section 4.1, we use Morton keys to sort and organize the input data. Instead of Morton keys we also tested Hilbert codes, as they have a better memory locality of spatially close data [MJFS01]. However, this only led to a minimal runtime improvement during the query phase, which was eliminated by the additional overhead of the Hilbert code calculation. The bottleneck here is not the memory access to the raw point-cloud data itself, but the randomized and highly divergent access to the traversed tree data structure.

6.2. Tree optimizations

For this reason we tested further LBVH algorithms, which delivered comparable hierarchy construction times but showed a different storage pattern [Kar12] or methods that perform a tree optimization after initial construction [KA13] and [DP15]. However, none of the tested methods achieved a noticeable acceleration either, since each further tree optimization consumed the previously gained time in the later query phase.

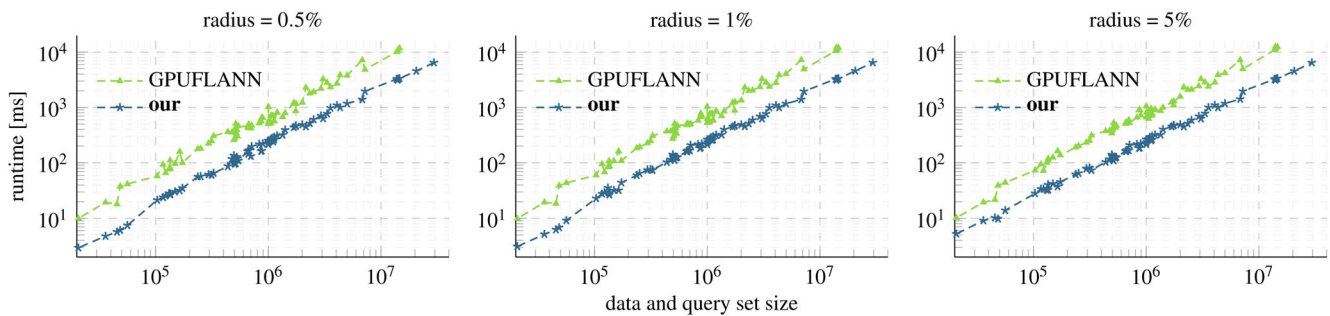


Figure 14: Runtime comparison of a radius-query of our approach compared to GPUFLANN [ML09]. All other reference methods do not offer a radius query. As search radius we have chosen 0.5%, 1% and 5% of the bounding box diagonal of each data set. For the times shown here, all neighbours of each point in each input data set are searched. Both our approach and GPUFLANN are configured to return a maximum of 64 (sorted) nearest neighbours within the search radius. Our method is on average over three times faster than GPUFLANN. As with the k NN query, GPUFLANN is not able to process large point clouds with large queries, so there is little measurement data missing.

6.3. Overlapping Kernel/Memcpy

We also experimented with overlapping query computation and result transfer. But to allow asynchronous memory transfer, pinned host memory must be allocated. In all of our experiments the massive overhead of allocating pinned memory on our test machine was higher than the achieved speedup during the query. However, with a different operating system or a more advanced hardware configuration this could change in the future, and thus bring a notable performance boost.

7. Conclusion

We have proposed an optimized massively parallel method for extremely fast k nearest neighbour search in point clouds. We proposed a new technique to optimize an existing LBVH and to accelerate the required k NN-heap during the query phase by using a heap, which is directly implemented on hardware registers instead of using slow off-chip memory.

In order to demonstrate the effectiveness of our optimizations, we performed multiple benchmarks on a data set from very different point clouds and created by different acquisition modalities, to obtain an analysis that was as realistic and diversified as possible. Compared to other state-of-the-art algorithms, our benchmarks show that the proposed method is on average about 3.3 times faster than the fastest reference algorithm, and thus offers almost real-time performance even with extremely large data sets.

Due to the extremely fast hierarchy construction and its optimization, our method is also suitable for dynamic scenes, simulations or real-time ICP. It is also robust against extremely inhomogeneously distributed point sets and can easily handle large neighbourhood queries. In addition, it is possible to optimize our approach to specific point cloud properties using the available parameters.

Due to its robustness and high query performance, our approach represents an important advance in many areas where algorithms rely on fast k NN or radius queries.

One of the main limitations of our presented work is the restricted data set size due to the available GPU memory. This could be elim-

inated by using out-of-core streaming techniques, which we want to address in future research. Another disadvantage is the restricted neighbourhood size due to the number of existing device registers, which depends on the GPU used.

