QuickNN: Memory and Performance Optimization of k-d Tree Based Nearest Neighbor Search for 3D Point Clouds

Reid Pinkham
University of Michigan
pinkhamr@umich.edu

Shuqing Zeng
General Motors
shuqing.zeng@gm.com

Zhengya Zhang
University of Michigan
zhengya@umich.edu

ABSTRACT

The use of Light Detection And Ranging (LiDAR) has enabled the continued improvement in accuracy and performance of autonomous navigation. The latest applications require LiDAR's of the highest spatial resolution, which generate a massive amount of 3D point clouds that need to be processed in real time. In this work, we investigate the architecture design for k-Nearest Neighbor (kNN) search, an important processing kernel for 3D point clouds. An approximate kNN search based on a k-dimensional (k-d) tree is employed to improve performance. However, even for today's moderate-sized problems, this approximate kNN search is severely hindered by memory bandwidth due to numerous random accesses and minimal data reuse opportunities. We apply several memory optimization schemes to alleviate the bandwidth bottleneck: 1) the k-d tree data structure is partitioned to two sets: tree nodes and point buckets, based on their distinct characteristics – tree nodes that have high reuse are cached for their lifetime to facilitate search, while point buckets with low reuse are organized in regular contiguous segments in external memory to facilitate efficient burst access; 2) write and read caches are added to gather random accesses to transform them to sequential accesses; and 3) tree construction and tree search are interleaved to cut redundant access streams. With optimized memory bandwidth, the kNN search can be further accelerated by two new processing schemes: 1) parallel tree traversal that utilizes multiple workers with minimal tree duplication overhead, and 2) incremental tree building that minimizes the overhead of tree construction by dynamically updating the tree instead of building it from scratch every time. We demonstrate the performance and memory-optimized QuickNN architecture on FPGA and perform exhaustive benchmarking, showing that up to a 19× and 7.3× speedup over k-d tree searches performed on a modern CPU and GPU, respectively, and a 14.5× speedup over a comparable sized architecture performing an exact search. Finally, we show that QuickNN achieves two orders of magnitude performance per watt increase over CPU and GPU methods.

1. INTRODUCTION

A point cloud [1] refers to a collection of data points in space and is often obtained by 3D scanning, e.g., by a 3D laser scanner or Light Detection And Ranging (LiDAR) [2, 3]. Autonomous navigation has increasingly relied on point clouds collected by rotating LiDAR scanning to perceive the environment and build an internal model of the environment [4, 5, 6, 7]. A sample point cloud is rendered in Figure 1, showing the points scattered in the 3D space indicating the obstacle locations. A typical point cloud for autonomous navigation consists of upwards of 100k points and a new frame is generated every 1/10th of a second [8], posing a significant processing workload. The trend towards even higher precision in newer generation of applications requires that the resolution, size, and frame rate of point clouds will continue to increase.

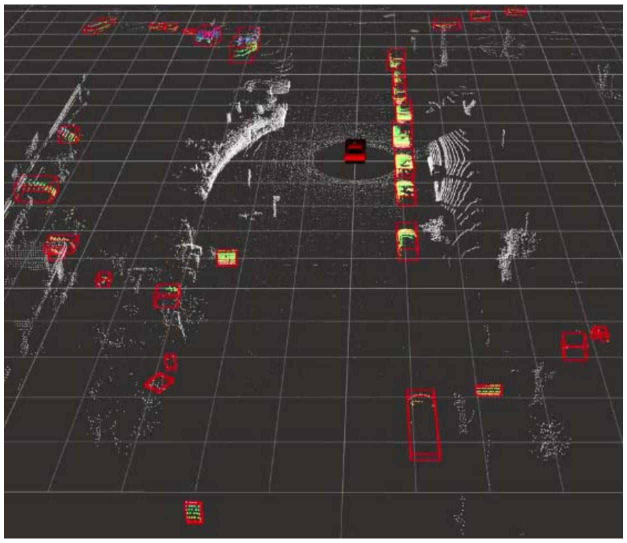


Figure 1: An example point-cloud frame from the KITTI dataset [9] with bounding boxes of the detected object.

Performing a k-Nearest Neighbor (kNN) search is a kernel step in point cloud processing, e.g., object detection [10, 11, 12], localization [13], path planning [14], and hazard detection [15]. In autonomous navigation tasks perceiving the dynamics of moving objects in the environment and estimating their relative position are crucial enablers. Iterative closest point (ICP) [16, 17] is the prevailing method to estimate the motion and surface based on point cloud where kNN is the key step to identify the closest neighborhood for the points. A benchmark of an ICP-based method [11] reveals that 75% of the ICP is spending on kNN search. In these tasks, each point from a query set must be matched with the *k* nearest points in a reference set. Often, the two sets chosen are two complete 3D point clouds. The kNN search always

2378-203X/20/$31.00 ©2020 IEEE
DOI 10.1109/HPCA47549.2020.00024

lies in the inner loop and the critical path of the perception and planning pipeline. It is pertinent that the latency of the kNN computation is kept minimal since the system is expected to react to the environment in a fraction of a second. The need for real time computation is apparent.

The most common solution is to perform the kNN search on a high-performance multi-core CPU. Unfortunately, because of the large data size of point clouds and memory-intensive nature of the kNN algorithm, this solution struggles to meet the stringent throughput and latency requirement [18]. GPU is another common solution. A high-end GPU provides the necessary memory bandwidth to meet the latency requirement, but the irregularity of point cloud data and the data structures needed to perform a kNN search pose a challenge for the efficient use of the memory bandwidth [19]. These challenges call for a more optimized architecture for kNN search on 3D point clouds. The architecture can be implemented as a hardware IP to accelerate the kNN search with the most efficient use of memory bandwidth.

In designing the kNN architecture for point clouds, we choose the target application of 3D LiDAR processing, and use the KITTI dataset [9] for evaluation. Additionally, we use the Ford Campus Vision and Lidar Data Set [20] to verify results. kNN search is applied to successive frames of 3D point clouds to estimate frame-to-frame movement. This successive-frame kNN search is used to differentiate the surroundings from moving objects. Since the successive-frame kNN search demands the largest possible amount of computation and data, the benchmarking gives the lower bound on the performance and bandwidth usage for all the other use cases. A typical LiDAR 3D point cloud frame can contain upwards of 100k points. Because ground points do not contribute information in this use case, it is common practice [21] to remove many of these points using a ground threshold as part of the pre-processing step, resulting in roughly 30k useful points. A kNN search is performed on these 30k points for benchmarking.

In Section 2, we provide an overview of kNN algorithms. In Section 3, we establish a linear kNN search as the baseline. In Section 4, we present the memory and performance optimization techniques in designing a QuickNN architecture that outperforms the baseline. In Section 6 and 7, QuickNN is benchmarked in software and prototyped on FPGA for evaluation and comparison with CPU and GPU implementations. Section 8 concludes this work.

2. KNN SEARCH METHODS

There are two classes of kNN search methods: exact and approximate. The exact methods find the exact k nearest neighbors. However, in most of the practical applications, including object tracking, kNN search is enclosed in an ICP [22, 23, 24] loop to estimate the transformation between object model the target frame. Iterations provide error tolerance, and an approximate kNN search can be applied [11]. Table 1 provides a qualitative overview of popular kNN search methods. We chose the approximate k-d tree method for our design due its good trade-off between complexity and accuracy.

[1]Accuracy for 30k points, 8 nearest neighbors

Table 1: Comparison of Popular kNN Methods

	Accuracy[1]	Search Complexity	Mem Reads
Linear	100%	N^2	N^2
Approx. k-means	99%	$N\log N$	$N\log N$
Approx. k-d Tree	91%	$N\log N$	$N\log N$
Approx. LSH	18.4%	$N\log N$	N

2.1 Exact Linear kNN Search

The linear kNN search [25] is the direct exact search method. It finds the closest points by calculating and comparing the distance between the query point and each point in the reference frame to find the k closest points. For the successive-frame use case, assume N points in both the reference frame and the query frame, the linear method requires $O(N^2)$ distance calculations and comparisons. Since frames of point clouds are often too large to fit on chip, the points are stored in external memory, and $O(N^2)$ point reads from external memory are needed for each frame.

