Crescent: Taming Memory Irregularities for Accelerating Deep Point Cloud Analytics

Yu Feng
University of Rochester
Rochester, NY, USA
yfeng28@ur.rochester.edu

Gunnar Hammonds
University of Rochester
Rochester, NY, USA
ghammon5@u.rochester.edu

Yiming Gan
University of Rochester
Rochester, NY, USA
ygan10@ur.rochester.edu

Yuhao Zhu
University of Rochester
Rochester, NY, USA
yzhu@rochester.edu

ABSTRACT

3D perception in point clouds presents exciting opportunities to transform the perception ability of future intelligent machines. Point cloud algorithms, however, are plagued by irregular memory accesses, leading to massive inefficiencies in the memory subsystem, which bottlenecks the overall efficiency.

This paper proposes Crescent, an algorithm-hardware co-design system that tames the irregularities in deep point cloud analytics while achieving high accuracy. To that end, we introduce two approximation techniques, approximate neighbor search and selectively bank conflict elision, that "regularize" the DRAM and SRAM memory accesses. Doing so, however, necessarily introduces accuracy loss, which we mitigate by a new network training procedure that integrates approximation into the network training process. In essence, our training procedure trains models that are conditioned upon a specific approximate setting and, thus, retain a high accuracy. Experiments show that Crescent doubles the performance and halves the energy consumption compared to an optimized baseline accelerator with $< 1\%$ accuracy loss. The code of our paper is available at: https://github.com/horizon-research/crescent.

CCS CONCEPTS

• **Computer systems organization → Neural networks**; **System on a chip**; **Special purpose systems**.

KEYWORDS

point cloud, accelerators, DNN, irregular memory accesses

ACM Reference Format:
Yu Feng, Gunnar Hammonds, Yiming Gan, and Yuhao Zhu. 2022. Crescent: Taming Memory Irregularities for Accelerating Deep Point Cloud Analytics. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3470496.3527395

1 INTRODUCTION

Recent years have seen an explosive rise of intelligent machines that can perceive, process, and understand visual data. 3D visual data, a.k.a., point clouds, have become increasingly important. Prime examples include localization and mapping in autonomous vehicles [35, 62] and robotics [52], object detection in Augmented and Virtual reality [56, 72], air pollutants detection [17], and geo-spatial mapping in cultural heritage preservation [7, 25, 58].

Despite much algorithmic development, point cloud networks are inefficient to execute on today's hardware architectures (e.g., GPUs, deep learning/stencil accelerators), most of which are designed and optimized for regular 2D perception domains such as video and image processing [27, 46]. Point cloud algorithms, however, exhibit highly irregular computation and memory behaviors and, thus, are ill-suited for architectures built for regular kernels.

The irregularity stems from the fact that memory accesses, which dominate the overall execution efficiency, are input-dependent. As a result, point cloud algorithms exhibit excessive and random (as opposed to streaming) DRAM accesses as well as frequent SRAM bank conflicts that stall the datapath. Many mature optimizations such as tiling, double-buffering, static data layout that are commonly applied to regular kernels such as conventional Deep Neural Networks (DNNs) are either ineffective or not applicable at all.

This paper proposes Crescent, an algorithm-hardware co-designed system aiming to tame the irregularities in point cloud algorithms. We start by understanding the sources of memory inefficiency in point cloud algorithms (Sec. 2), which points to two main sources. First, point cloud algorithms spend a significant amount of time (up to 80%) [18] in explicit neighbor searches, which exhibit statically-unknown memory access patterns. Second, the irregular neighbor search necessitates that any subsequent operations must explicitly aggregate data points through irregular gather operations instead of simply indexing the memory as in conventional DNNs.

Our key idea is to *impose structures* on memory accesses. We propose an approximate neighbor search algorithm (Sec. 3) that turns irregular DRAM accesses into streaming accesses. While there are search algorithms that preserve streaming accesses, they often do so at a cost of increasing the search work and/or redundant DRAM accesses by resorting to exhaustive search [44, 66]. We use a different strategy: we use an irregular tree-based algorithm to reduce the search work, and selectively elide on-chip bank conflicts to tame the irregularities stemmed from tree traversals (Sec. 4). This strategy reduces the search work and DRAM traffic by over 40%.

Our techniques are inexact by nature. Without care, applying them during inference leads to drastic network accuracy loss. To retain accuracy, we propose approximation-aware training (Sec. 5). Specifically, we integrate the approximation operations into training by modeling hardware behaviors (e.g., bank conflicts) at training time. We show that training with a generic hardware model is usually sufficient, which allows us to avoid tightly coupling training with a particular hardware configuration.

Our training procedure yields models that provide accuracy-vs-performance trade-offs *at inference time without re-training*. This is achieved without increasing the network size or inference overhead. The key is to train a network by sampling not only the input distribution (as with conventional DNN training) but also the distribution of a set of approximation knobs that dictate the accuracy-vs-performance trade-off. In this way, the model's inference is *conditioned* upon a specific approximate setting **h**, naturally presenting a different accuracy-vs-performance trade-off for a given **h**.

We implement the Crescent hardware in a 16nm process node and evaluate it on a set of popular point cloud models. We show that the optimizations introduced in Crescent require virtually zero hardware cost and, meanwhile, provide on average 1.9 × speedup (up to 3.1 ×) and 1.5 × energy reduction (up to 4.2 ×) compared to an optimized baseline point cloud accelerator without our optimizations. Notably, the performance and energy gains are achieved with less than 1.0% accuracy loss.

In summary, this paper makes the following contributions.

- We introduce an approximate neighbor search algorithm and its co-designed hardware, which guarantees completely streaming DRAM accesses while reducing the DRAM traffic in point cloud DNNs.
- We introduce selectively bank conflict, a lightweight mechanism that avoids datapath stalls from bank conflicts and reduces SRAM traffic in point cloud networks.
- We propose a network training procedure that integrates the approximate neighbor search and selective bank conflict into training to mitigate the accuracy loss while providing a flexible accuracy-vs-performance trade-off at inference time.
- We show that our optimizations collectively achieve 1.9 × speedup and 1.5 × energy reduction for a set of popular point cloud networks compared to a baseline accelerator while sacrificing less than 1% accuracy.

The rest of this paper is organized as follows. We first characterize the memory inefficiencies, both in DRAM and on-chip SRAM, of today's point cloud networks (Sec. 2). We then introduce two techniques to tame the memory inefficiencies: approximate neighbor search that guarantees fully streaming DRAM accesses (Sec. 3) and selectively bank conflict elision, which streamlines on-chip memory accesses (Sec. 4). We then introduce a neural network training procedure that integrates both approximate techniques into the training process to mitigate the accuracy loss (Sec. 5). After describing the experimental setup (Sec. 6), we demonstrate the efficiency of Crescent (Sec. 7). We then discuss Crescent in the broad literature (Sec. 8) before concluding the paper (Sec. 9).

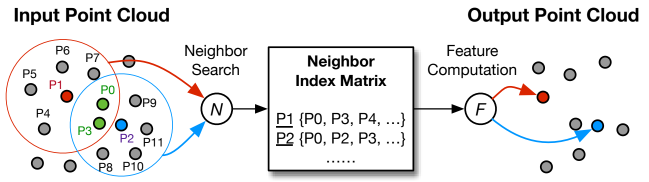


Fig. 1: A typical layer in a point cloud neural network, which has two main stages: neighbor search and feature computation. Neighbor search in itself is highly irregular as it requires tree traversal and is input-dependent. The feature computation requires irregular memory accesses because the input data are from the neighbor search results.

2 MOTIVATION

We first briefly describe the scope of deep point cloud algorithms that this paper targets, and describe the two main algorithmic stages, neighbor search and feature computation, in these algorithms (Sec. 2.1). We then quantify the memory inefficiencies in both the neighbor search stage (Sec. 2.2) and the feature computation stage (Sec. 2.3).

2.1 Background: Deep Point Cloud Analytics

A point cloud is a collection of points, each of which is represented by the [x, y, z] coordinates in the 3D space. Point cloud data are becoming ever more relevant mainly because of two trends: 1) the prevalence of convenient point cloud acquisition devices, e.g., stereo cameras [39] and LiDAR [55], and 2) the emergence of deep learning algorithms that can effectively extract semantics information from point clouds. Today, deep point cloud models are routinely deployed in real-world systems such as Waymo's self-driving cars [8] and Google's Augmented Reality toolkit [1].

We focus on algorithms that directly operate on raw points, which is by far the most common form of deep point cloud analytics. We refer interested readers to Guo et al. [26] for a comprehensive survey on deep learning for point clouds.

Key Operations Generally, a point cloud DNN can be abstracted as two stages, as shown in Fig. 1. Each input point undergoes a neighbor search process. The neighbor search results are stored in a matrix, where each row stores the neighbor indices of a point in the input. The feature computation stage aggregates the neighbors of a point, on which a transformation, usually a Multilayer Perceptron (MLP), is applied, to generate a new output point.

Both stages are important to optimize. A recent study on five popular point cloud networks shows that the execution time ratio of the two stages varies between 1:4 to 4:1 [18], suggesting that neither stage universally dominates.