A possible future project is to develop a better collapse heuristic, which leads to even more homogeneous hierarchies, and thus faster queries.

Acknowledgements

We thank the respective authors for providing all point-clouds in our benchmark-dataset, for which we used data-sets provided by AIM@SHAPE-VISIONAIR Shape Repository, Stanford 3D Scanning Repository, Digital Michelangelo Project, CyberwareInc, Bangor University, the Robotics 3D Scan Repository, Nefertiti Hack (nefertitihack.alloversky.com/), Georgia Institute of Technology and Oliver Laric (threedscans.com) as also LaserDesign (laserdesign.com) and finally 3D Reality Maps. The source-code of **BFKNN** is publicly available at <https://github.com/geomlab-ucd/bf-knn>. The source-code of **kNNCUDA** is licensed under the Attribution-NonCommercial-ShareAlike 3.0 License and is publicly available at <https://github.com/vincentfpgarcia/kNN-CUDA>. The source-code of **ExactCUDAKNN** is licensed under the MIT License and is publicly available at <https://github.com/gosteq/Exact-CUDA-knn>. The source-code of **FAISS** is licensed under the MIT License and is publicly available at <https://github.com/facebookresearch/faiss>. The source-code of **GPULSH** is publicly available at <http://gamma.cs.unc.edu/KNN/>. The source-code of **GPUFLANN** is licensed under the BSD License and is publicly available at github.com/mariusmuja/flann. This research was funded by the Bayerische Forschungstiftung under the project *For3D*.

Open access funding enabled and organized by Projekt DEAL.

References

[AMN*98] ARYA S., MOUNT D. M., NETANYAHU N. S., SILVERMAN R., WU A. Y.: An optimal algorithm for approximate nearest