It is straightforward to parallelize the linear method. A query point can be compared with many reference points in parallel; and multiple query points can be searched in parallel.

2.2 Approximate k-d Tree Search

The linear method performs many unnecessary comparisons. In reality, only a modest number of points in a space local to the query point need to be searched. This is what a k-d tree method [26, 27] accomplishes.

The k-d tree is a binary tree that subdivides the space of the reference frame into smaller regions or "buckets" with approximately equal number of points. These buckets are attached to the leafs of the binary tree, and each represents a local region in space. With a k-d tree, searching a query point involves first traversing the tree to find the nearest region or bucket; and then searching the bucket that is the most likely to contain the nearest neighbors.

Building a k-d tree A k-d tree is constructed for a point-cloud frame in two steps: construct the tree using a subset of representative points, and insert all the points to the buckets. An illustration of the tree construction phase is shown in Figure 2. To begin, a subset of n points from an N-point frame ($n < N$) are selected, sorted, and split into two new groups. The sorting and splitting process is repeated until either the desired depth of the tree is reached or the leafs contain a minimum occupancy of points.

Next, the entire frame of N points must be placed into the buckets. The tree is traversed from the top. At each tree node, the associated threshold is compared with the point value on that dimension to determine which branch to traverse to in the next step. The process continues until the point reaches a leaf and is placed in the bucket.

If the exact k nearest neighbors are needed, more than the nearest region may have to be searched, e.g., when the target point is very close to a region's boundary. With a so-called backtracking method, the k-d tree method becomes an exact method. However, this additional step is not always required if a small loss in accuracy is tolerable [28, 29, 30]. With approximate k-d tree search and a probabilistic ICP algorithm, the end-to-end performance of position and velocity estimates in object tracking can reach decimeter and decimeter/second,

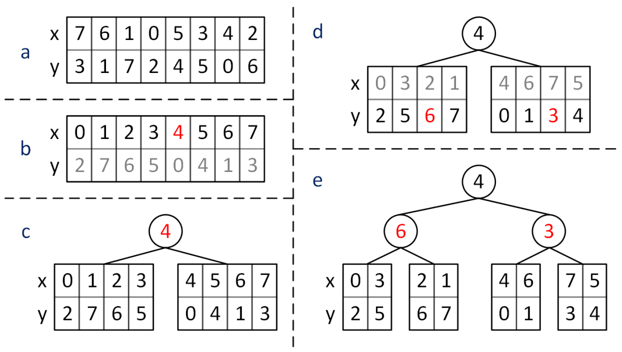


Figure 2: Example of the tree construction process. Eight points are used to construct a tree with 4 leaves. (a) The original set of 8 points, unsorted. (b) First, sorting occurs across the *x* dimension. The median is chosen, marked in red. (c) The top node is formed, with the value of 4 marking the threshold value. The points are split into two groups based on this threshold value. (d) The two new groups of points are sorted along the next dimension, *y*. Two median points are selected. (e) Two new nodes are formed with the indicated thresholds. The final result is 4 leaves each containing two points.

respectively, and 5 degrees for vehicle heading [11].

Complexity The key benefit of the k-d tree method is that it segments a large 3D space into smaller regions and thus reduces the search space and memory accesses. Assume a balanced k-d tree is built to store N points with B_N points per bucket. The depth of the tree is $d = \log_2 \frac{N}{B_N}$. A search requires d comparisons to reach the nearest bucket, and the B_N points in the bucket are searched to find the k nearest neighbors. For a query frame of N points as in the successive-frame use case, the complexity of the k-d tree search is NB_Nd, or $O(N\log(N))$, and it requires $O(NB_N)$ point reads from external memory for each frame.

Accuracy In simulation, we define accuracy as the likelihood the k nearest neighbors are present in the top $k+x$ nearest neighbors. Figure 3 shows the accuracy of the k-d tree search on the KITTI dataset for $k = 5$, $x = 0$ to 5, bucket size $B_N = 256$ to 4K points. How often the top-1 nearest neighbor is contained in the results is also shown. If we aim at 75% top-10 accuracy, the minimum bucket size B_N is 256. Clearly, the larger bucket sizes provide the better accuracy. However, the number of comparisons increases, and so does the latency of the computation.

2.3 Other Approximate Search Methods

k-means clustering [31] is another way to partition the search space. Instead of sorting and splitting along each dimension, clusters in the 3D space are identified. The process is repeated to form clusters to subdivide the partitions until the clusters reach a minimum size.

The k-means tree search has a similar complexity as the k-d tree search, but building clusters is more complex than building a k-d tree. The difference between k-d tree search and k-means tree search is examined using Fast Library for Approximate Nearest Neighbor (FLANN) [32, 33] on the KITTI dataset. The k-means tree search provides on average

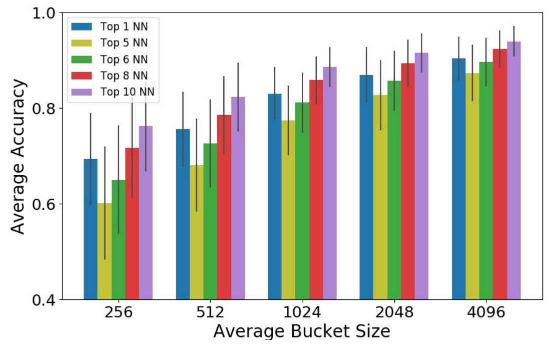


Figure 3: Accuracy of k-d tree search (k = 5) based on KITTI dataset.

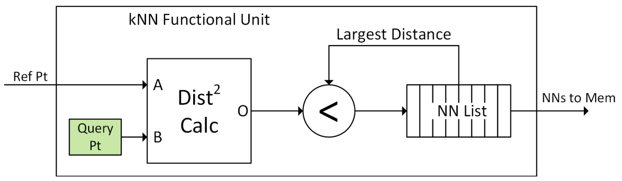


Figure 4: A Functional Unit that processes one query point and keeps a running list of k nearest neighbors.

5.6% higher accuracy than the k-d tree search, but the execution time for the k-means method is over twice that of the k-d tree method.

Simple Locality Sensitive Hashing (LSH) [34] and Multi-Probe Locality Sensitive Hashing (MPLSH) [35] are common hash based algorithms for approximate nearest neighbor search. These LSH methods use a set of hash tables and hash functions to partition points. This method was originally designed for kNN search across high-dimensional spaces. LSH addresses the curse of dimensionality problem by using fixed space partitioning by the chosen hash functions. Since our situation only requires partitioning across three dimensions, this method does not perform nearly as well as the simpler k-d tree method.

3. LINEAR SEARCH ARCHITECTURE

We present a baseline architecture for the linear search method as comparison for our k-d tree search architecture. Since the point clouds are often large and it is assumed that they are stored in external DRAM. The top level of the linear architecture contains multiple Function Units (FUs), control, and DRAM access controller.

An FU performs the distance calculation and keeps a running list of k nearest neighbor candidates. A diagram of this FU is shown in Figure 4. To start processing, query points are loaded to the FUs, one per FU. The points from the reference frame are streamed in from external memory and broadcasted to the the FUs. After all points from the reference frame stream through, an FU possesses the exact k nearest neighbors. Finally, the FUs flush the results to external memory. This process is repeated until all query points are done.

All external memory access by the linear architecture is sequential. Reading points from the reference frame follows sequential order. Points from the query frame are processed in sequential order, and the results are written back in sequential order. Sequential memory access is highly efficient. The measured memory bandwidth utilization during RTL simulation with our DRAM model is at 97%.

Two major pitfalls of this architecture are the amount of external memory accesses and the high number of operations required. A large majority of the accesses and operations are not necessary.

4. K-D TREE SEARCH ARCHITECTURE

A k-d-tree kNN processor is divided into two parts: tree builder (TBuild), responsible for tree building; and tree searcher (TSearch) to search query points to find the nearest neighbors. The processor follows three steps: 1) initial sampling: TBuild requests a subset of points from the reference frame to construct the tree; 2) point placement: TBuild places all the points from the reference frame into buckets; and 3) NN search: TSearch traverses the tree to find the target buckets, and searches the target bucket for the nearest neighbors.

