Mesorasi: Architecture Support for Point Cloud Analytics via Delayed-Aggregation

Yu Feng*
University of Rochester
yfeng28@ur.rochester.edu

Boyuan Tian*
University of Rochester
btian2@ur.rochester.edu

Tiancheng Xu*
University of Rochester
txu17@ur.rochester.edu

Paul Whatmough
Arm Research
paul.whatmough@arm.com

Yuhao Zhu
University of Rochester
yzhu@rochester.edu

horizon-lab.org

***Abstract*—Point cloud analytics is poised to become a key workload on battery-powered embedded and mobile platforms in a wide range of emerging application domains, such as autonomous driving, robotics, and augmented reality, where efficiency is paramount. This paper proposes Mesorasi, an algorithm-architecture co-designed system that simultaneously improves the performance and energy efficiency of point cloud analytics while retaining its accuracy.**

Our extensive characterizations of state-of-the-art point cloud algorithms show that, while structurally reminiscent of convolutional neural networks (CNNs), point cloud algorithms exhibit inherent compute and memory inefficiencies due to the unique characteristics of point cloud data. We propose *delayed-aggregation*, a new algorithmic primitive for building efficient point cloud algorithms. Delayed-aggregation hides the performance bottlenecks and reduces the compute and memory redundancies by exploiting the *approximately distributive* property of key operations in point cloud algorithms. Delayed-aggregation let point cloud algorithms achieve 1.6× speedup and 51.1% energy reduction on a mobile GPU while retaining the accuracy (-0.9% loss to 1.2% gains). To maximize the algorithmic benefits, we propose minor extensions to contemporary CNN accelerators, which can be integrated into a mobile Systems-on-a-Chip (SoC) without modifying other SoC components. With additional hardware support, Mesorasi achieves up to 3.6× speedup.

***Index Terms*—Point cloud; DNN; accelerator;**

***Artifact*—https://github.com/horizon-research/efficient-deep-learning-for-point-clouds**

I. Introduction

In recent years, we have seen the explosive rise of intelligent machines that operate on *point clouds*, a fundamental visual data representation that provides a direct 3D measure of object geometry, rather than 2D projections (i.e., images). For instance, Waymo's self-driving cars carry five LiDAR sensors to gather point clouds from the environment in order to estimate the trajectory over time and to sense object depths [15]. Augmented Reality (AR) development frameworks such as Google's ARCore enable processing point clouds for localization (SLAM) and scene understanding [2]. While point cloud algorithms traditionally use "hand-crafted" features [48], [51], they are increasingly moving towards learned features in deep learning [43], [53], posing efficiency challenges.

We present Mesorasi[1], an algorithm-architecture co-designed system that simultaneously improves the performance and energy efficiency of point cloud algorithms without hurting the accuracy. Mesorasi applies algorithmic and architectural optimizations that exploit characteristics unique to point cloud. Critically, our algorithmic optimizations can directly benefit software running on commodity GPUs without hardware support. Minor augmentations to contemporary DNN accelerators (NPU) unlock more gains and widen the applicability.

We start by understanding the characteristics of point cloud algorithms. They inherit the key idea of conventional image/video processing algorithms (e.g., CNNs): extracting features from local windows (neighborhoods) iteratively and hierarchically until the final output is calculated. However, since points in a point cloud are arbitrarily spread in the 3D space, point cloud algorithms require explicit neighbor search and point aggregation operations (as opposed to direct memory indexing) before the actual feature computation.

This leads to two fundamental inefficiencies. First, the three key steps—neighbor search, aggregation, and feature computation—are serialized, leading to long critical path latency. In particular, neighbor search and feature computation dominate the execution time. Second, feature computation operates on aggregated neighbor points, which are inherently redundant representations of the original points, leading to massive memory and computation redundancies.

We propose *delayed-aggregation*, a new algorithmic primitive for building efficient point cloud networks. The key idea is to delay aggregation after feature computation by exploiting the *approximately distributive* property of feature computation over aggregation. In this way, feature computation operates directly on original input points rather aggregated neighbors, significantly reducing the compute cost and memory accesses. In addition, delayed-aggregation breaks the serialized execution

*Equal contribution

[1] [me-sˈəra-zē] Between two vision modes. *meso-*: in the middle; from Ancient Greek μέσος. *orasi*: vision; from Greek όραση.