- neighbor searching fixed dimensions. *Journal of the ACM* 45, 6 (1998), 891–923.
- [Ape14] APETREI C.: Fast and simple agglomerative LBVH construction. In *Theory and Practice of Computer Graphics, Leeds, United Kingdom, 2014. Proceedings (2014)*, Borgo R., Tang W., (Eds.), Eurographics Association, Darmstadt, pp. 41–44.
- [ARBM12] AREFIN A. S., RIVEROS C., BERRETTA R., MOSCATO P.: Gpu-fs-knn: A software tool for fast and scalable knn computation using gpus. *PLOS ONE* 7, 8 (08 2012), 1–13.
- [BDHK06] BUSTOS B., DEUSSEN O., HILLER S., KEIM D.: A graphics hardware accelerated algorithm for nearest neighbor search. In *Computational Science – International Conference on Computational Science 2006* (Berlin, Heidelberg, 2006), Alexandrov V. N., van Albada G. D., Sloot P. M. A., Dongarra J., (Eds.), Springer, Berlin Heidelberg, pp. 196–199.
- [Ben75] BENTLEY J. L.: Multidimensional binary search trees used for associative searching. *Communications of the ACM* 18, 9 (1975), 509–517.
- [BGT*11] BARRIENTOS R., GÓMEZ J., TENLLADO C., PRIETO M., MARIN M.: knn query processing in metric spaces using gpus. In *17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011)* (Bordeaux, France, September 2011), vol. 6852 of *Lecture Notes in Computer Science*, Springer, pp. 380–392.
- [BGTP10] BARRIENTOS R., GÓMEZ J., TENLLADO C., PRIETO M.: Heap-based k-nearest neighbor search on gpus. In *XXI Jornadas de Paralelismo (JP 2010)* (Septiembre 2010), pp. 559–566.
- [BM92] BESL P. J., MCKAY N. D.: A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14, 2 (1992), 239–256. .
- [DKMD13] DASHTI A., KOMAROV I., M D’SOUZA R.: Efficient computation of k-nearest neighbour graphs for large high-dimensional data sets on GPU clusters. *PLoS ONE* 8 (2013), e74113.
- [DMG10] DERZAPF E., MENZEL N., GUTHE M.: Parallel View-dependent refinement of compact progressive meshes. In *Eurographics Symposium on Parallel Graphics and Visualization (2010)*, AHRENS J., DEBATTISTA K., PAJAROLA R., (Eds.), Eurographics Association, Darmstadt.
- [DP15] DOMINGUES L. R., PEDRINI H.: Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proceedings of the 7th Conference on High-Performance Graphics* (New York, NY, USA, 2015), HPG ’15, ACM, pp. 13–20.
- [FC16] FU C., CAI D.: EFANNA : An extremely fast approximate nearest neighbor search algorithm based on knn graph. *CoRR abs/1609.07228* (2016). <http://arxiv.org/abs/1609.07228>.
- [GDB08] GARCIA V., DEBREUVE E., BARLAUD M.: Fast k nearest neighbor search using gpu. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops* (June 2008), pp. 1–6. <https://doi.org/10.1109/CVPRW.2008.4563100>.
- [GGG08] GUENNEBAUD G., GERMANN M., GROSS M.: Dynamic sampling and rendering of algebraic point set surfaces. *Computer Graphics Forum* 27, 2 (2008), 653–662.
- [GJM77] GINGOLD R., J. MONAGHAN J.: Smoothed particle hydrodynamics - theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society* 181 (11 1977), 375–389.
- [HDW*11] HAPALA M., DAVIDOVIČ T., WALD I., HAVRAN V., SLUSALLEK P.: Efficient stack-less bvh traversal for ray tracing. *SCCG ’11, Association for Computing Machinery*, pp. 7–12. <https://doi.org/10.1145/2461217.2461219>.
- [HJ10] HACHISUKA T., JENSEN H. W.: Parallel progressive photon mapping on GPUs. In *ACM SIGGRAPH ASIA 2010 Sketches (2010)*, SA ’10, ACM, pp. 54:1–54:1. <https://doi.org/10.1145/1899950.1900004>.
- [JDJ17] JOHNSON J., DOUZE M., JÉGOU H.: Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734* (2017).
- [KA13] KARRAS T., AILA T.: Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG ’13, ACM, pp. 89–99. <https://doi.org/10.1145/2492045.2492055>.
- [Kar12] KARRAS T.: 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* (Goslar Germany, Germany, 2012), EGGH-HPG’12, Eurographics Association, Darmstadt, pp. 33–37. <https://doi.org/10.2312/EGGH/HPG12/033-037>.
- [KD18] KŁUSEK A., DZWINEL W.: Multi-gpu k-nearest neighbor search in the context of data embedding. *Advances in Parallel Computing* 32 (2018), 359–368. .
- [KH10] KATO K., HOSINO T.: Solving k-nearest neighbor problem on multiple graphics processors. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (2010)*, pp. 769–773. <https://doi.org/10.1109/CCGRID.2010.47>.
- [KH12] KATO K., HOSINO T.: Multi-GPU algorithm for k-nearest neighbor problem. *Concurrency and Computation: Practice and Experience* 24, 1 (01 2012), 45–53.
- [KZ09] KUANG Q., ZHAO L.: A practical gpu based knn algorithm. In *In Proceedings of the Second Symposium on International Computer Science and Computational Technology (ISCST ’09) (Dec. 2009)*, Academy Publisher (01 2009), pp. 151–155.
- [LA15] LI S., AMENTA N.: Brute-force k-nearest neighbors search on the gpu. In *Proceedings of the 8th International Conference on Similarity Search and Applications - Volume 9371* (Berlin, Heidelberg, 2015), SISAP 2015, Springer-Verlag, Berlin, pp. 259–270.