2.2 Memory Inefficiencies in Neighbor Search

Neighbor search in low-dimensional space (e.g., 3D) commonly uses the K-d tree [14], which recursively subdivides the search space into two half-spaces using axis-aligned planes. The sub-spaces are organized as a tree, and neighbor search becomes a tree traversal problem. Compared to exhaustive search, the space subdivision strategy is more efficient as it prunes the search space: if the distance

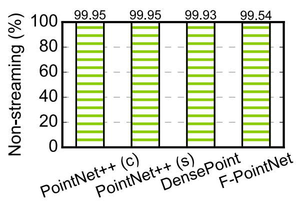


Fig. 2: Percentage of non-continuous DRAM accesses.

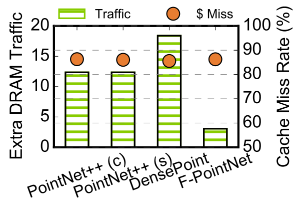


Fig. 3: Ratio of actual DRAM traffic vs. the theoretical minimum and cache miss rate in neighbor search.

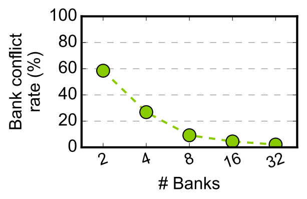


Fig. 4: Neighbor search bank conflict rate in Pointnet++(c) vs. the number of banks, assuming 8 concurrent queries.

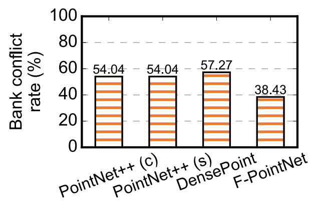


Fig. 5: SRAM bank conflict rate in aggregation, assuming 16 banks and 16 concurrent memory requests.

of a query Q and the boundary of a subspace $\mathbb{S}$ is greater than the search radius, all the points in $\mathbb{S}$ can be skipped.

While K-d tree search is inherently parallel (as different search queries are independent), tree traversals are hardware unfriendly. In particular, the memory access patterns are known only at run time, leading to massive inefficiencies in both DRAM and SRAM accesses, which we quantify below.

DRAM DRAM access inefficiency in neighbor search is manifested in two ways: non-streaming accesses and redundant accesses. The DRAM accesses are non-streaming because the inputs (points) are arbitrarily distributed in the search space. If two queries being processed in parallel are spatially far-apart, they will likely exercise different parts of the K-d tree that are non-contiguous in memory. Even within the same query, tree nodes consecutively accessed during traversals are likely non-continuous in memory due to the control-flow heavy nature of tree traversal. Fig. 2 shows the percentage of non-continuous DRAM accesses across four popular point cloud DNNs (see Sec. 6 for a comprehensive experimental setup). Almost all DRAM accesses are non-continuous.

The non-streaming nature coupled with large point cloud data size leads to redundant DRAM accesses. For instance, in the popular KITTI dataset [20], the total points and queries in a typical scene *alone* can be over tens of MBs (not considering the network weights, activations, etc.), larger than what a mobile SoC can accommodate. Thus, points are loaded on-chip in chunks (analogous to tiling in conventional DNNs). Since not all data in each chunk will be used when they are loaded due to the non-streaming access pattern, a great amount of DRAM accesses are wasted.

Fig. 3 quantifies the excessive DRAM accesses and cache miss rate in neighbor search. The left y-axis shows the ratio of the amount of DRAM requests (in bytes) to the actual data theoretically needed by the algorithm (i.e., reading each query and search point once). The data are obtained by simulating an unrealistic 10 MB fully-associated cache running a neighbor search on a typical KITTI-constructed scene with about 1.2 million points. Even with this unrealistic SRAM structure, searches in many models have about 10× more DRAM traffic than what is strictly required. Realistic mobile accelerators would allocate an even smaller buffer for neighbor search to accommodate other data structures such as DNN weights and activations. The right y-axis quantifies the corresponding cache miss rates, which are over 85%.

SRAM The on-chip memory accesses in K-d tree search are also inefficient because of the frequent bank conflicts. In regular kernels such as stencil pipelines [46, 61] where the memory access pattern is statically known, one could carefully interleave data in the SRAM banks to avoid bank conflicts [33, 71]. In contrast, on-chip memory accesses in neighbor search are input-dependent and, thus, bank conflicts are inevitable.

Fig. 4 quantifies the bank conflicts by showing the percentage of SRAM accesses that are bank-conflicted and how the percentage varies with the number of banks. We assume an unrealistically large 10 MB buffer and 8 concurrent SRAM requests. With 4 banks the bank conflict rate is 26.9%. The bank conflict rate is reduced to 2.1% only when the number of banks quadruples the number of simultaneous requests.

Using a heavily-banked SRAM design is highly undesirable. A large number of banks requires a more costly crossbar design [9, 24], as the crossbar area grows quadratically with the number of banks. Using an Arm memory compiler [3], we find that the crossbar area is twice as much as the memory arrays under a 32-bank configuration. In addition, a higher bank count also reduces the memory array size, which increases the per-bank overhead (peripheral circuits, BIST, redundancy) [60].

2.3 Memory Inefficiencies in Feature Computation

Unlike neighbor search, the DRAM accesses in the feature computation stage are completely streaming. The on-chip memory accesses, however, are met with frequent bank conflicts.

Feature computation is broken down into two steps: 1) aggregate the neighbors for each input point $\mathbf{p}_i$ using the neighbor indices generated in the neighbor search stage, and 2) compute an output point $\mathbf{p}_o$ from each $\mathbf{p}_i$ by applying a function, usually a MLP, to the neighbors of $\mathbf{p}_i$. Step 2 is accelerated on today's DNN accelerators.

Step 1 is analogous to fetching data from the input feature map in a conventional DNN. However, conventional DNNs access consecutive feature map elements with statically-known patterns. Therefore, a compiler lays out data in the SRAM such that a simple single-bank, single-port memory array (using wide words) could serve memory requests from tens or hundreds of PEs in one cycle without stalling the PEs [2, 32, 71].

However, point cloud networks access non-consecutive memory in this step, because the neighbors of a point can be arbitrary. Therefore, the SRAM serving points are usually banked. Worse, the access pattern is statically-unknown, as it depends on the neighbor search results, which, in turn, depend on the inputs. Therefore, bank conflicts are inevitable.

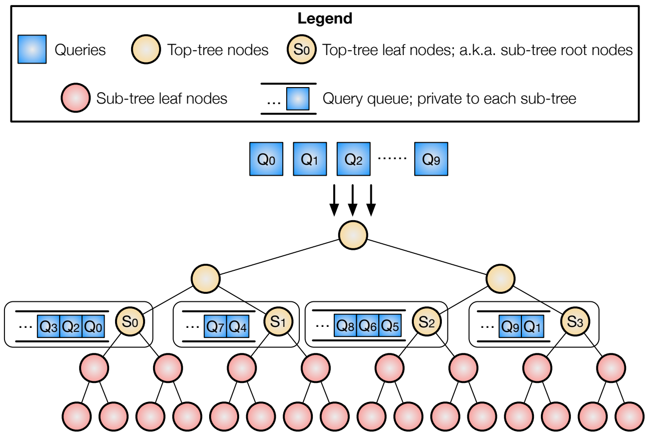


Fig. 6: The two-level tree data structure of our neighbor search algorithm. In the first stage, queries traverse the top-tree and are assigned to a particular sub-tree in the end. In the second stage, queries search neighbors in their assigned sub-tree, and backtracking is limited to within the sub-tree.

Fig. 5 quantifies the severity of bank conflicts in point aggregation by showing the percentage of SRAM accesses that are bank-conflicted in aggregating the points. We assume a 16-bank SRAM design with a total size of 64 KB. Across the four models, the bank conflict rate is at least 38.43% and can be as high as 57.27%. Increasing the number of banks is undesirable as it requires a more costly crossbar and/or a higher per-bank overhead due to the smaller memory arrays [60].

3 FULLY-STREAMING NEIGHBOR SEARCH ALGORITHM

We introduce our neighbor search algorithm and explain how it fundamentally improves the DRAM access efficiency by allowing completely streaming memory accesses (Sec. 3.1). We then describe the co-designed neighbor search hardware (Sec. 3.2). Finally, we discuss the key knob in our algorithm that dictates the accuracy-vs-performance trade-off (Sec. 3.3).

3.1 Algorithm

Our algorithm splits the K-d tree into a top tree and a set of sub-trees. Each top-tree leaf node is also the root node of a sub-tree. The search is then naturally divided into two stages: a top-tree search stage and a sub-tree search stage. The two stages themselves are massively parallel but are serialized with each other. Fig. 6 illustrates the idea.

In the first stage, all the queries search the top-tree (a binary search tree) until they reach the leaf nodes of the top-tree, at which point the queries are assigned to the corresponding sub-trees. Conceptually, each sub-tree has a queue that stores all the incoming queries. At the end of the first stage, queries in the sub-tree queues are written back to the memory in preparation for the second stage. In actual hardware, a queue has a fixed size. Thus, the store back to the memory is phased, as we will discuss later.

Once all the queries finish the first stage, the algorithm enters the second stage, where queries in each sub-tree are searched against the corresponding tree. For each sub-tree, the search process is exactly the same as that in the top-tree with a critical difference: queries are allowed to backtrack when they reach a leaf node of the sub-tree. This is necessary for a query to find all its neighbors.