4.1 Data Structure and Caching Scheme

The primary data structure is the k-d tree. TBuild creates the k-d tree and TSearch uses it. The k-d tree consists of tree nodes, and a bucket attached to each leaf node to store points. Each tree node contains a threshold, a dimension indicator, and pointers to the parent and the two child nodes. A leaf node contains a pointer to a bucket containing the points. A bucket either directly stores the points, or contains pointers to them. A point contains the $\{x,y,z\}$ coordinate in the 3D space. The straightforward way, as employed by popular software approaches [32], is to store this entire k-d tree data structure, both the tree nodes and the buckets, in DRAM.

Data Size: Assuming a balanced k-d tree that stores N points and B_N points per bucket, the tree consists of $N_t = 2\frac{N}{B_N} - 1$ nodes. For the KITTI data set, $N \approx 30k$ after removing ground points. As discussed previously, the search accuracy and search speed both depend on the bucket size. The larger the bucket, the higher the search accuracy and the slower the search. If we aim at a minimum 75% top-10 search accuracy, the bucket size, $B_N \geq 256$. For all practical purposes, $N \gg N_t$, i.e., the buckets take much more storage space than the tree nodes.

Data Reuse: For the successive-frame use case, each point in the query frame is searched against the reference frame, meaning that the tree is traversed N times. If we assume equal hit rate to each point bucket, a bucket is searched B_N times. For all practical purposes, $N \gg B_N$, i.e., the tree nodes are reused many more times than the buckets.

Data Access: With the tree nodes and the buckets stored in external DRAM, when TBuild traverses the data structure, it needs $\log_2 \frac{N}{B_N}$ random DRAM accesses to reach the leaf, and B_N random DRAM accesses to go through one bucket of points. The random access of points from DRAM constitutes the most significant delay.

The quick analyses above suggest that the tree nodes and the buckets are data of distinct characteristics: the tree nodes take much less storage and they are reused much more often. Therefore it is advantageous to cache the tree nodes on chip for its lifetime. Given a limited on-chip cache, the

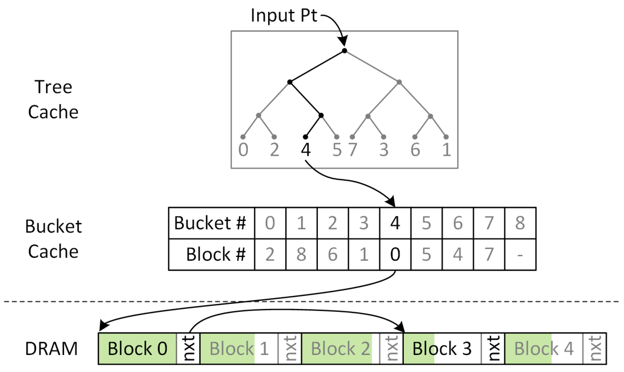

Input Pt
Tree Cache
0 2 4 5 7 3 6 1
Bucket Cache
Bucket # 0 1 2 3 4 5 6 7 8
Block # 2 8 6 1 0 5 4 7 -
DRAM
Block 0 nxt Block 1 nxt Block 2 nxt Block 3 nxt Block 4 nxt

Figure 5: The tree and bucket block map are stored in separate on-chip caches. Bucket blocks containing points are stored in external DRAM. This shows an example scheme to place a point into a bucket block.

buckets can be kept in external DRAM, but they cannot be scattered in memory due to the significant delay penalty. To remove this penalty, a bucket can be organized in a contiguous chunk to support an efficient burst access. With caching of the tree nodes, the external memory access is reduced from $O(N\log N)$ to $O(N)$, because only the buckets need to be accessed from memory.

We used the KITTI dataset with $N = 30$k as the starting point for analysis. As LiDAR resolution continues to increase, we expect the size of point cloud frames with ground points removed to grow well above 100k or even 1M in the near future. As the point density increases, we anticipate that bucket size to increase proportionally to deliver the benefits of higher resolution as the search accuracy depends on the bucket size. In such a scaled-up problem, the same arguments regarding tree and bucket size, reuse, and access still hold.

Motivated by the data characteristics, our proposed caching and memory structure is depicted in Figure 5. We use two small on-chip caches to manage the tree nodes and the bucket structure. The first cache stores the tree nodes. Each parent/child pointer is to another part of the cache. Leaf nodes in the cache point to a location in the second cache which keeps a memory map matching a bucket to a start address in DRAM. Separating these two caches allows for them to be accessed in parallel, and simplifies structuring.

A bucket block in memory is a unit of bucket storage. A block is of a fixed length and contains the points belonging to the bucket as well as a pointer to the next linked bucket block, or an end token. The bucket block size is set to be large enough to accommodate the size of a common bucket. During tree construction and point placement, if more points than what can fit in a single block are placed into a single bucket, more blocks are allocated and linked.

4.2 Memory Bandwidth Management

The kNN processor interacts with external memory through 3 read and 2 write streams, as illustrated in Figure 6. TBuild reads points from the reference frame in sequential order (Rd1), places them in buckets and writes them back in random order (Wr1). TSearch reads points from the query frame in sequential order (Rd2), reads the bucket of points in se-

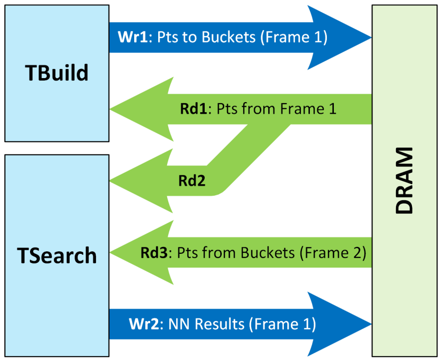

TBuild
TSearch
DRAM
Wr1: Pts to Buckets (Frame 1)
Rd1: Pts from Frame 1
Rd2
Rd3: Pts from Buckets (Frame 2)
Wr2: NN Results (Frame 1)

Figure 6: Five memory streams to and from external DRAM. Rd1 and Rd2 are combined to reduce overall memory traffic.

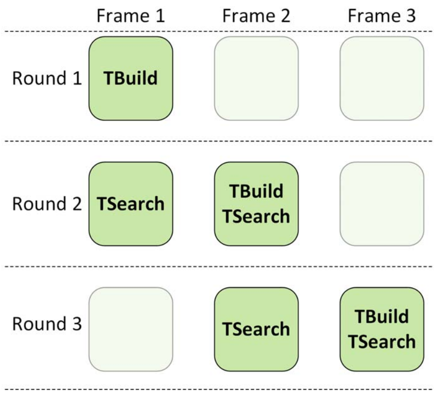


Figure 7: Rounds of computation and sharing of data frame between TBuild and TSearch.

quential order (Rd3), and writes back the nearest neighbors in sequential order (Wr2). Due to the large data size, the design of a kNN processor is easily limited by memory bandwidth.

Data Sharing: The successive-frame use case is the most demanding use of kNN search for point cloud processing. The processing is divided to rounds, as illustrated in Figure 7. In round 1, TBuild builds the tree and places frame 1 into the buckets. In round 2, TSearch uses frame 1 as the reference frame and frame 2 as the query frame to find the nearest neighbors; in the meantime, TBuild builds the next tree and places frame 2 into the buckets. Notice that frame 2 is used by both TBuild and TSearch in round 2; similarly, frame 3 is used by both TBuild and TSearch in round 3. Overlapping TBuild and TSearch cuts the processing latency, and it allows the Rd1 and Rd2 streams to be merged. We design TBuild to drive the requests to data, and allow TSearch to snoop the data. The Rd2 stream is completely eliminated.