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00087

chain in existing algorithms, overlapping neighbor search and feature computation—the two performance bottlenecks—to hide long latencies.

To maximize the benefits of delayed-aggregation, we propose minor extensions to conventional DNN accelerators. We find that delayed-aggregation increases the overhead of aggregation, which involves irregular gather operations. The hardware extension co-designs an intelligent data structure partitioning strategy with a small but specialized memory unit to enable efficient aggregation. Our hardware extensions are integrated into generic DNN accelerators without affecting the rest of a mobile Systems-on-a-Chip (SoC).

We evaluate Mesorasi on a set of popular point cloud algorithms and datasets. On the mobile Pascal GPU on TX2, a representative mobile platform today, the delayed-aggregation algorithm alone without hardware support achieves 1.6× speedup and 51.1% energy reduction while retaining the accuracy (-0.9% loss to 1.2% gains). We implement and synthesize the Mesorasi hardware support in a 16nm process node and integrate it into a state-of-the-art SoC that incorporates a GPU and an NPU. With 3.8% area overhead to the NPU (<0.05% of a typical SoC area), Mesorasi achieves up to 3.6× speedup, which increases to 6.7 on a futuristic SoC with a dedicated neighbor search accelerator.

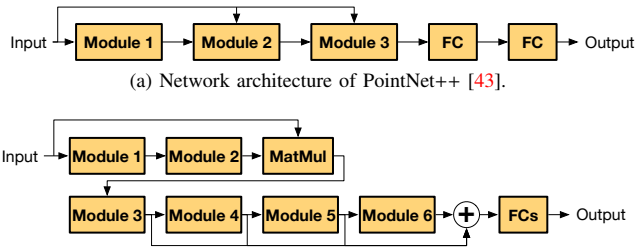
The artifact is publicly available at https://github.com/horizon-research/efficient-deep-learning-for-point-clouds. In summary, this paper makes the following contributions:

- We comprehensively characterize the performance bottlenecks as well as the memory and compute cost of state-of-the-art point cloud algorithms, and identify the root-causes of the algorithmic inefficiencies.
- We propose delayed-aggregation, an efficient algorithm primitive that enables point cloud algorithms to hide the performance bottlenecks and to reduce the overall workload. Delayed-aggregation can readily achieve significant speedup and energy savings on current-generation mobile GPUs without hardware modification.
- We co-design hardware with delayed-aggregation to achieve even greater speedups with minor, yet principled, augmentations to conventional DNN accelerators while retaining the modularity of existing SoCs.

II. Background

Point Cloud A point cloud is an unordered set of points in the 3D Cartesian space. Each point is uniquely identified by its $< x, y, z >$ coordinates. While point cloud has long been used as a fundamental visual data representation in fields such as 3D modeling [16] and graphics rendering [26], [33], [40], [47], it has recently received lots of attention in a range of emerging intelligent systems such as autonomous vehicles [24], robotics [55], and AR/VR devices [50].

Point Cloud Analytics Similar to conventional visual analytics that analyzes images and videos, point cloud analytics distill semantics information from point clouds. Examples include object detection [24], semantics segmentation [17], and classification [58]. While image and video analytics have been well-optimized, point cloud analytics require different algorithms and are much less optimized.



(a) Network architecture of PointNet++ [43].

(b) DGCNN [53] network architecture. "+" is tensor concatenation.

Fig. 1: Point cloud networks consist of a set of modules, which extract local features from the input point cloud iteratively and hierarchically to calculate the final output.

Point cloud algorithms operate by iteratively extracting features of each point. Conventional point cloud algorithms use "hand-crafted" features such as FPFH [48] and SHOT [51]. Recent deep learning-based algorithms use learned features and have generally out-performed conventional algorithms [20]. This paper thus focuses on deep learning-based algorithms.

We focus on deep learning-based algorithms that directly manipulate raw point clouds. Other data representations such as 2D projections of 3D points and voxelization suffer from low accuracy and/or consume excessively high memory [35].