However, we limit the backtracking to the sub-tree. The intuition is that nodes in other sub-trees are naturally far away from the query and thus are less likely to be neighbors. Architecturally, this ensures that each sub-tree and each query is loaded to SRAM once — at a cost of accuracy loss. We will discuss the accuracy implication of this design decision in Sec. 3.3 and how to mitigate the accuracy loss through approximation-aware network training in Sec. 5.

3.2 Hardware Design

The hardware designed to exploit the algorithm is shown in Fig. 7. The search is carried out by a set of PEs, each of which can execute a query independently. The PEs access data from the on-chip SRAM that stores various data structures. The SRAM interfaces with the DRAM through a DMA, as all DRAM accesses are streaming.

SRAM The SRAM is split into two global buffers and two local buffers. The global tree buffer and query buffer are accessed by all the PEs. Each PE is also equipped with a local result buffer and a local stack buffer private to each query.

The global tree buffer is accessed by the PEs simultaneously. To sustain a high read bandwidth, the tree buffer is heavily banked. Unlike in regular kernels, bank conflicts here could not be avoided by optimizing the data layout in the banks, because the access pattern of the PEs is known only at run time. We will show in Sec. 4 how to mitigate the performance impact of bank conflicts.

PE Design The PE design follows the algorithm of how a query traverses the K-d tree to search for its neighbors. As shown in the left blown-up panel on Fig. 7, a PE is pipelined into five stages, starting from reading the top of the traversal stack (RS) to fetch the next tree node to visit (FN), followed by calculating the distance between the query and the three node (CD), storing results (SR) is a neighbor is found, and ended with updating the stack (US). The pipeline stalls only when the FN stage meets a bank conflict when reading the global tree buffer.

Hardware Reuse Due to the uniform traversal-based search in both top- and sub-tree searches, the hardware is reused in both phases. For instance, the PEs are designed with the generic traversal logic that is agnostic to what the search tree is and what the queries are. The US stage is skipped/bypassed in the top-tree search where no backtracking takes place (i.e., no update to the query stack).

The SRAM is also reused between the two phases. Specifically, the PE-local result buffer is re-purposed between storing the sub-tree queues in the top-tree search phase and storing the neighbor results in the sub-tree search phase. The global tree buffer is re-purposed between storing the top-tree and storing the sub-tree. During top-tree search, whenever a result buffer is full all the queries assigned to that queue (thus far) are streamed back to the DRAM.

3.3 Accuracy and Performance Trade-off

A key parameter that governs our algorithm is the top-tree height (TTH). TTH must be set to ensure both the top-tree and the sub-trees can be held in the on-chip SRAM. At the same time, TTH

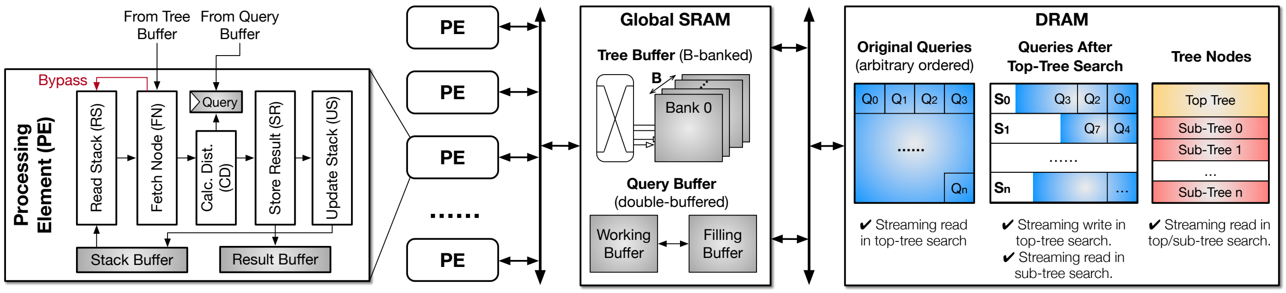


Fig. 7: Neighbor search hardware engine, which enables a fully-streaming access to DRAM. The same hardware is used for both top-tree search and for the sub-tree searches, simplifying the hardware design.

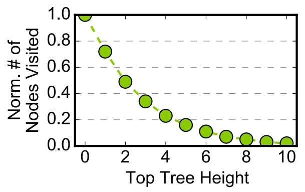


Fig. 8: Number of tree nodes visited per query reduces as the top-tree height increases.

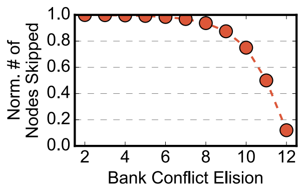


Fig. 9: Number of tree nodes skipped per query reduces as the elision height increases.

also dictates the performance-vs-accuracy trade-off. We explore the implication of TTH in this section.

First, the top-tree height is dictated by the tree buffer size. We require that the entirety of the top-tree or a sub-tree, while is being searched, is completely stored in the tree buffer. This ensures that the PE pipeline does not stall because the required data are off-chip. Thus, the top-tree height h_t must be within the range $[\mathcal{H} + 1 - \log_2(\mathcal{S} + 1),\ \log_2(\mathcal{S} + 1)]$ to satisfy the following two inequalities, where $\mathcal{H}$ is the total K-d tree height and $\mathcal{S}$ is the total tree buffer size:

$$2^{h_t} - 1 \leq \mathcal{S} \tag{1}$$

$$2^{\mathcal{H}-h_t+1} - 1 \leq \mathcal{S} \tag{2}$$

Given that a TTH is within the permissible range, a shorter top-tree increases the neighbor search accuracy at a cost of more computation, and vice versa. This can be explained by a first-order analytical model, where the total number of nodes a query accesses is proportional to the sum of:

(i) the number of nodes visited during forward traversal from the root node of the top-tree to a leaf node of the sub-tree,
(ii) the number of nodes visited during sub-tree backtracking.

The cost of (i) is constant, as it depends only on the total tree height. The cost of (ii) inversely depends on the TTH: a taller top-tree translates to visiting fewer nodes in the sub-tree backtracking, reducing the cost of (ii) and, by extension, the total cost. Fig. 8 quantifies how the total number of nodes accessed per query (y-axis) varies with the TTH (x-axis) using the average statistics of PointNet++(c) on the KITTI dataset. As the TTH increases to 10, only 2% of the tree nodes are accessed by a query. Visiting fewer nodes improves the search speed but also degrades the accuracy.

An assumption we make, as with Tigris [66] and QuickNN [44], is that a sub-tree can be stored completely on-chip. This is a reasonable assumption: a typical 10 MB point cloud using a 5-level top-tree would result in a sub-tree size of 640 KB, smaller than a typical on-chip buffer size found in mobile SoCs. In case of excessively large point clouds, Crescent can in theory recursively split a sub-tree; we do not observe this need in common datasets.

3.4 Efficiency Discussion

The split-tree algorithm enables completely streaming DRAM accesses. The panel on the right of Fig. 7 shows how the different data structures are laid out in the DRAM and how they are accessed in a streaming fashion. Converting random DRAM accesses to streaming accesses reduces the DRAM energy [6, 19], and enables double-buffering, which improves performance because: 1) off-chip data accesses are overlapped with computation, and 2) data needed by the datapath are readily available on-chip without stall.

Compared to prior neighbor search algorithms that also enable streaming accesses such as Tigris [66] and QuickNN [44], we reduce both the search load and DRAM traffic. We qualitatively discuss it here, and quantify the gains in Sec. 7.5.

First, Tigris and QuickNN use exhaustive search in the sub-trees, whereas we retain K-d tree search in the sub-trees and thereby reduce the total search load. Retaining K-d tree search is not an obvious design decision, as it introduces irregular *on-chip* memory accesses in the form of bank conflicts, which prior work aims to avoid at a cost of more search work.

Our strategy is different: we reduce the search work by retaining K-d tree search and mitigate the resulting irregular on-chip memory accesses through inference-training co-design. Specifically, we will show a selective bank conflict elision scheme to significantly reduce bank conflicts (Sec. 4), which, when coupled with an approximation-aware training procedure (Sec. 5), retains the application accuracy.

Second, we reduce the amount of DRAM accesses compared to Tigris and QuickNN, both of which load (and reload) a sub-tree from DRAM whenever the corresponding query buffer is full. We instead first stage all the queries to a sub-tree in DRAM and then process them in a batch, thus loading each sub-tree exactly once.

4 SELECTIVE BANK CONFLICT ELISION

This section addresses inefficiencies pertaining to on-chip memory accesses. We first describe our main idea of selectively eliding bank conflicts (Sec. 4.1). We then discuss how point cloud algorithms proceed when bank conflicts are elided (Sec. 4.2) and the hardware support (Sec. 4.3). Finally, we identify the key knobs that dictate the accuracy-vs-performance trade-off (Sec. 4.4).

4.1 Main Idea

A key requirement of the SRAM design is to feed data required by the PEs without stalling them. Such a requirement is easy to meet in conventional DNNs or other regular kernels, where data access patterns are statically known and thus SRAM data layout can be statically optimized accordingly [71]. The on-chip memory access patterns in point cloud algorithms, however, are only dynamically known, introducing SRAM bank conflicts that are detrimental to overall performance.