Write-Gather Cache: TBuild reads points from external memory in sequential order, but the points are placed into a random order of buckets. Due to the random write addresses, Wr1 is highly inefficient. Our solution is to add a small write-gather cache to group points destined for the same bucket before being sent to external memory. The write-gather cache is designed to temporarily store w_b buckets of up to w_n points each. When a temporary bucket is full, the points are flushed

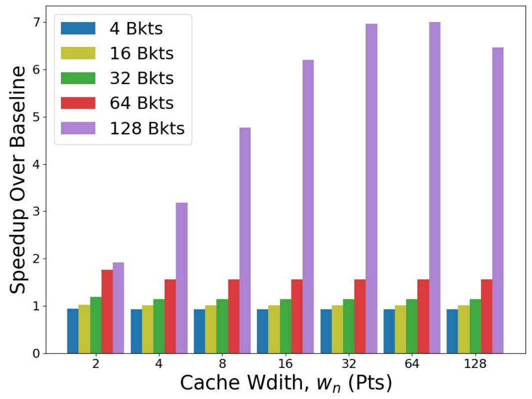


Figure 8: Speedup of external memory access for various configurations of the write-gather cache.

to memory. Since a bucket is stored in a contiguous memory chunk, Wr1 is transformed to sequential access. When the cache is full, i.e., all w_b buckets have been allocated, the fullest one is flushed to memory to make room.

We simulated the speedup of memory access using the write-gather cache with a model of the DRAM. The simulation is done for the KITTI dataset with 30k points per frame, 256 points per bucket and 128 buckets. As seen in Figure 8, the speedup changes dramatically when there are more buckets. The results show that it is more important to prioritize w_b over w_n. Even a modest cache which stores 128 buckets and 4 points per bucket can provide a 3$\times$ speedup in external memory access.

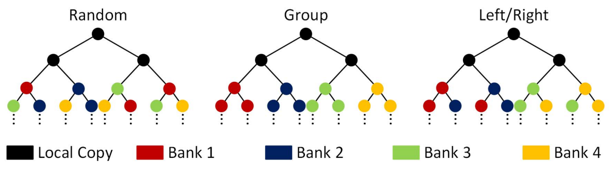
Read-Gather Cache TSearch snoops query points from TBuild's read stream Rd1. For each query point, TSearch traverses the tree to find the reference bucket to perform NN search over. If multiple query points belong to the same bucket, the bucket is reused and the Rd3 bandwidth is reduced. The read-gather cache is designed to temporarily store r_b buckets of up to r_n query points each.

The read-gather cache operates in a similar way as the write-gather cache. When a temporary bucket is full, the corresponding bucket is read from memory and NN search is executed. When the cache is full, the fullest bucket is evicted from cache and the query points distributed to the FUs. The read-gather cache provides the same speed up as seen in Figure 8. The only difference is that in order to maximally utilize all FUs, $r_n \geq N_{FU}$, where N_{FU} is the number of FUs.

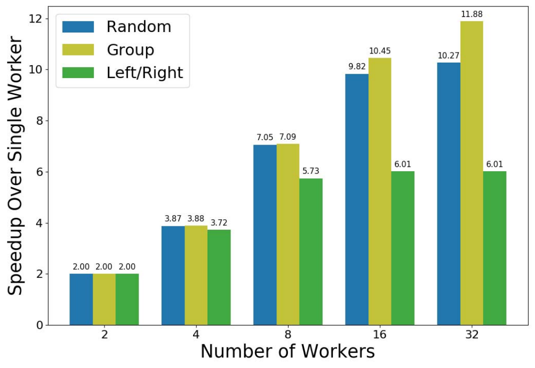
4.3 Parallel Tree Traversal

Tree traversal is a common operation in TBuild and TSearch. In TBuild, after the tree is constructed, the points of the reference frame are placed into the buckets. That is, for each point, the tree is traversed to find the bucket to insert the point into. In TSearch, for each point in the query frame, the tree is searched to find the bucket to perform the linear search over.

Each tree traversal requires a worker and a tree cache. For each point, the worker fetches nodes starting from the root of the tree, and then compares the point with the threshold and dimension of each node to decide which of the next child nodes to fetch. The traversal is complete when a leaf node is



(a)



(b)

Figure 9: (top) Three cache partition schemes for splitting nodes between local storage and banks of the cache. (bottom) Speedup with multiple workers with the three cache partition schemes.

reached.

We propose an efficient parallel tree traversal scheme. The parallel tree traversal uses multiple workers in parallel, and each needs a local copy of the tree nodes to prevent access collision. Replicating workers is relatively inexpensive due to the simple processing logic, but keeping multiple local copies of the tree nodes can be costly.

To reduce the cost of this replication, we note that in parallel tree traversal, multiple workers operate on multiple points in parallel, likely following different paths down the tree. Assuming equal likelihood of accessing a node at any given level, at level i of the tree, the probability of accessing a node by a worker is 2^{-i}. The few nodes in the upper portion of the tree are more likely to be accessed by multiple workers simultaneously. Nodes in the lower portion of the tree are increasingly less likely to be accessed, but the number of them is greater. Therefore, we duplicate only the upper portion of the tree and provide a local copy to each worker; while only a single copy of the lower portion of the tree is kept in cache.

After a worker has gone through the upper portion of the tree, it must request the lower tree nodes from the cache one layer at a time. As more workers are allocated, the cache bandwidth must also increase. To increase the cache bandwidth, the lower portion of the tree is split and stored in multiple banks. On requesting a node from the cache, the worker must decide which bank to access and submit a request. As the worker traverses deeper into the tree, the competition for cache bandwidth decreases.

We modeled the order of access to the caches in simulation to understand the best way to partition the tree across the banks of the cache. Results presented here are for a cache with 4 banks, though similar conclusions can be made for more banks. Three partition methods were simulated, as shown in Figure 9a. The first method is a random assignment of a node to a cache bank. The rationale is that because the traversal through the tree is through a random path, on average workers should be requesting nodes uniformly from different banks. The second method is a group method where entire subtrees under some upper-level node are stored in a single cache bank. The rationale is that since each worker has an approximately equal likelihood of ending up in each subtree, the distribution of requests to the cache banks should be even. The final method is a left/right method. This method splits nodes in a group into nodes which are left children and right children, and places them into their own cache bank. The goal is to help ease contention for a group's cache bank if multiple workers are placing points in that group.

As shown in Figure 9b, the speedup obtained is nearly linear for random and group methods for up to 8 workers with 4 banks. The additional gains from more workers diminishes, showing that the cache banks are fully utilized. In general, n cache banks supports up to $2n$ workers for a $2n$ increase in throughput. The group method for cache partition gives the best performance. Interestingly, the left/right partition method performs poorly. One reason is that larger buckets tend to be either a left or right child, and thus more bandwidth is required for those nodes.

4.4 Incremental Tree Update

To ease the burden of building a k-d tree, it is possible to reuse portions of a previous tree in a new tree's construction. For example, the tree is first constructed based on the initial reference frame, and remains static for all subsequent frames, i.e., the tree nodes (thresholds that define the partitions) remain static. For all subsequent frames, only the buckets are updated. For a mostly static scene, this aggressive approach can save a significant amount of computation in TBuild. The drawback is that a static tree may poorly fit the point distribution in later frames, causing either too many or too few partitions of a space. Such an imbalanced tree creates inefficiencies in the tree's use, leading to a higher search latency and a lower search accuracy.

We propose incremental tree update to maintain a balanced tree without having to do tree construction from scratch for every frame. The approach applies an upper and a lower bound to the number of points in each bucket after point insertion. If the number of points in a bucket dips below the lower bound, the leaf is absorbed into a higher node; if the number of points in a bucket rises above the upper bound, the leaf is split to two or more new leafs.

The incremental tree update can be implemented after point insertion using two steps: merging and splitting. Starting with the leaf nodes of the least depth, if the size of a bucket is below the lower bound, the leaf is marked delinquent. All the points that lie below the delinquent leaf's parent node are merged and the subtree below the parent node is rebuilt. After merging of delinquent leafs, leafs that contain more points above the upper bound are marked oversized. For each oversized leaf, a new subtree is created using the same method as in tree construction.