III. Motivation

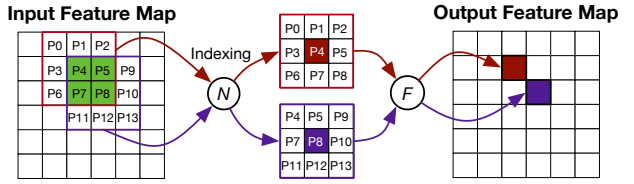
We first introduce the general flow of point cloud algorithms and identify key operators (Sec. III-A). We then characterize point cloud algorithms on today's hardware systems to understand the algorithmic and execution bottlenecks (Sec. III-B), which motivate the Mesorasi design.

III-A. Point Cloud Network Architecture

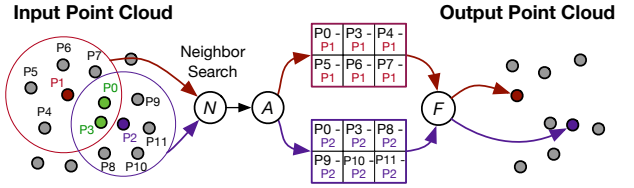
Module The key component in point cloud algorithms is a *module*. Each module transforms an input point cloud to an output point cloud, similar to how a convolution *layer* transforms an input feature map to an output feature map in conventional CNNs. A point cloud network assembles different modules along with other common primitives such as fully-connected (FC) layers. Fig. 1a and Fig. 1b illustrate the architecture of two representative point cloud networks, PointNet++ [43] and DGCNN [53], respectively.

Module Internals Each point **p** in a point cloud is represented by a feature vector, which in the original point cloud is simply the 3D coordinates of the point. The input point cloud to a module is represented by an $N_{in} \times M_{in}$ matrix, where N_{in} denotes the number of input points and M_{in} denotes the input feature dimension. Similarly, the output point cloud is represented by an $N_{out} \times M_{out}$ matrix, where N_{out} denotes the number of output points and M_{out} denotes the output feature dimension. Note that N_{in} and N_{out} need not be the same; neither do M_{in} and M_{out}.

Internally, each module extracts local features from the input point cloud. This is achieved by iteratively operating on a small *neighborhood* of input points, similar to how a convolution



(a) Convolution in conventional CNNs can be thought of as two steps: 1) neighbor search ($\mathcal{N}$) by directly indexing adjacent pixels and 2) feature computation ($\mathcal{F}$) by a dot product.



(b) Point cloud networks consist of three main steps: neighbor search ($\mathcal{N}$), aggregation ($\mathcal{A}$), and feature computation ($\mathcal{F}$). $\mathcal{N}$ requires an explicit neighbor search; $\mathcal{A}$ normalizes neighbors to their centroid; $\mathcal{F}$ is an MLP with batched inputs (i.e., shared MLP weights).

Fig. 2: Comparing a convolution layer in conventional CNNs and a module in point cloud networks.

layer extracts local features of the input image through a sliding window. Fig. 2 illustrates this analogy.

Specifically, each output point $\mathbf{p}_o$ is computed from an input point $\mathbf{p}_i$ in three steps — neighbor search ($\mathcal{N}$), aggregation ($\mathcal{A}$), and feature computation ($\mathcal{F}$):

$$\mathbf{p}_o = \mathcal{F}(\mathcal{A}(\mathcal{N}(\mathbf{p}_i), \quad \mathbf{p}_i)) \tag{1}$$

where $\mathcal{N}$ returns K neighbors of $\mathbf{p}_i$, $\mathcal{A}$ aggregates the K neighbors, and $\mathcal{F}$ operates on the aggregation ($\mathbf{p}_i$ and its K neighbors) to generate the output $\mathbf{p}_o$.

The same formulation applies to the convolution operation in conventional CNNs as well, as illustrated in Fig. 2. However, the specifics of the three operations differ in point cloud networks and CNNs. Understanding the differences is key to identifying optimization opportunities.