Motivated by the error-tolerance nature of neural networks, our idea is to dynamically and selectively ignore bank conflicts when appropriate. That is, when multiple memory requests fall in the same bank, instead of serializing the accesses we allow only one request to access the SRAM; other requests return immediately without stalling. While conceptually simple, actually realizing this idea requires answering three questions:

(1) What happens when a bank conflict occurs? That is, how should the algorithm proceed without the correct data?
(2) How to support bank conflict elision in hardware?
(3) When is it appropriate to elide bank conflicts without accuracy drop?

The answers to these questions depend on where a bank conflict takes place in the algorithm, because different memory accesses request data of different significance. Both neighbor search stage and feature computation introduce bank conflicts. In neighbor search, bank conflicts are caused by accessing the tree buffer; all other accesses are regular. In feature computation, aggregating neighbors of a point as inputs to the MLP causes bank conflicts; SRAM accesses incurred during MLP are regular. We now elaborate how the three questions above are addressed in both stages.

4.2 How Algorithms Proceed with Bank Conflicts Elision

Feature Computation To aggregate neighbors, SRAM accesses are made to retrieve neighbors of a point. Thus, ignoring a conflicted access essentially ignores a point's neighbor, in which case we must fill in the missing neighbors, as the subsequent MLP anticipates an input matrix of a given size (decided at the training time).

To meet the size requirement, we propose to simply reuse the point returned from the request that *is* allowed to access the bank. The intuition is that concurrent accesses, say A and B, are guaranteed to be requesting neighbors of the same point P [18]. Reusing the returned data from A for B is equivalent to replicating one of P's neighbors. This replication strategy is commonly done in point cloud network design to meet the size requirement in case a neighbor search does not return enough neighbors [48, 49]. Our design essentially performs this replication in hardware, implicitly.

Neighbor Search The situation is slightly different for neighbor search, where bank conflicts happen when the PEs access the tree nodes during tree traversal. One could use the same replication strategy used in the feature computation stage: if accesses A_1 from PE 1 and A_2 from PE 2 conflict on the same bank, reuse the data

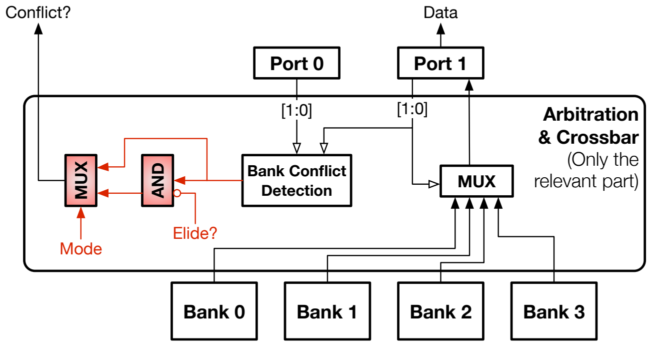


Fig. 10: Supporting bank conflict elision is trivial in hardware, as many existing hardware structures can be reused. The shaded/colored components are the augmentation, which is required for each SRAM port. Only the relevant part of the hardware is shown for simplicity. The *Mode* signal selects between the neighbor search mode and the feature computation mode. The AND gate lowers the *Conflict* signal when bank conflict elision is enabled in the neighbor search stage.

returned from A_1 for A_2. However, this could lead to side effects such as program crash, redundant computation, and infinite loop. For instance, when the node returned from A_1 is in the part of the tree that PE 2 has already visited, pushing A_1 onto PE 2's stack leads to an infinite loop or, at least, redundant traversals.

Our design simply ignores the conflicted accesses. Upon a conflict, the FN stage in a PE skips the rest three pipeline stages, and reads the next item on the stack. This is indicated by the "bypass" signal in the PE architecture shown in Fig. 7. Algorithmically, this is equivalent to skipping all the nodes beneath the lost node during tree traversal. This strategy omits potential neighbors but guarantees that the traversal terminates.

A potential optimization that we leave for future work is to check whether the node returned from A_1, the request allowed to access the SRAM, is below beneath the node (in the tree) that would have been returned from A_2 if the bank conflict were to be observed; if so, using A_1 to continue the search in P_2 is guaranteed to terminate without side effects. Doing so would skip fewer nodes and potentially increase the accuracy.

4.3 Hardware Support

Eliding bank conflicts is virtually free to implement in hardware by using many existing structures in banked SRAM design. As an example, Fig. 10 shows a simple banked SRAM with 2 ports and 4 banks. The key to a banked SRAM is the arbitration and crossbar logic, which detects bank conflicts and routes data from a bank to the right port (a MUX here). For simplicity, we show only the relevant hardware and assume a low-order interleaving, i.e., the two least significant bits in the address select a bank.

Assume both accesses from the two ports fall into Bank 0, and Port 0 is allowed access. In the baseline SRAM, the MUX before Port 1 would select data returned from Bank 0, but this data will be ignored because the bank conflict detection logic would raise the *Conflict* signal, indicating to Port 1 that a bank conflict occurs and the memory request is to be issued again. But, critically, the data returned from Bank 0 is exactly what Port 1 needs in the feature computation stage under bank conflict elision. We simply lower the *Conflict* signal in this case, which is accomplished by ANDing the output of bank conflict detection and the negation of the *Elide* signal, which indicates whether bank conflict elision is enabled.

The *Mode* signal operates a MUX to select between the neighbor search mode and the feature computation mode. In neighbor search, the original bank conflict signal is used, except the PE will not re-issue the memory request; instead, the PE simply continues the search with the next item on the stack.

4.4 When to Elide Bank Conflicts?

Eliding bank conflicts returns incorrect data to the PEs and, thus, hurts accuracy. We find that eliding bank conflicts in feature computation leads to little to none accuracy loss whereas eliding bank conflicts during neighbor search, without care, has significant accuracy implications (Sec. 7.3). This is because in feature computation the data that would have been returned (if bank conflicts were observed) are replaced with the data returned from the conflicting access; in neighbor search, however, eliding bank conflicts directly skips all the computations associated with that node altogether. We thus focus on the neighbor search stage here.

Intuitively, the accuracy loss is smaller when ignoring a memory access made to a lower level tree node, as fewer tree nodes would be skipped later in the traversal. Fig. 9 shows how the percentage of skipped tree nodes (y-axis) varies with the tree level below which bank conflicts are elided (x-axis). The statistics are averaged across all the queries of PointNet++(c) on the ModelNet dataset, where the total tree height is 14. When bank-conflicted accesses below level 2 are ignored, almost 100% of the tree nodes are skipped, which degrades the model accuracy to almost zero (not shown). When the elision level is 12, only 10% of the tree nodes are skipped.

Skipping more nodes degrades accuracy but increases the search speed. Therefore, a natural knob that controls the trade-off of accuracy-vs-performance is the *elision height* h_e, which is defined as the tree level beneath which all conflicted memory accesses are ignored. Sec. 5 will show how incorporating h_e into model training can minimize the accuracy loss while providing the accuracy-vs-performance trade-off without retraining.

5 APPROXIMATION-AWARE NETWORK TRAINING

Our neighbor search algorithm and bank conflict elision, if applied directly on a trained point cloud DNN at inference-time, will decrease the accuracy sharply (Sec. 7.1). This is because the original network is not trained with the various approximation techniques in mind. To mitigate the accuracy drop, we propose a modified network training procedure that mitigates the accuracy loss.

The goal here is to learn a DNN that retains a high accuracy under approximation compared to the baseline network. In particular, we consider two approximation knobs: the top-tree height h_t and the elision height h_e. Briefly, a larger h_t decreases accuracy but increases the performance; conversely, a larger h_e increases the accuracy at a cost of a lower performance.

A straightforward idea is to integrate $\mathbf{h} =< h_t, h_e >$ as part of the inference such that the DNN is trained for a particular $\mathbf{h}$. In

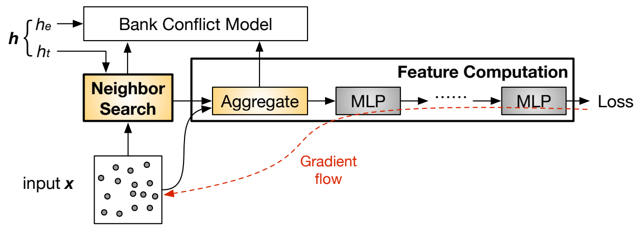


Fig. 11: Training a point cloud network with approximate neighbor search and bank conflict elision. Note that the training is end-to-end differentiable as in conventional DNN training. The non-differentiable parts, neighbor search and aggregation, do not participate in the gradient flow.

essence, this is similar to fine-tuning a compressed model to regain the accuracy, where a network learns to adjust its weights given the approximation introduced by a particular compression setting.

While one could train a dedicated model for each possible $\mathbf{h}$ and build an ensemble, that would increase the training overhead and deployment complexity. Instead, we propose to learn one model that adapts to different $\mathbf{h}$. Mathematically, we aim to learn a DNN distribution $f(\cdot, \mathbf{h}; \theta) \sim F$ such that different DNNs sampled from the distribution F share the same model parameter θ and provide similar accuracy given an input $\mathbf{h}$ (along with the input point cloud).

To that end, our training procedure augments the conventional training with one simple extension: conventional training samples input data; our training also randomly samples an $\mathbf{h}$ *for each input*. During the forward propagation, $\mathbf{h}$ is used to modulate the neighbor search and bank conflict elision. In this way, the model parameter θ is trained to accommodate the approximations introduced during the forward inference. The training flow is shown in Fig. 11.