The goal of incremental tree update is to maintain a bal-

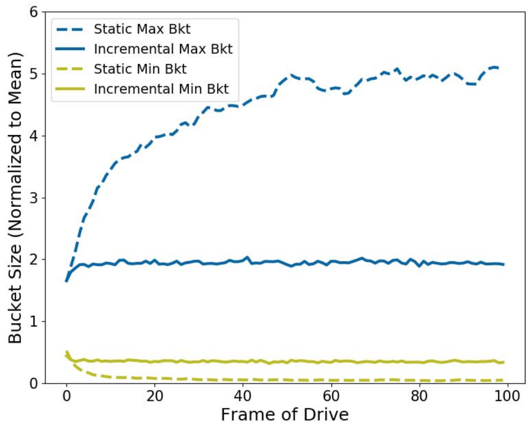


Figure 10: Maximum and minimum bucket size with static and incremental tree update over time.

anced tree. To motivate this, the performance of incremental tree update over successive frames was studied and compared to the static approach. For this test, a full tree was prepared based on the first frame. On new frames, the tree was reused. As Figure 10 shows, the tree's balance deteriorates after only a few frames, as indicated by the divergence of the maximum and the minimum bucket size. The incremental update maintains the tree balance, and the maximum and the minimum bucket size remain relatively constant at twice and half of the average, respectively. Similar results were obtained across all test drives in the KITTI dataset.

The complexity of incremental tree update is low. It involves merging and splitting tree nodes, local sorting, and partitioning. Merging and splitting tree nodes require small additional control logic, and sorting and partitioning are already supported by the sorter and partition modules inside TBuild without requiring additional hardware. During incremental updates, the collapsed nodes needing to be sorted are predominantly in the lower levels of the tree involving far fewer points than N (sort complexity is $N \log N$), the incremental tree update is significantly faster than building a tree from scratch.

However, at our current operating point of less than 100k points, the tree building takes less than a quarter of the total processing time in TBuild. As such, we chose not to apply incremental tree update in the prototype design. Expanding the current system to an operating point on the order of 1M points, the tree construction time will grow to be the more significant part of TBuild, and incremental tree update will be essential.

5. COMPLETE ARCHITECTURE

The complete QuickNN architecture is shown in Figure 11, consisting of two primary parts, TBuild and TSearch that share an interface to external DRAM.

TBuild TBuild makes use of multiple FSM-based modules coordinated by control logic to orchestrate the construction of the tree. TBuild contains a scratchpad for sample point storage, a tree cache storing tree nodes, a bucket cache storing the memory map of bucket blocks, and a write-gather cache. The total cache size for TBuild is 38.6 kB when sized for frames with 30k points. To simplify the retrieval and management of points, a point control module provides a simplified interface. Similarly, the bucket control module stores and maintains the bucket mapping to DRAM with the bucket cache. All caches in QuickNN use a standard word-addressable format.

TBuild operates in 3 phases: 1) points which will be used to construct the tree are sampled from the reference frame and brought into the scratchpad, 2) k-d tree is constructed based on the sample points in the scatchpad, and 3) the remaining points from the reference frame are streamed in from DRAM, placed into the buckets, and written back to memory.

One computationally demanding step of the tree construction process is sorting of the points. Sorting is done with a dedicated merge sort accelerator similar to that in [36]. It performs an n-way merge sort in rounds with a complexity of $N \log_n(N)$ for N points.

TSearch TSearch performs the nearest neighbor search using the k-d tree constructed by TBuild. TSearch contains a tree cache and a bucket cache, identical to those in TBuild, and a read-gather cache. The total cache size for TSearch is 33-243 kB for designs with 16-128 FUs sized for frames with 30k points. The vast majority of the cache in TSearch is used for the read-gather cache where the query points are temporarily stored. When enough query points have accumulated for a bucket in the read-gather cache, they are distributed to the FUs to perform the search.

TSearch adopts a modified version of the linear search architecture introduced in Section 3, including identical FUs. The control for the linear search was modified to read buckets, rather than the full frame. Because only a bucket is searched, TSearch consumes a significantly lower external memory bandwidth than the linear search architecture, resulting in a much faster speed and a higher energy efficiency.

6. SIMULATION AND PROTOTYPING

Both the linear and the QuickNN architectures were modeled in SystemVerilog. This allowed for simulations to be performed to verify and tune the architectures before they were placed on FPGA. A Xilinx VCU118 evaluation board [37] was used for FPGA prototyping. Software simulations and hardware emulation on FPGA were tested using the KITTI dataset [9]. To ensure our results were consistent across multiple situations, key benchmarks were crosschecked with the Ford Campus Vision and Lidar Data Set [20].

6.1 Modeling and Prototyping Platform

An accurate memory model that reflects the random access penalty is very important. In simulations, we used a custom model of the external DRAM. The memory parameters used in the model were based on a representative DDR4 RAM chip [38]. A SystemVerilog model was created to reflect the various latency penalties based on the order of access. Because the behavior of a kNN processor changes depending on the exact latency of memory requests, the memory SystemVerilog model is co-simulated with the kNN processor architecture model.

A Xilinx VCU118 FPGA [37] was used to prototype the kNN designs. This platform was chosen because it had a

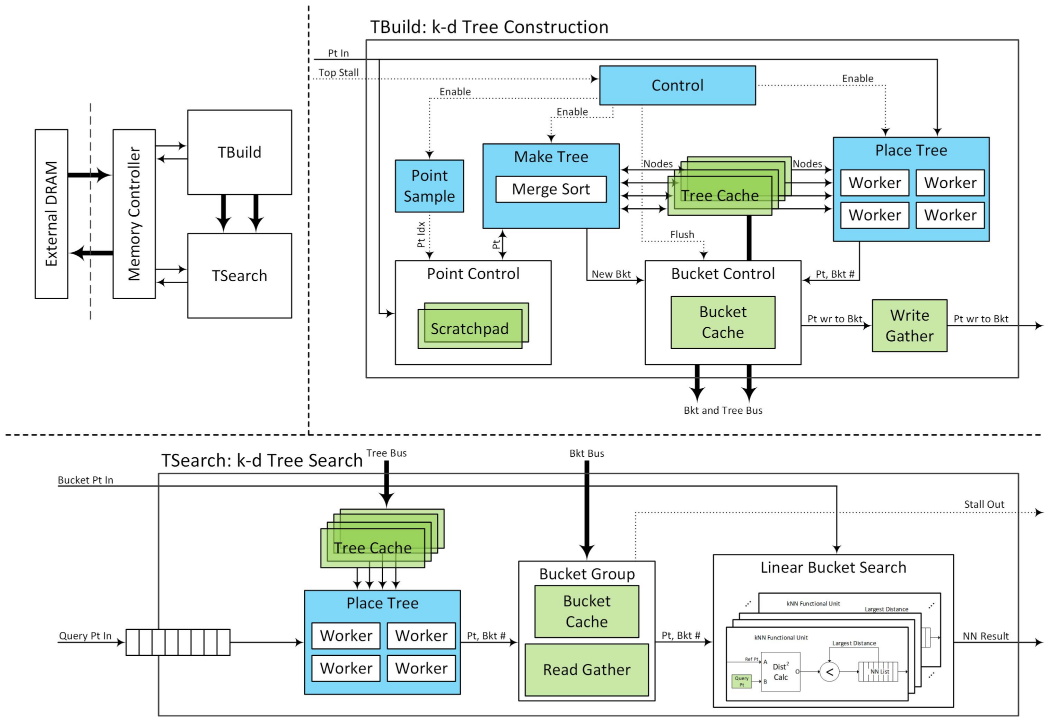


Figure 11: Complete QuickNN architecture diagram. Blue blocks represent FSM driven components. Green components represent on chip storage. (Top, Left) Top level view of architecture. (Top, Right) Tree builder half of architecture. (Bottom) Tree search half of architecture.

realistic DDR4 memory interface. We utilized the on-board DDR4 memory through the Xilinx MIG controller [39]. Communication with the board was through a serial interface to the host PC. This interface allowed transferring of the LiDAR frames to the DDR4 memory, reading back results, and monitoring execution progress. Timing and cycle counts presented here consider both the time the accelerator was running and/or accessing external DRAM. The core clock for the design is 100MHz. Peripheral and wrapper logic run at 300MHz. A 64-bit wide memory interface is used to match the DDR4 width.

The FPGA resource utilization and the percentage of total FPGA utilization for both the linear and the QuickNN architectures with 64 FUs are presented in Table 2 and Table 3, respectively. Post synthesis utilization for QuickNN is broken down into TBuild and TSearch. The total includes TBuild, TSearch, and the wrapper logic needed around the core, including the DDR4 controller and the PC interface. The post place and route utilization reported is higher than what it needs to be, due to the relaxed space on the FPGA. For example, to ease routing, 14 DSP slices were used per FU instead of the expected 8. Additionally, much of the cache is implemented in register arrays instead of block RAM (BRAM) explaining the decrease in BRAM post place and route. The power presented is given by the Xilinx Power Estimator [40].