- [LLWJ09] LIANG S., LIU Y., WANG C., JIAN L.: A cuda-based parallel implementation of k-nearest neighbor algorithm. In *2009 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery* (Oct 2009), pp. 291–296. <https://doi.org/10.1109/CYBERC.2009.5399145>.
- [LLWJ10] LIANG S., LIU Y., WANG C., JIAN L.: Design and evaluation of a parallel k-nearest neighbor algorithm on cuda-enabled gpu. In *2010 IEEE 2nd Symposium on Web Society* (Aug 2010), pp. 53–60. <https://doi.org/10.1109/SWS.2010.5607480>.
- [LSP*12] LI S., SIMONS L., PAKARAVOOR J. B., ABBASINEJAD F., OWENS J. D., AMENTA N.: kANN on the GPU with Shifted Sorting. In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics* (2012), Dachsbacher C., Munkberg J., Pantaleoni J., (Eds.), Eurographics Association, Darmstadt. <https://doi.org/10.2312/EGGH/HPG12/039-047>.
- [LTF*09] LEITE P. J. S., TEIXEIRA J. M. X. N., FARIAS T. S. M. C. d., TEICHRIEB V., KELNER J.: Massively parallel nearest neighbor queries for dynamic point clouds on the GPU. In *2009 21st International Symposium on Computer Architecture and High Performance Computing* (2009), pp. 19–25. <https://doi.org/10.1109/SBAC-PAD.2009.18>.
- [LWLJ09] LIANG S., WANG C., LIU Y., JIAN L.: CUKNN: A parallel implementation of k-nearest neighbor on CUDA-enabled GPU. In *Computing and Telecommunication 2009 IEEE Youth Conference on Information* (Sept 2009), pp. 415–418. <https://doi.org/10.1109/YCICT.2009.5382329>.
- [LŽ15] LUKAČ N., ŽALIK B.: *Fast Approximate k-Nearest Neighbours Search Using GPGPU*. Springer Singapore, Singapore, 2015, pp. 221–234. https://doi.org/10.1007/978-981-287-134-3_14.
- [MB17] MEISTER D., BITTNER J.: Parallel locally-ordered clustering for bounding volume hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics PP* (02 2017), 1–1. <https://doi.org/10.1109/TVCG.2017.2669983>.
- [MJFS01] MOON B., JAGADISH H. V., FALOUTSOS C., SALTZ J. H.: Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering* 13, 1 (2001), 124–141.
- [ML09] MUJA M., LOWE D. G.: Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP (1)* (2009), Ranchordas A., Araújo H., (Eds.), INSTICC Press, Setubal, Portugal, pp. 331–340.
- [MLM13] MARA M., LUEBKE D., MCGUIRE M.: Toward practical real-time photon mapping: Efficient gpu density estimation. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2013), I3D '13, Association for Computing Machinery, p. 71–78. <https://doi.org/10.1145/2448196.2448207>.
- [PCS15] POMERLEAU F., COLAS F., SIEGWART R.: A review of point cloud registration algorithms for mobile robotics. *Found. Trends Robot* 4, 1 (May 2015), 1–104. <https://doi.org/10.1561/2300000035>.
- [PDC*03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Goslar, DEU, 2003), HWWS '03, Eurographics Association, Darmstadt. pp. 41–50.
- [PLM10] PAN J., LAUTERBACH C., MANOCHA D.: Efficient nearest-neighbor computation for GPU-based motion planning. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2010), pp. 2243–2248. <https://doi.org/10.1109/IROS.2010.5651449>.
- [PM11] PAN J., MANOCHA D.: Fast GPU-based locality sensitive hashing for k-nearest neighbor computation. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (2011), GIS '11, ACM, pp. 211–220. <https://doi.org/10.1145/2093973.2094002>.
- [PM12] PAN J., MANOCHA D.: Bi-level locality sensitive hashing for k-nearest neighbor computation. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering* (USA, 2012), ICDE '12, IEEE Computer Society, pp. 378–389. <https://doi.org/10.1109/ICDE.2012.40>.
- [QMN09] QIU D., MAY S., NÜCHTER A.: GPU-accelerated nearest neighbor search for 3d registration. In *Computer Vision Systems* (2009), Fritz M., Schiele B., Piater J. H., (Eds.), Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 194–203.
- [SBMN16] SCHAUER J., BEDKOWSKI J., MAJEK K., NÜCHTER A.: Performance comparison between state-of-the-art point-cloud based collision detection approaches on the cpu and gpu. *IFAC-PapersOnLine* 49, 30 (2016), 54–59. 4th IFAC Symposium on Telematics Applications TA 2016. <https://doi.org/10.1016/j.ifacol.2016.11.125>.
- [SPS12] SISMANIS N., PITSIANIS N., SUN X.: Parallel search of k-nearest neighbors with synchronous operations. In *2012 IEEE Conference on High Performance Extreme Computing* (2012), pp. 1–6. <https://doi.org/10.1109/HPEC.2012.6408667>.
- [THE*15] TANG X., HUANG Z., EYERS D., MILLS S., GUO M.: Efficient selection algorithm for fast k-NN search on GPUs. In *2015 IEEE International Parallel and Distributed Processing Symposium* (2015), pp. 397–406. <https://doi.org/10.1109/IPDPS.2015.115>.
- [WWSHL16] WIESCHOLLEK P., WANG O., SORKINE-HORNUNG A., LENSCH H. P. A.: Efficient large-scale approximate nearest neighbor search on the GPU. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 2027–2035. <https://doi.org/10.1109/CVPR.2016.223>.
- [ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-time KD-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 Papers* (2008), SIGGRAPH Asia '08, ACM, pp. 126:1–126:11. <https://doi.org/10.1145/1457515.1409079>.

Supporting Information

Additional supporting information may be found online in the Supporting Information section at the end of the article.

Figure 1: Overall runtime of our approach using different graphics cards.

Figure 2: Total algorithm runtime break-down.

Figure 3: Algorithm runtimes of the fastest reference algorithm GPUFLANN vs.

Figure 4: Algorithm runtimes of the fastest reference algorithm GPUFLANN vs.