In order to replay the same inference-time approximation during training, we integrate a hardware simulator for modeling the bank conflict. The bank conflict model is called by both neighbor search and feature computation (the aggregation operation) , as Fig. 11 shows. The bank conflict simulator takes in two parameters: 1) h_e, which indicates the tree level below which bank conflicts are elided, and 2) the hardware banking configuration (e.g., number of banks, bank size). We find that training with the exact banking configuration on the inference hardware yields higher accuracy, but absent an exact hardware configuration training with a generic banking configuration provides noticeable benefits, too (Sec. 7.3).

Finally, note that neighbor search and aggregation do not participate in gradient descent; they simply construct inputs to the MLP layers. Thus, the model is end-to-end differentiable even though neighbor search and aggregation are not.

6 EXPERIMENTAL SETUP

Architecture Design Fig. 12 shows the overall point cloud accelerator, which includes three main components: a neighbor search engine as described in Sec. 3.2, a neighbor aggregation unit, which uses the design proposed in Mesorasi [18], and a DNN accelerator for executing the MLPs. Without losing generality, we assume a systolic-array-based DNN accelerator, which is configured to have a 16×16 MAC array, where each MAC unit mimics the design of that in the TPU [31].

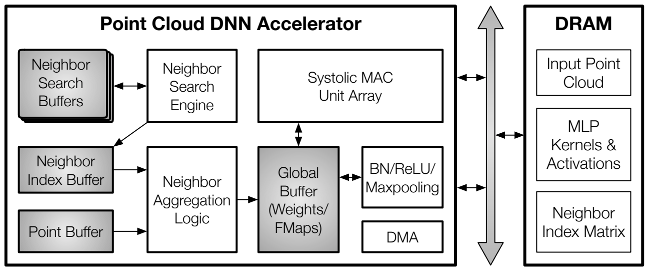


Fig. 12: Overall architecture of the point cloud DNN accelerator, which includes three main components: a Neighbor Search Engine, an Aggregation Unit, and a systolic array for executing the MLPs in feature computation. The Neighbor Search Buffers include all the buffers shown in Fig. 7.

Table 1: Evaluation models.

Application Domains	Algorithm	Dataset
Classification	PointNet++ (c) DensePoint	ModelNet40
Segmentation	PointNet++ (s)	ShapeNet
Detection	F-PointNet	KITTI

The on-chip SRAM is partitioned to serve different purposes. The global buffer serves the weight and activations for the systolic array. It is configured to be 1.5 MB in size. The Point Buffer is a 64 KB 16-banked buffer serving points during aggregation. The Neighbor Index Buffer is sized at 12 KB with a single bank. The Tree buffer and the Query buffer are sized at 6 KB and 3 KB with 4 banks and 1 bank, respectively. These two buffers support selective bank elision as described in Sec. 4.3. The neighbor search engine has 4 PEs, each with a dedicated result buffer and a stack buffer, which are sized at 1.5 KB and 256 B, respectively.

Experimental Methodology We synthesize, place, and route the datapath of the neighbor search engine, the systolic array, and the aggregation unit using an EDA flow consisting of Synopsys and Cadence tools with the TSMC 16 nm FinFET technology. The SRAMs are generated using the Arm Artisan memory compiler. Power is estimated using Synopsys PrimeTimePX by annotating the switching activity. We then build a cycle-accurate simulator of the architecture with the latency of each component parameterized from the post-synthesis results of the RTL design.

The DRAM is modeled after Micron 16 Gb LPDDR3-1600 (4 channels) according to its datasheet [5]. The DRAM energy is obtained using Micron System Power Calculators [6]. On average, the energy ratio between a random DRAM access and a streaming DRAM access is about 3:1, and the energy ratio between a random DRAM access and an SRAM access is about 25:1, both matching prior work [19, 67].

Software Setup Tbl. 1 lists the four point cloud networks used in the evaluation. To show the general applicability of our design, the models cover a wide range of common point cloud tasks including classification, segmentation, and detection. For classification, we evaluate the classic PointNet++(c) [49] and DensePoint [37] on the ModelNet40 dataset [65]. We use the overall accuracy as accuracy metric. For segmentation, we evaluate PointNet++(s) [49] on the ShapeNet dataset [15]. The metric used in segmentation is the standard Intersection-over-Unit (mIoU) accuracy. For detection, we evaluate F-PointNet [47] on the KITTI dataset [20] and report the geometric mean of the IoU metric on the car class.

To obtain more competitive baselines and to ensure that the improvements from Crescent are not due to the inefficiencies of the network implementation, we use the versions of these models optimized by Feng et al. [18], which removes redundant MLP computations and on average achieves 1.6× speedup over the corresponding author-released implementations.

Baseline We compare against three baselines:

- GPU: the mobile Pascal GPU on Nvidia's Jetson TX2 development board [4].
- Tigris+GPU: this baseline executes the neighbor search on Tigris [66], a recent neighbor search accelerator that does not perform approximate eighbor search and selectively bank conflict elision, and executes the feature computation on the mobile Pascal GPU.
- Mesorasi, a prior point cloud network accelerator [18] that uses Tigris [66] for neighbor search and executes the feature computation on a dedicated systolic-array without selectively bank conflict elision. The exact same systolic array configuration is used in Crescent with the exception that Crescent performs selective bank conflict elision.

Area Overhead Our accelerator has a total area of 1.55 mm^2, in which the Crescent-specific portion is almost negligible. The only hardware extension is one that selectively elides the bank conflict (Fig. 10), which requires an additional MUX and an AND gate for each port of the SRAM.

Traininig Overhead Our approximation-aware training increases the training time by 38%. The main overhead is to simulate bank conflicts, which currently is a multi-threaded CPU implementation. Using a random **h** does not further increase the training overhead, since we still perform one search per inference. Note that the training overhead is amortized across all subsequent inferences.

Variants We evaluate two variants of Crescent to decouple the contribution of the two optimizations:

- ANS performs approximate neighbor search but does not elide bank conflicts.
- ANS+BCE performs approximate neighbor search while also eliding bank conflicts in neighbor search and aggregation.

7 EVALUATION

We first show that Crescent achieves similar accuracy as the baseline (Sec. 7.1) but delivers significant speedups and energy reductions (Sec. 7.2). We then provide a detailed analysis of our training procedure and understand how its effectiveness varies with respect to different algorithmic and hardware configurations (Sec. 7.3). We perform sensitivity study to understand Crescent's performance and energy savings vary under different settings (Sec. 7.4). Finally, we provide an quantative comparison with prior neighbor search accelerators (Sec. 7.5).

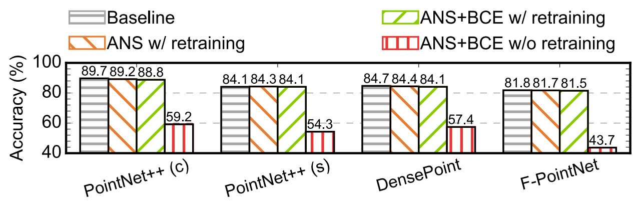


Fig. 13: Accuracy comparison between the baseline models, ANS+BCE without re-training, ANS with re-training under $h_t = 4$, and ANS+BCE with re-training under $h_t = 4$ and $h_e = 12$.

7.1 Accuracy

We find that directly applying Crescent optimizations without retraining significantly degrades the model accuracy. Integrating approximation into the training process elevates the accuracy to the baseline level. Fig. 13 compares the model accuracy between four schemes: 1) the baseline models, 2) ANS+BCE without re-training, 3) ANS+BCE with re-training, and 4) ANS with re-training. In this specific case, each re-trained model is trained specifically for the approximate setting where $h_t = 4$ and/or $h_e = 12$.

Directly applying the two optimizations at inference time degrades the accuracy between 27.3% to 40.5%, making the models practically useless. Re-training regains the accuracy with an accuracy drop of at most 0.9% (PointNet++(c)). In PointNet++(s), re-training completely recovers the accuracy loss introduced in approximation. The fact that we can almost completely recover the accuracy loss with ANS+BCE, the most aggressive approximation setting, shows the effectiveness of our approximation-aware training. The accuracy of ANS alone is slightly higher than that of ANS+BCE, as the latter applies two approximations whereas the former applies only one.

7.2 Performance and Energy

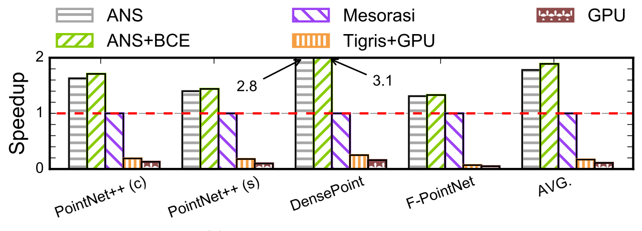
Using the re-trained ANS and ANS+BCE model shown in Fig. 13, we compare Crescent's performance and energy consumption over the baseline accelerator, shown in Fig. 14.

Speedup Fig. 14a shows the speedup of ANS and ANS+BCE against the three baselines; all data are normalized to Mesorasi. Among the three baselines, Tigris+GPU and GPU are much slower than Mesorasi, because the latter accelerates feature computation on a systolic array.