The performance of the FPGA designs was measured in terms of core clock cycles and it is interchangeable with wall time by multiplying the number of cycles by the 10 ns clock period. While varying one parameter, all others were kept constant. Results are the average of running 100 consecutive

Table 2: FPGA Resource Utilization for Linear Search Implementation with 64 FUs

Architecture	Linear	
Component	Post Synthesis	Post Place & Route
LUTs	45,458 (3.84%)	139,876 (11.83%)
Registers	40,024 (1.69%)	112,371 (4.75%)
BRAM	30 (1.39%)	0 (0.00%)
DSPs	512 (7.49%)	896 (13.10%)
Power	-	4.44W

Table 3: FPGA Resource Utilization for QuickNN Implementation with 64 FUs

Architecture	QuickNN			
Component	Post Synthesis			Post Place & Route
	TBuild	TSearch	Total	
LUTs	13,731	74,092	90,754	203,758 (17.23%)
Registers	11,535	45,682	79,002	152,962 (6.47%)
BRAM	0	1	31	1 (0.05%)
DSPs	0	512	512	896 (13.10%)
Power	-	-	-	4.73W

Table 4: Measured performance in FPS on FPGA (Linear Architecture)

	Frame Size w/o ground points		
FUs	10k Pts	20k Pts	30k Pts
32	**20.5**	5.2	2.3
64	**40.3**	**10.3**	4.6
128	**77.8**	**20.2**	9.1

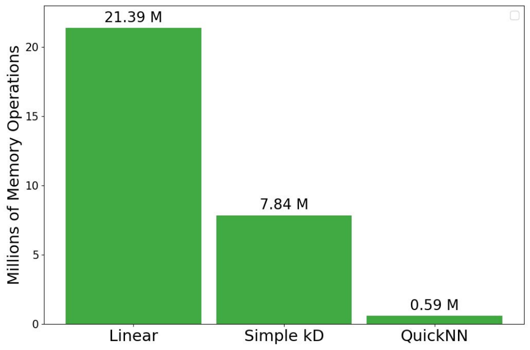


Figure 12: Number of external memory accesses required per frame (64 FUs, 30k points, 8 nearest neighbors)

frames on the device.

6.2 Linear Architecture

The performance of a linear search kNN is expected to grow proportionately with the number of FUs present as long as there is sufficient memory bandwidth to sustain the operations. The memory bandwidth utilization is high, due to the all-sequential access. The processing latency is expected to grow quadratically with the frame size.

For the linear architecture, the measured latency agrees with the predicted scaling and the overhead for adding FUs is negligible. Doubling the number of FUs from 32 to 64 gives a 1.99× speedup, and quadrupling from 32 to 128 gives a 3.93× speedup at 30k points per frame. Memory bandwidth utilization is very high, at 98.7%.

The prototype linear kNN runs at a clock speed of 100 MHz. The measured maximum frame rate for various configurations are shown in Table 4. Most modern LiDAR sensors are capable of producing over 10 frames per second. Thus, only the bold configurations in the table would be suitable. Additionally, many applications require even smaller latency for the system to react in real time. In these situations, a faster accelerator is needed.

6.3 QuickNN Architecture

The k-d tree narrows the search space and enables a significant reduction in memory access. Figure 12 shows the QuickNN architecture with our proposed memory optimizations cuts the external memory access by 36× compared to the linear architecture, and 13× compared to a Simple k-d architecture with only a simple cache and no memory

Table 5: Measured performance in FPS on FPGA (QuickNN)

	Frame Size w/o ground points		
FUs	10k Pts	20k Pts	30k Pts
16	**138.6**	**74.8**	**44.2**
32	**221.5**	**120.4**	**73.1**
64	**325.2**	**176.3**	**110.1**
128	**422.7**	**224.8**	**145.6**

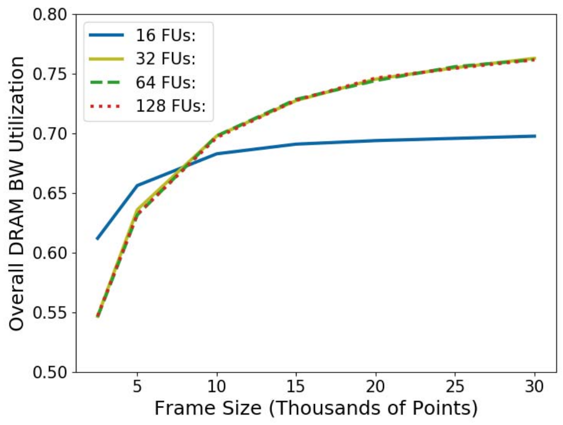


Figure 13: Measured memory bandwidth utilization for the QuickNN architecture on FPGA.

optimizations. Since the same amount of computation is performed in both the QuickNN and Simple k-d architectures, the reduction in external memory access directly translates to energy and latency savings.

In addition, as Figure 13 shows, the memory bandwidth utilization for the QuickNN architecture reaches 76% for all configurations with 32 FUs or more for 30k-point frames. Memory access reduction, together with a high bandwidth utilization, ensures more effective acceleration. The processing latency of a 64-FU QuickNN is measured to be 908k cycles per frame, a 24.1× speedup compared to a 64-FU linear architecture.

Number of Nearest Neighbors Varying the number of nearest neighbors, k, primarily affects the buffer required within each FU, and the number of memory transactions required to write the results back to memory. Both the buffering and the additional memory transactions are relatively minor for small k. Only when the number of FUs is large, the overhead becomes noticeable, as seen in Figure 14.

Frame Size Performance for different frame sizes and number of FUs is shown in Figure 15. The latency for the architecture scales nearly linearly across this range of frame sizes. Though the tree search latency scales with $O(N \log N)$, since we cache the tree on-chip, the total latency is now dominated by accessing N points from external memory, which explains the linear dependency of latency on frame size.

Number of FUs Increasing the number of FUs initially gives a near linear increase in performance and reduction in latency as shown in Figure 15 and Table 5. However, the gain

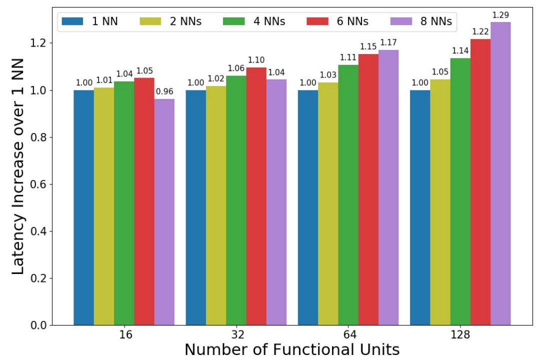


Figure 14: Latency increase with the number of nearest neighbors for the QuickNN architecture.

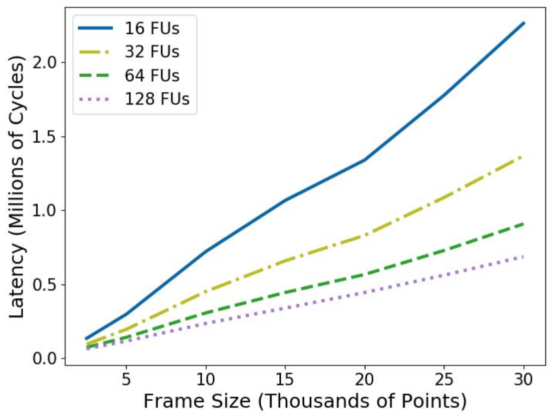


Figure 15: Total latency per frame for the QuickNN architecture.