Neighbor Search $\mathcal{N}$ in convolution returns K adjacent pixels in a regular 3D tensor by simply *indexing* the input feature map (K dictated by the convolution kernel volume). In contrast, $\mathcal{N}$ in point cloud networks requires explicit *neighbor search* to return the K nearest neighbors of $\mathbf{p}_i$, because the points are irregularly scattered in the space. Similar to the notion of a "stride" in convolution, the neighbor search might be applied to only a subset of the input points, in which case N_{out} would be smaller than N_{in}, as is the case in Fig. 2b.

Aggregation Given the K pixels, convolution in CNNs directly operates on the raw pixel values. Thus, conventional convolution skips the aggregation step.

In contrast, point cloud modules operate on the *relative* value of each point in order to correlate a neighbor with its centroid. For instance, a point $\mathbf{p}_3$ could be a neighbor of two centroids $\mathbf{p}_1$ and $\mathbf{p}_2$ (as is the case in Fig. 2b). To differentiate the different contributions of $\mathbf{p}_3$ to $\mathbf{p}_1$ and $\mathbf{p}_2$, $\mathbf{p}_3$ is *normalized* to the two centroids by calculating the offsets $\mathbf{p}_3 - \mathbf{p}_1$ and $\mathbf{p}_3 - \mathbf{p}_2$ for subsequent computations.

Generally, for each neighbor $\mathbf{p}_k \in \mathcal{N}(\mathbf{p}_i)$, the aggregation operation calculates the offset $\mathbf{p}_k - \mathbf{p}_i$ (a $1 \times M_{in}$ vector). All K neighbors' offsets form a Neighbor Feature Matrix (NFM) of size $K \times M_{in}$, effectively aggregating the neighbors of $\mathbf{p}_i$.

Feature Computation $\mathcal{F}$ in convolution is a dot product between the pixel values in a window and the kernel weights. In contrast, $\mathcal{F}$ in point cloud applies a multilayer perceptron (MLP) to each row vector in the NFM. Critically, all K row vectors share the same MLP; thus, the K input vectors are batched into a matrix and the MLP becomes a matrix-matrix product, transforming a $K \times M_{in}$ matrix to a $K \times M_{out}$ matrix.

In the end, a reduction operation then reduces the $K \times M_{out}$ matrix to a $1 \times M_{out}$ vector, which becomes the feature vector of an output point. A common choice for reduction is to, for each column independently, take the max of the K rows.

Example Fig. 3 shows the first module in PointNet++ [43], a classic point cloud network that many other networks build upon. This module transforms a point cloud with 1024 (N_{in}) points, each with a 3-D (M_{in}) feature vector, to a point cloud with 512 (N_{out}) points, each with an 128-D (M_{out}) feature vector, indicating that the neighbor search is applied to only 512 input points. Each neighbor search returns 32 (K) neighbors and forms a 32×3 NFM, which is processed by a MLP with 3 layers to generate a 32×128 matrix, which in turn is reduced to a 1×128 feature vector for an output point. In this particular network, all the NFMs also share the same MLP.

Note that while feature computation is not always MLP and normalization is not always differencing from centroids, they are the most widely used, both in classic networks (e.g., PointNet++ [43]) and recent ones (e.g., DGCNN [53]).

III-B. Performance Characterizations

We characterize point cloud networks on today's systems to understand the bottlenecks and optimization opportunities. To that end, we profile the performance of five popular point cloud networks on the mobile Pascal GPU on the Jetson TX2 development board [10], which is representative of state-of-the-art mobile computing platforms. Please refer to Sec. VI for a detailed experimental setup.

Time Distribution Fig. 4 shows the execution times of the five networks, which range from 71 *ms* to 5,200 *ms*, clearly infeasible for real-time deployment. The time would scale proportionally as the input size grows.

Fig. 5 further decomposes the execution time into the three components, i.e., Neighbor Search ($\mathcal{N}$), Aggregation ($\mathcal{A}$), and Feature Computation ($\mathcal{F}$). $\mathcal{N}$ and $\mathcal{F}$ are the major performance bottlenecks. While $\mathcal{F}$ consists of MLP operations that are well-optimized, $\mathcal{N}$ (and $\mathcal{A}$) is uniquely introduced in point cloud networks. Even if $\mathcal{F}$ could be further accelerated on a DNN accelerator, $\mathcal{N}$ has compute and data access patterns different from matrix multiplications [59], and thus does not fit on a DNN accelerator.