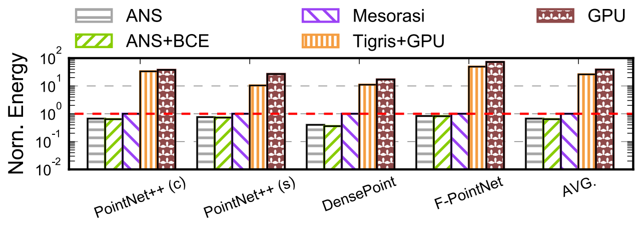
Overall, ANS and ANS+BCE achieve a 1.7× and 1.9× speedup, respectively, over Mesorasi. Comparing the speed of ANS+BCE and ANS shows that approximation neighbor search contributes more to the speedup than bank conflict elision. The speedups on DensePoint are the highest (2.8× and 3.1×, respectively) because DensePoint's time is dominated by neighbor search (81%) whereas neighbor search takes “only” about 55% of the time in other models.

To understand the sources of speedup, Fig. 15a and Fig. 15b show the speedup of ANS+BCE on neighbor search and on the aggregation operation in feature computation, respectively. On average, ANS+BCE achieves a 4.9× speedup on neighbor search and a 2.1× speedup on aggregation.

Energy Savings Fig. 14b shows the energy consumption of ANS and ANS+BCE normalized to Mesorasi. On average, ANS

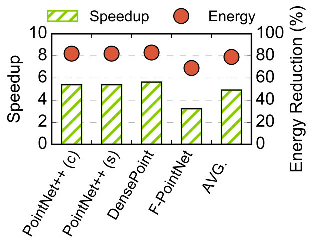


(a) Speedup. Higher is better.

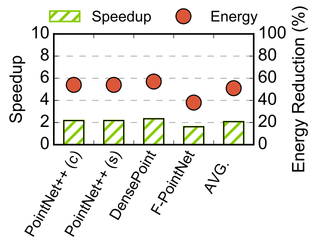


(b) Normalized energy. Lower is better.

Fig. 14: End-to-end speedup and normalized energy of ANS and ANS+BCE over the baseline.



(a) Neighbor search.



(b) Aggregation.

Fig. 15: Speedup and energy savings of ANS+BCE on neighbor search and aggregation alone.

and ANS+BCE saves 33% and 36% of the total energy, respectively. The energy saving is mainly contributed by approximate neighbor search rather than bank conflict elision, because the former optimizes the DRAM traffic, which contributes more to the energy than the SRAM traffic, which the latter optimizes for. DensePoint, again, has the highest energy saving because it is dominated by neighbor search. As a comparison, Tigris+GPU and GPU consume 25× and 38× more energy, respectively, compared to Mesorasi.

Fig. 15a and Fig. 15b on the right y-axes show the energy savings on neighbor search and aggregation. DensePoint's savings on these two operations *in isolation* are on par with other networks, confirming that its significant end-to-end savings are primarily attributed to the dominance of neighbor search in its execution time.

Tease Apart Contributions To understand the sources of energy savings, Fig. 16 decouples the memory energy savings into four components: converting random DRAM accesses to streaming accesses, DRAM traffic reduction, SRAM traffic reduction in neighbor search, and SRAM traffic reduction from aggregation. The former two are from our neighbor search algorithm, and the latter two are from bank conflict elision.

Generally, the main energy saving contributor is the SRAM traffic reduction in neighbor search, which frequently accesses the Tree

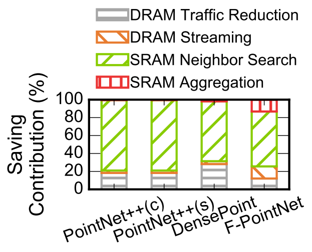


Fig. 16: Memory energy saving contribution.

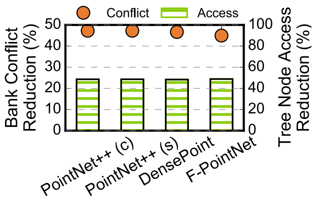


Fig. 17: Tree node access saving and bank conflict reduction of ANS+BCE

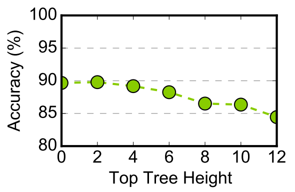


Fig. 18: Accuracy of dedicated PointNet++(c) models under different top-tree heights (h_t).

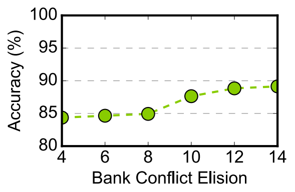


Fig. 19: Accuracy of dedicated PointNet++(c) models under different elision heights (h_e).

Buffer. While the DRAM savings are relatively smaller, we expect the DRAM savings will become more significant in the future as the point clouds grow in size.

We quantify the impact of selective bank conflict elision (BCE) in Fig. 17, where we show the reduction in bank conflicts (left y-axis) and, as a result, the reduction in the number of tree nodes visited (right y-axis). The results are obtained by comparing ANS+BCE with ANS. Overall, BCE avoids over 45% of bank conflicts and reduces 50% of tree node accesses in neighbor search. This result explains the 1.9× speedup over MESORASI by ANS+BCE.

7.3 Understanding the Training Procedure

We use PointNet++(c) as a representative model to drive the analyses in this section. The conclusions generally hold.

Dedicated Models We first evaluate the accuracy of models trained with dedicated approximation settings.

Fig. 18 shows the accuracy of PointNet++(c) trained under different top-tree heights (h_t) and then inferenced under the same h_t. The setting h_t being 0 is the baseline model with exact search. As the h_t increases, the accuracy decreases. This is because a larger h_t reduces the search space and, thus, it is less likely to find the exact neighbors for each query. The accuracy is acceptable initially, dropping from 89.6% to 88.8% as h_t increase from 0 to 4. Beyond 4, the accuracy drop becomes more significant. As the top-tree height reaches 12, the accuracy is only 84.4%. As we will shown later, however, a higher h_t leads to a higher speedup, providing a large trade-off space.

Fig. 19 performs a similar study while varying the elision height h_e. Each marker in the figure represents a dedicated ANS+BCE model trained with different h_e ranging from 4 to 14; h_t in this example is fixed at 4. As h_e increases, the accuracy increases. This is because a higher elision height skips fewer tree nodes during

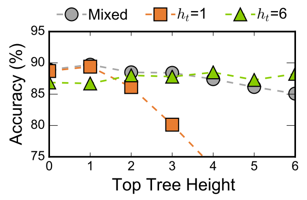


Fig. 20: Accuracy comparison of different training schemes.

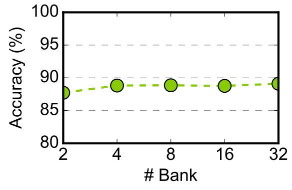


Fig. 21: Sensitivity of bank conflict simulation in training.

tree traversal, leading to a better search result. At a h_e of 12, the accuracy loss is only 0.8%. The accuracy loss is over 5% when h_e reduces to 4, essentially ignoring almost all nodes in the sub-tree.

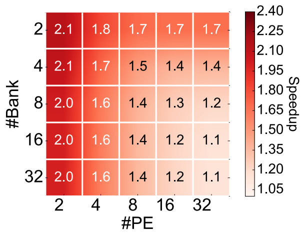
Mixed Training We now evaluate how a model trained by sampling different approximation settings adapts to different approximation levels at inference time. Fig. 20 compares three schemes: 1) a model trained with $h_t = 1$, 2) a model trained with $h_e = 6$, and 3) a model trained by random sampling h_t between 1 and 6 for each input ("Mixed" in the figure). We show their accuracy under different inference-time h_t.

When a dedicated model is trained with $h_t = 1$, the accuracy significantly drops when the inference-time h_t is greater than 1. This is not surprising: a model trained with little approximation in mind does not perform well when inference performs aggressive approximation. When a dedicated model is trained with $h_t = 6$, however, it performs reasonably well across different h_t at inference-time, even for h_t settings that are not seen in the training time.

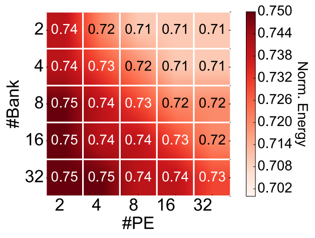
The mixed model consistently provides higher or similar accuracy compare the dedicated $h_t = 1$ model. Compared to the dedicated $h_t = 6$ model, the mixed model is significantly better when higher accuracy is required (i.e., $h_t \leq 3$). The accuracy is only noticeably worse than the dedicated $h_t = 6$ model when the inference-time h_t is 6, which is what the dedicated $h_t = 6$ model is trained to do well on. The mixed model is favorable when accuracy requirement is high, which is arguably more important than the low-accuracy regime.

Bank Conflict Simulation In order to integrate bank conflict elision into training, we simulate the bank conflicts in the forward propagation process during training. However, at training time the exact banking configuration of the target hardware might be unknown. Fig. 21 show the accuracy of training a model assuming 4 banks in the SRAM while inferencing under different numbers of banks. The accuracy beyond 8 is largely stable; the accuracy has about 2% drop when inferencing on a 2-banked SRAM.