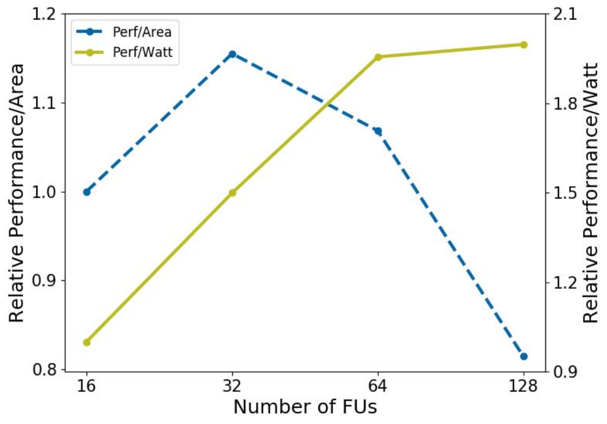


Figure 16: Performance per area and per watt scaling with number of functional units for the QuickNN architecture on FPGA.

diminishes with more FUs due to the imperfect utilization. To realize the full gain from more FUs, the read-gather cache in the TSearch needs to be enlarged accordingly to collect more query points to keep the FUs busy.

As a design space exploration, Figure 16 shows how the number of FUs affects the performance per unit area and power. Here, the area is the FPGA design footprint in terms of logic and memory (LUT+FF). As the number of FUs increases, the performance per watt increases as the memory bandwidth utilization improves while the fixed overhead of the memory interface is amortized. However, the performance per area decreases after 32 FUs, suggesting lower utilization of the growing read-gather cache. In the next section, we use the 16-FU design to represent the low-area low-power design point and the 128-FU design as the high-performance design point.

7. COMPARISON WITH OTHER METHODS

Currently, LiDAR data is typically processed on a large multi-core CPU, so we benchmarked the performance of both the linear and the k-d tree search on an Intel i7-7700k CPU using the popular FLANN library [32,33]. Additionally, an open-source k-d tree search [41] is benchmarked on an Nvidia GTX 1080 Ti GPU. This CPU and GPU are both good representations of what is commonly found in an automotive platform. Identical datasets were used in both CPU and GPU benchmarking, and FPGA implementation.

Figure 17 shows the performance of the k-d tree search methods running on CPU and GPU compared to the implementations on FPGA. The FPGA-based QuickNN implementations scale the same as the software k-d tree search running on CPU and GPU, but the FPGA-based implementations run at least an order of magnitude faster, as it is more straightforward to parallelize k-d tree search on FPGA. The memory optimization is essential to ensure that QuickNN does not become memory-bounded. The linear method is also implemented on FPGA as a baseline to show that k-d tree search scales more favorably with the number of points.

Table 6: Speedup and performance per Watt normalized to k-d tree search on CPU (30k points, 8 nearest neighbors)

Device	CPU	GPU	FPGA	
Design	k-d tree	k-d tree	QuickNN 16 FUs	QuickNN 128 FUs
Speedup	1	2.62	6.82	19.0
Perf/Watt	1	3.55	152	334

The speedup of the FPGA-based implementations over the k-d tree search on CPU and GPU are shown in Table 6. A relatively small 16-FU QuickNN on FPGA achieves a 6.82× speedup over the k-d tree search running on CPU, and a 2.6× speedup over k-d tree search on GPU. A larger 128-FU QuickNN on FPGA has a 19.0× speedup over the k-d tree search on CPU, and a 7.3× speedup over k-d tree search on GPU.

The performance per watt of QuickNN on FPGA is compared with the CPU and GPU benchmarks in Table 6. Our 128-FU QuickNN has a measured performance per watt of

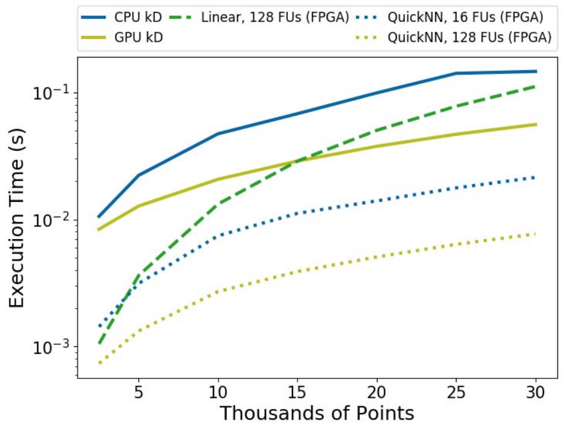


Figure 17: Latency comparison of software implementations of k-d tree search on CPU and GPU with hardware implementations of linear search and QuickNN on FPGA (8 nearest neighbors).

334× higher than the CPU, and 94× higher than the GPU. QuickNN running on FPGA has a clear advantage over CPU and GPU methods when it comes to power efficiency.

7.1 Survey of Prior Accelerators

To the best of our knowledge, this work is the first that focuses on large-size nearest neighbor search for 3D point clouds. Previous work on kNN acceleration targets predominately graphics applications [42, 43], physical simulation [19, 44], or matching high dimensional data [45].

A previous kNN design [19] focuses on accelerating the k-d tree search. The design leaves k-d tree building and maintenance to software. An example application is hydrodynamic fluid simulation of up to 5k points. Directly comparing this to a 128-FU QuickNN on FPGA operating on 5k-point LiDAR frames, QuickNN outperforms by 75×. The speedup is attributed to the QuickNN architecture that maintains the entire k-d tree structure and manages memory access proactively.

The work in [45] focuses on high dimensional queries. It targets applications such as word embedding where the space can have hundreds to thousands of dimensions. Additionally, the search phase of the application consists of only a few queries, whereas our application involves queries on the same order as the dataset size. The work [45] reported performance on the order of hundreds of queries per second, which is still four orders of magnitude lower than what QuickNN can deliver.

There has been prior work on accelerating k-d tree construction for graphics applications on GPUs. For example, the work [42] accelerates k-d tree construction for ray tracing. The work presented an algorithm modification for ray tracing to enable simpler construction of an approximate tree on GPU. [43] presents FastTree, an architecture dedicated to constructing and using k-d trees for ray tracing applications. Both pieces of work do not consider k-d tree search. Scaling QuickNN to FastTree's benchmark of roughly 65k points, QuickNN performs the k-d tree construction and nearest neighbor search 13% faster than FastTree which only does tree construction.

7.2 Scaling for Future Workloads

The trend in autonomous driving is to equip higher resolution LiDAR that produces an ever increasing amount of data for processing. As data size increases, the kNN search becomes more challenging. Working with one or two orders of magnitude more points will bring out new bottlenecks in the system.

Tree construction latency is expected to grow faster than any other part of the system when moving towards much larger point clouds. We foresee two techniques to mitigate this difficulty. The first is by relying on the incremental tree update that is described in Section 4.4, and the second is to utilize a high-bandwidth memory.

Currently, the most significant bottleneck in the system is the limited external memory bandwidth. Multiple parts in the architecture are limited by the rate at which they can access external memory. In total, one frame of points are read or written at least three times, once in reading the points for point insertion, once in writing the points to their buckets, and the third time in reading the points for the nearest neighbor search. One emerging choice to solve this problem is using near-chip memory such as High Bandwidth Memory (HBM) [46]. Many applications that rely on fast access to large amounts of data have added HBM to the system such as Nvidia's latest Volta GPUs [47], some of Xilinx's Virtex Ultrascale+ FPGAs [48], and Google's latest TPU [49]. Using a platform that can store point clouds near the processor would alleviate the bottleneck.

The memory bandwidth usage of QuickNN can be adjusted by adjusting the parallelism of the point placement step. This involves partial replication of the tree cache, which is relatively small. The number of FUs can be scaled up to allow for a massively parallel search across the buckets. These modifications represent natural progressions of the architecture to make it adaptable to the expected advances in sensor and memory interface bandwidth.

8. CONCLUSION

This paper presents QuickNN, a new architecture for k-d tree based approximate nearest neighbor search on 3D point clouds. The architecture relies on a judicious data structure caching scheme and proactive reduction and regularization of memory accesses to reduce the memory bandwidth and increase the effective utilization. The memory management enables more speedup by parallel processing. We prototyped the QuickNN architecture on FPGA, demonstrating up to 19× and 7.3× speedups over k-d tree searches performed on a modern CPU and GPU, respectively, and a two-order-of-magnitude increase in performance per watt from both the CPU and the GPU designs. QuickNN is a parameterized architecture which can easily be scaled to meet the requirements of future workloads.

9. REFERENCES

[1] L. Kobbelt and M. Botsch, "A survey of point-based techniques in computer graphics," *Computers & Graphics*, vol. 28, no. 6, pp. 801–814, 2004.