BCE in Aggregation vs. Neighbor Search We perform bank conflict elision in both neighbor search and in feature aggregation. We find that the overall accuracy is insensitive to bank conflict elision in aggregation even *without* re-training. Across five networks, directly applying bank conflict elision in aggregation alone (while turning off other approximations) results in at most 0.3% accuracy loss. In contrast, accuracy typically drops by double digits if bank conflict elision is applied in neighbor search without re-training. As discussed in Sec. 4.4, this is because in the latter case eliding bank conflicts completely skips subsequent search operations.

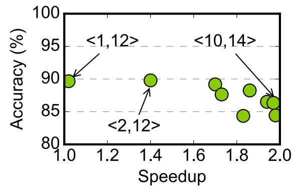


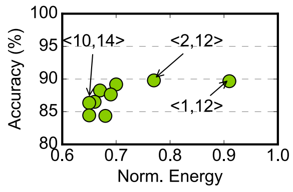
(a) Speedup sensitivity.



(b) Energy sensitivity.

Fig. 22: Sensitivity of speedup and (normalized) energy to hardware configuration (PE and bank counts) on PointNet++(c).





(a) Speedup-vs-accuracy trade-off. (b) Energy-vs-accuracy trade-off.

Fig. 23: Accuracy vs. performance vs. energy trade-off on PointNet++(c) under different $< h_t, h_e >$ combinations.

7.4 Sensitivity Study

Hardware Configuration Fig. 22a and Fig. 22b show how Crescent's speedup and energy vary, respectively, as the numbers of PEs and the number of Tree Buffer banks vary. The energy is normalized to the corresponding baseline.

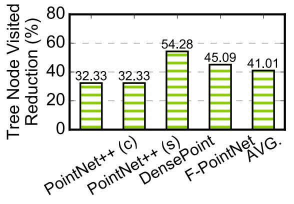
Naturally, the speedup is higher on less-capable baselines and diminishes on more capable baselines (e.g. 32 PEs and 32 banks), because performance optimizations are less important when the hardware is faster to begin with. Note, however, that a 16-bank memory introduces large cross-bar overhead and is generally impractical for mobile-grade accelerators [9, 71].

The significant energy saving is consistent across different hardware configurations. Even with a 32 PE 32 bank configuration, Crescent still saves about 27% energy on PointNet++(c). This is because the energy is roughly proportional to the amount of work done. Changing the hardware configuration does not affect the bulk of the work needed to be done.

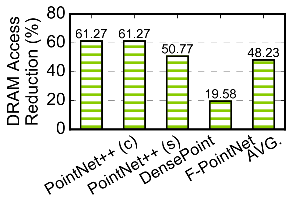
Approximation Degrees Fig. 23a and Fig. 23b show the accuracy-vs-speedup and accuracy-vs-energy trade-offs, respectively, with different h_t and h_e combinations, which dictate different approximation strengths. The data are reported from PointNet++(c), but the trend generally holds. Overall, varying h_t from 0 to 12 and h_e from 4 to 14 provide a trade-off space of about 5% accuracy range, $2.0\times$ performance range, and $1.5\times$ energy range.

7.5 Comparison with Prior Neighbor Search Accelerators

QuickNN [44] and Tigris [66] are two recent neighbor search accelerators that both use a split-tree data structure. As discussed in Sec. 3.4, Crescent reduces both the search load and DRAM traffic.



(a) Reduction in total tree nodes visited from Tigris [66].



(b) DRAM access (in Bytes) reduction from QuickNN [44].

Fig. 24: Comparison with prior neighbor search accelerators.

Fig. 24a shows that the K-d tree-based search reduces the total number of tree nodes visited by 41% compared to exhaustive search. This explains the one order of magnitude performance improvement over the Tigris-based accelerator shown in Sec. 7.2.

QuickNN, similar to Crescent, also presents a completely streaming DRAM accesses — at the expense of redundant DRAM accesses, since each sub-tree is potentially loaded onto the accelerator multiple times. Comparing to a QuickNN implementation with the same PE configuration, Fig. 24b shows that Crescent reduces the total DRAM accesses by 48%.

Finally, we target DNN-based algorithms and, thus, can mitigate the potential accuracy loss through end-to-end network training, which is not available to QuickNN and Tigris; both target a non-DNN algorithm (point cloud registration).

8 RELATED WORK

Deep Learning for Point Clouds Point cloud algorithms are increasingly moving toward DNNs, which has spurred recent interests in accelerating point cloud networks [18, 29, 36]. Point cloud DNNs mainly come in two forms: one that operates on raw points [37, 48, 49, 59, 70], and the other that first voxelizes points and operates on voxels, which are grid-aligned points [16, 22]. The former requires explicitly neighbor search whereas the latter accesses neighbors through simple indexing. It is unclear whether future point cloud algorithms will definitively favor one form over the other. Crescent focuses on optimizing point-based algorithms, whose flexibility and compact data representation are shown to be critical in many application domains [26], such as object detection, localization (SLAM), segmentation, and classification.

PointAcc [36], Point-X [69], and Mesorasi [18] are all recent point cloud accelerators. They are fundamentally orthogonal to our work in that they focus on accelerating the feature computation in point cloud DNNs. For instance, Point-X and Mesorasi exploit the spatial locality and computation redundancy, respectively. All three use brute-force neighbor search and, thus, can directly benefit from the optimizations (approximate neighbor search and selective bank conflict elision) proposed in this paper. We show $1.9\times$ speedup and 36% energy reduction over Mesorasi in Sec. 7.2.

Neighbor Search This paper targets neighbor search in low-dimensional space (2/3D), which is a fundamental building block in many computational science and engineering fields, where physical objects naturally lie in 2/3D space, such as computational fluid dynamics [30], computer graphics [68], and vision [38, 66]. Prior work has explored both algorithmic and hardware solutions to accelerate neighbor search [11, 21, 28, 34, 44, 50, 63, 66], many of which are approximate in their nature [12, 23, 28, 40, 41, 45]. We provide a quantitative comparison with QuickNN [44] and Tigris [66], two most relevant accelerators in Sec. 7.5.

Optimizing Irregular Memory Accesses Recent work has made significant strides in domain-agnostic prefetching for irregular applications [10, 43, 57]. Our split-tree structure can be seen as an application-specific prefetcher and achieves "perfect prefetching" in that 1) off-chip data accesses are overlapped with computation, 2) data needed by the accelerator are readily available on-chip without stall, and 3) no redundant DRAM accesses are needed.

Our split-tree structure also serves as an irregular tiling strategy, akin to propagation blocking for graph algorithms [13], but the decision as to which partition (sub-tree) a point is stored is based on the geometric position of a point.

Approximation Techniques Our approximation techniques exploit the inexact nature of DNNs. Selective bank conflict elision can be seen as a form of value approximation, bearing similarity to such approximation in general-purpose processors [42, 51, 53, 54, 64]. However, different from prior systems where the accuracy control is empirical, we integrate approximation into the training process; this allows us to provide statistical accuracy guarantees.

9 CONCLUSION

The mismatch between 3D perception algorithms and today's hardware designed and optimized for 2D perception will only increase in the future, where 3D perception applications are expected to be much more compute- and memory-intensive while at the same time being deployed in more resource-constrained platforms such as micro aerial vehicles.

The mismatch between 3D perception algorithms and today's hardware designed and optimized for 2D perception will only increase in the future. Crescent demonstrates an algorithm-hardware collaborative approach toward taming the irregularities in point cloud algorithms. The key idea behind Crescent is to intentionally introduce approximations at both the algorithm and the hardware level to reduce memory inefficiencies (e.g., converting random DRAM accesses to streaming accesses, selectively eliding SRAM bank conflicts), and the mitigate the accuracy loss through approximate-aware network retraining.

10 ACKNOWLEDGEMENTS

We thank the anonymous reviewers from ISCA 2022 for their valuable feedback. The work was supported, in part, by NSF under grants #2044963 and #2126642.

REFERENCES

[1] "ARCore," https://developers.google.com/ar.

[2] "ARM's First Generation ML Processor, HotChips 30." [Online]. Available: https://www.hotchips.org/hc30/2conf/2.07_ARM_ML_Processor_HC30_ARM_2018_08_17.pdf

[3] "Artisan Memory Compilers," https://developer.arm.com/ip-products/physical-ip/embedded-memory.

[4] "Jetson TX2 Module," https://developer.nvidia.com/embedded/jetson-tx2.

[5] "Micron 178-Ball, Single-Channel Mobile LPDDR3 SDRAM Features." [Online]. Available: https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/mobile-dram/low-power-dram/lpddr3/178b_8-16gb_2c0f_mobile_lpddr3.pdf

[6] "Micron System Power Calculators." [Online]. Available: https://www.micron.com/support/tools-and-utilities/power-calc

[7] "OpenHeritage 3D dataset," https://www.openheritage3d.org.

[8] "Waymo Offers a Peek Into the Huge Trove of Data Collected by Its Self-Driving Cars," https://spectrum.ieee.org/cars-that-think/transportation/self-driving/waymo-opens-up-part-of-its-humongous-selfdriving-database.

[9] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "Garnet: A detailed on-chip network model inside a full-system simulator," in *2009 IEEE international symposium on performance analysis of systems and software*. IEEE, 2009, pp. 33–42.

[10] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," *ACM Sigplan Notices*, vol. 53, no. 2, pp. 578–592, 2018.

[11] M. Aly, M. Munich, and P. Perona, "Distributed kd-trees for retrieval from very large image collections," in *Proceedings of the British machine vision conference (BMVC)*, vol. 17, 2011.

[12] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 891–923, 1998.

[13] S. Beamer, K. Asanović, and D. Patterson, "Reducing pagerank communication via propagation blocking," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 820–831.

[14] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[15] A. X. Chang, T. Funkhouser, L. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, J. Xiao, L. Yi, and F. Yu, "ShapeNet: An Information-Rich 3D Model Repository," Stanford University — Princeton University — Toyota Technological Institute at Chicago, Tech. Rep. arXiv:1512.03012 [cs.GR], 2015.

[16] C. Choy, J. Gwak, and S. Savarese, "4d spatio-temporal convnets: Minkowski convolutional neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 3075–3084.

[17] L. Comba, A. Biglia, D. R. Aimonino, and P. Gay, "Unsupervised detection of vineyards by 3d point-cloud uav photogrammetry for precision agriculture," *Computers and electronics in agriculture*, vol. 155, pp. 84–95, 2018.

[18] Y. Feng, B. Tian, T. Xu, P. Whatmough, and Y. Zhu, "Mesorasi: Architecture support for point cloud analytics via delayed-aggregation," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1037–1050.

[19] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[20] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Proceedings of the 25th IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

[21] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel, "Buffer kd trees: processing massive nearest neighbor queries on gpus," in *International Conference on Machine Learning*. PMLR, 2014, pp. 172–180.

[22] B. Graham, M. Engelcke, and L. Van Der Maaten, "3d semantic segmentation with submanifold sparse convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 9224–9232.

[23] 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.

[24] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu, "Kilo-noc: a heterogeneous network-on-chip architecture for scalability and service guarantees," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2011, pp. 401–412.

[25] P. Guerrero, Y. Kleiman, M. Ovsjanikov, and N. J. Mitra, "Pcpnet learning local shape properties from raw point clouds," in *Computer Graphics Forum*, vol. 37, no. 2. Wiley Online Library, 2018, pp. 75–85.

[26] Y. Guo, H. Wang, Q. Hu, H. Liu, L. Liu, and M. Bennamoun, "Deep learning for 3d point clouds: A survey," *IEEE transactions on pattern analysis and machine intelligence*, 2020.

[27] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: Compiling high-level image processing code into hardware pipelines," 2014.

[28] S. Heinzle, G. Guennebaud, M. Botsch, and M. H. Gross, "A hardware processing unit for point sets," in *Acm Siggraph/eurographics Symposium on Graphics Hardware*, 2008.

[29] B. Hyun, J. Lee, and M. Rhu, "Characterization and analysis of deep learning for 3d point cloud analytics," *IEEE Computer Architecture Letters*, vol. 20, no. 2, pp. 106–109, 2021.

[30] M. Ihmsen, N. Akinci, M. Becker, and M. Teschner, "A parallel sph implementation on multi-core cpus," in *Computer Graphics Forum*, vol. 30, no. 1. Wiley Online Library, 2011, pp. 99–112.

[31] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, R. C. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proc. of ISCA*, 2017.

[32] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture*, 2017.

[33] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[34] T. Kuhara, T. Miyajima, M. Yoshimi, and H. Amano, *An FPGA Acceleration for the Kd-tree Search in Photon Mapping*, 2013.

[35] M. Liang, B. Yang, S. Wang, and R. Urtasun, "Deep continuous fusion for multi-sensor 3d object detection," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 641–656.

[36] Y. Lin, Z. Zhang, H. Tang, H. Wang, and S. Han, "Pointacc: Efficient point cloud accelerator," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 449–461.

[37] Y. Liu, B. Fan, G. Meng, J. Lu, S. Xiang, and C. Pan, "Densepoint: Learning densely contextual representation for efficient point cloud processing," in *Proceedings of the 14th IEEE International Conference on Computer Vision*, 2019.

[38] Y. Lu, Y. Zhu, and G. Lu, "3d sceneflownet: Self-supervised 3d scene flow estimation based on graph cnn," in *2021 IEEE International Conference on Image Processing (ICIP)*. IEEE, 2021, pp. 3647–3651.

[39] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, 1981.

[40] V. C. Ma and M. D. McCool, "Low latency photon mapping using block hashing," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Eurographics Association, 2002, pp. 89–99.

[41] L. Miclet and M. Dabouz, "Approximative fast nearest-neighbour recognition," *Pattern Recognition Letters*, vol. 1, no. 5-6, pp. 277–285, 1983.

[42] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, "Doppelgänger: a cache for approximate computing," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 50–61.

[43] A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Vector runahead," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 195–208.

[44] R. Pinkham, S. Zeng, and Z. Zhang, "Quicknn: Memory and performance optimization of kd tree based nearest neighbor search for 3d point clouds," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 180–192.

[45] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware," in *ACM SIGGRAPH 2005 Courses*. ACM, 2005, p. 258.

[46] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: balancing efficiency & flexibility in specialized computing," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 24–35, 2013.

[47] C. R. Qi, W. Liu, C. Wu, H. Su, and L. J. Guibas, "Frustum pointnets for 3d object detection from rgb-d data," in *Proceedings of the 31st IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 918–927.

[48] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 652–660.

[49] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "Pointnet++: Deep hierarchical feature learning on point sets in a metric space," in *Advances in neural information processing systems*, 2017, pp. 5099–5108.

[50] D. Qiu, S. May, and A. Nüchter, "Gpu-accelerated nearest neighbor search for 3d registration," in *Proceedings of the 9th International Conference on Computer Vision Systems*, 2009.

[51] P. V. Rengasamy, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "Exploiting staleness for approximating loads on cmps," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 343–354.

[52] R. B. Rusu, Z. C. Marton, N. Blodow, M. Dolha, and M. Beetz, "Towards 3d point cloud based object maps for household environments," *Robotics and Autonomous Systems*, vol. 56, no. 11, pp. 927–941, 2008.

[53] J. San Miguel, J. Albericio, N. E. Jerger, and A. Jaleel, "The bunker cache for spatio-value approximation," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[54] J. San Miguel, M. Badr, and N. E. Jerger, "Load value approximation," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 127–139.

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

[56] J. D. Stets, Y. Sun, W. Corning, and S. W. Greenwald, "Visualization and labeling of point clouds in virtual reality," in *SIGGRAPH Asia 2017 Posters*. ACM, 2017, p. 31.

[57] N. Talati, K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun *et al.*, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 654–667.

[58] G. Vosselman, S. Dijkman *et al.*, "3d building model reconstruction from point clouds and ground plans," *International archives of photogrammetry remote sensing and spatial information sciences*, vol. 34, no. 3/W4, pp. 37–44, 2001.

[59] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic graph cnn for learning on point clouds," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 5, pp. 1–12, 2019.

[60] N. H. Weste and D. Harris, *CMOS VLSI design: a circuits and systems perspective*. Pearson Education India, 2015.

[61] P. N. Whatmough, C. Zhou, P. Hansen, S. K. Venkataramanaiah, J.-s. Seo, and M. Mattina, "Fixynn: Efficient hardware for mobile computer vision via transfer learning," *arXiv preprint arXiv:1902.11128*, 2019.

[62] M. Whitty, S. Cossell, K. S. Dang, J. Guivant, and J. Katupitiya, "Autonomous navigation using a real-time 3d point cloud," in *2010 Australasian Conference on Robotics and Automation*, 2010.

[63] F. Winterstein, S. Bayliss, and G. A. Constantinides, "Fpga-based k-means clustering using tree-based data structures," in *International Conference on Field Programmable Logic & Applications*, 2013.

[64] D. Wong, N. S. Kim, and M. Annavaram, "Approximating warps with intra-warp operand value similarity," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 176–187.

[65] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao, "3d shapenets: A deep representation for volumetric shapes," in *Proceedings of the 28th IEEE conference on computer vision and pattern recognition*, 2015.

[66] T. Xu, B. Tian, and Y. Zhu, "Tigris: Architecture and algorithms for 3d perception in point clouds," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 629–642.

[67] A. Yazdanbakhsh, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "Ganax: A unified mimd-simd acceleration for generative adversarial networks," in *Proceedings of the 45th ACM/IEEE Annual International Symposium on Computer Architecture*, 2018.

[68] W. Yifan, F. Serena, S. Wu, C. Öztireli, and O. Sorkine-Hornung, "Differentiable surface splatting for point-based geometry processing," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 6, pp. 1–14, 2019.

[69] J.-F. Zhang and Z. Zhang, "Point-x: A spatial-locality-aware architecture for energy-efficient graph-based point-cloud deep learning," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1078–1090.

[70] K. Zhang, M. Hao, J. Wang, C. W. de Silva, and C. Fu, "Linked dynamic graph cnn: Learning on point cloud via linking hierarchical features," *arXiv preprint arXiv:1904.10014*, 2019.

[71] Y. Zhou, M. Yang, C. Guo, J. Leng, Y. Liang, Q. Chen, M. Guo, and Y. Zhu, "Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2021, pp. 214–225.

[72] Y. Zhu, R. Mottaghi, E. Kolve, J. J. Lim, A. Gupta, L. Fei-Fei, and A. Farhadi, "Target-driven visual navigation in indoor scenes using deep reinforcement learning," in *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2017, pp. 3357–3364.