[2] R. A. Fowler, "The lowdown on lidar," *Earth Observation Magazine*, vol. 19, 2000.

[3] B. Schwarz, "Lidar: Mapping the world in 3d," *Nature Photonics*, vol. 4, no. 7, p. 429, 2010.

[4] A. Avadi, C. Premebida, P. Peixoto, and U. Nunes, "3d lidar-based static and moving obstacle detection in driving environments: An approach based on voxels and multi-region ground planes," *Robotics and Autonomous Systems*, vol. 83, pp. 299–311, September 2016.

[5] R. Halterman and M. Bruch, "Velodyne hdl-64e lidar for unmanned surface vehicle obstacle detection," in *Unmanned Systems Technology XII*, vol. 7692. International Society for Optics and Photonics, 2010, p. 76920D.

[6] J. Liu, P. Jayakumar, J. L. Stein, and T. Ersal, "A multi-stage optimization formulation for mpc-based obstacle avoidance in autonomous vehicles using a lidar sensor," in *Dynamic Systems and Control Conference*, 2014.

[7] J. Zhu and P. Morton, "Methods and systems for pedestrian avoidance using lidar," Apr. 19 2016, uS Patent 9,315,192.

[8] *Puck: VLP-16*, Velodyne LiDAR, Inc., 2019, rev-J.

[9] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *International Journal of Robotics Research (IJRR)*, 2013.

[10] M. S. Belshaw and M. A. Greenspan, "A high speed iterative closest point tracker on an fpga platform," in *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*. IEEE, 2009, pp. 1449–1456.

[11] S. Zeng, "An object-tracking algorithm for 3-d range data using motion and surface estimation," *IEEE Transactions on Intelligent Transportatin Systems*, vol. 14, no. 3, pp. 1109–1118, 2013.

[12] J. Robinson, M. Piekenbrock, L. Burchett, S. Nykl, B. Woolley, and A. Terzuoli, "Parallelized iterative closest point for autonomous aerial refueling," in *International Symposium on Visual Computing*. Springer, 2016, pp. 593–602.

[13] J. Levinson and S. Thrun, "Robust vehicle localization in urban environments using probabilistic maps," in *IEEE International Conference on Robotics and Automation*. IEEE, 2010.

[14] B. Yohannan, D. Chandy, and A. Christinal, "Detection of drivable path in lidar image using gray level neighbours matrix and k- nearest neighbour," *International Journal of Advanced Research in Education & Technology (IJARET)*, vol. 4, no. 2, 2017.

[15] P. Navarro, C. Fernandez, R. Borraz, and D. Alonso, "A machine learning approach to pedestrian detection for autonomous vehicles using high-definition 3d range data," *Sensors*, vol. 17, no. 1, p. 18, 2017.

[16] Z. Zhang, *Iterative Closest Point (ICP)*. Boston, MA: Springer US, 2014, pp. 433–434. [Online]. Available: https://doi.org/10.1007/978-0-387-31439-6_179

[17] A. Segal, D. HÃď'hnel, and S. Thrun, "Generalized-icp," 06 2009.

[18] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 751–766. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173191

[19] S. Heinzle, G. Guennebaud, M. Botsch, and M. Gross, "A hardware processing unit for point sets," 06 2008, pp. 21–31.

[20] G. Pandey, J. R. McBride, and R. M. Eustice, "Ford campus vision and lidar data set," *International Journal of Robotics Research*, vol. 30, no. 13, pp. 1543–1552, 2011.

[21] D. Zermas, I. Izzat, and N. Papanikolopoulos, "Fast segmentation of 3d point clouds: A paradigm on lidar data for autonomous vehicle applications," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2017, pp. 5067–5073.

[22] P. J. Besl and N. D. McKay, "Method for registration of 3-d shapes," in *Sensor fusion IV: control paradigms and data structures*, vol. 1611. International Society for Optics and Photonics, 1992, pp. 586–606.

[23] Y. Chen and G. Medioni, "Object modelling by registration of multiple range images," *Image and vision computing*, vol. 10, no. 3, pp. 145–155, 1992.

[24] Z. Zhang, "Iterative point matching for registration of free-form curves and surfaces," *International journal of computer vision*, vol. 13, no. 2, pp. 119–152, 1994.

[25] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[26] J. H. Friedman, F. Baskett, and L. J. Shustek, "An algorithm for finding nearest neighbors," *IEEE Transactions on computers*, vol. 100, no. 10, pp. 1000–1006, 1975.

[27] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975. [Online]. Available: http://doi.acm.org/10.1145/361002.361007

[28] M. Greenspan and M. Yurick, "Approximate kd tree search for efficient icp," in *Fourth International Conference on 3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings.* IEEE, 2003, pp. 442–448.

[29] A. Nüchter, K. Lingemann, J. Hertzberg, and H. Surmann, "6d slamâĂŤ3d mapping outdoor environments," *Journal of Field Robotics*, vol. 24, no. 8-9, pp. 699–722, 2007.

[30] W.-S. Choi, Y.-S. Kim, S.-Y. Oh, and J. Lee, "Fast iterative closest point framework for 3d lidar data in intelligent vehicle," in *2012 IEEE Intelligent Vehicles Symposium*. IEEE, 2012, pp. 1029–1034.

[31] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, no. 7, pp. 881–892, 2002.

[32] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *International Conference on Computer Vision Theory and Application VISSAPP'09)*. INSTICC Press, 2009, pp. 331–340.

[33] M. Muja and D. Lowe, "Flann-fast library for approximate nearest neighbors user manual," *Computer Science Department, University of British Columbia, Vancouver, BC, Canada*, 2009.

[34] A. Gionis, P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing," in *Vldb*, vol. 99, no. 6, 1999, pp. 518–529.

[35] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: Efficient indexing for high-dimensional similarity search," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 950–961. [Online]. Available: http://dl.acm.org/citation.cfm?id=1325851.1325958

[36] S. H. Pugsley, A. Deb, R. Balasubramonian, and F. Li, "Fixed-function hardware sorting accelerators for near data mapreduce execution," in *Computer Design (ICCD), 2015 33rd IEEE International Conference on*. IEEE, 2015, pp. 439–442.

[37] *UG1224: VCU118 Evaluation Board*, Xilinx, 2018, rev. 1.4.

[38] *4Gb: x4, x8, x16 DDR4 SDRAM*, Micron, 2014, rev. k.

[39] *PG150: UG440: UltraScale Architecture-Based FPGAs Memory IP v1.4*, Xilinx, 2018, rev. 1.4.

[40] *UG440: Xilinx Power Estimator User Guide*, Xilinx, 2018, rev. 2018.3.

[41] A. Mock, "cudanormals," https://github.com/aock/kNNcuda, 2017.

[42] Z. Li, T. Wang, and Y. Deng, "Fully parallel kd-tree construction for real-time ray tracing," in *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '14. New York, NY, USA: ACM, 2014, pp. 159–159. [Online]. Available: http://doi.acm.org/10.1145/2556700.2566638

[43] X. Liu, Y. Deng, Y. Ni, and Z. Li, "Fasttree: A hardware kd-tree construction acceleration engine for real-time ray tracing," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 1595–1598.

[44] T. Kuhara, T. Miyajima, M. Yoshimi, and H. Amano, "An fpga acceleration for the kd-tree search in photon mapping," in

International Symposium on Applied Reconfigurable Computing. Springer, 2013, pp. 25–36.

[45] V. T. Lee, A. Mazumdar, C. C. del Mundo, A. Alaghi, L. Ceze, and M. Oskin, “Application-driven near-data processing for similarity search,” *arXiv preprint arXiv:1606.03742*, 2016.

[46] *JESD235: High bandwidth memory (hbm) dram*, JEDEC, 2013.

[47] NVIDIA, “V100 gpu architecture,” 2017.

[48] M. Wissolik, D. Zacher, A. Torza, and B. Da, “Virtex ultrascale+ hbm fpga: A revolutionary increase in memory performance,” *Xilinx Whitepaper*, 2017.

[49] Google. (2018) Cloud tpu system architecture. [Online]. Available: https://cloud.google.com/tpu/docs/system-architecture