Critically, $\mathcal{N}$, $\mathcal{A}$, and $\mathcal{F}$ are serialized. Thus, they all contribute to the critical path latency; optimizing one alone

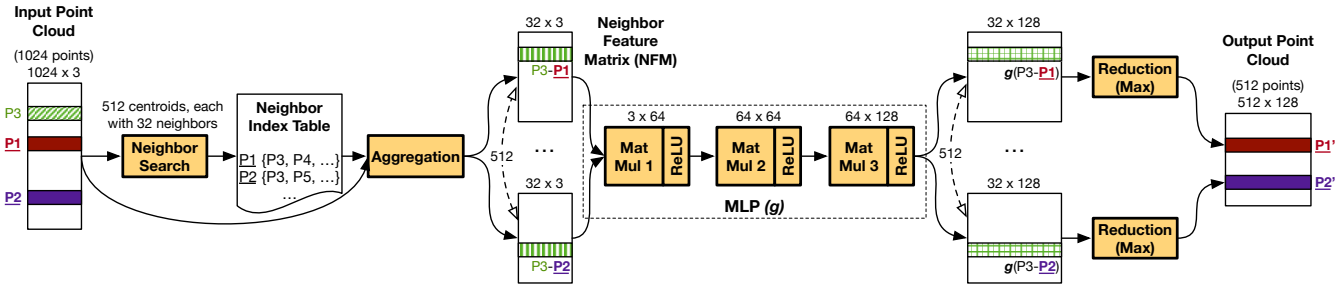


Fig. 3: The first module in PointNet++ [43]. The same MLP is shared across all the row vectors in a Neighbor Feature Matrix (NFM) and also across different NFMs. Thus, MLPs in point cloud networks process batched inputs, effectively performing matrix-matrix multiplications. The (shared) MLP weights are small in size, but the MLP activations are much larger. This is because the same input point is normalized to different values in different neighborhoods before entering the MLP. For instance, P3 is normalized to different offsets with respect to P1 and P2 as P3 is a neighbor of both P1 and P2. In point cloud algorithms, most points are normalized to 20 to 100 centroids, proportionally increasing the MLP activation size.

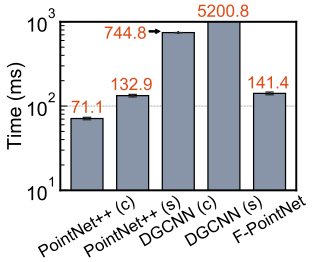


Fig. 4: Latency of five point cloud networks on the Pascal GPU on TX2. Results are averaged over 100 executions, and the error bars denote one standard deviation.

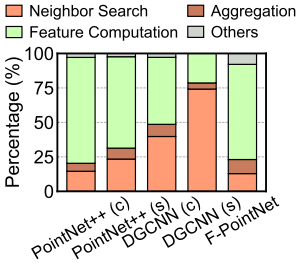


Fig. 5: Time distribution across the three main point cloud operations ($\mathcal{N}$, $\mathcal{A}$, and $\mathcal{F}$). The data is averaged on the mobile Pascal GPU on TX2 over 100 executions.

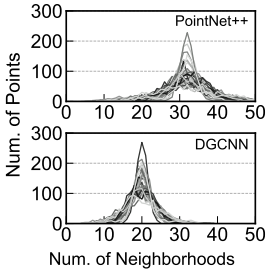


Fig. 6: Distribution of the number of points (y-axis) that occur in a certain number of neighborhoods (x-axis). We profile 32 inputs (curves).

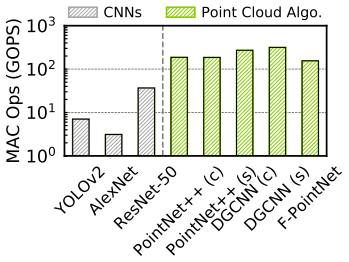


Fig. 7: MAC operation comparison between point cloud networks (130K input points per frame [24]) and conventional CNNs (nearly 130K pixels per frame).

would not lead to universal speedups. The serialization is inherent to today's point cloud algorithms: in order to extract local features of a point ($\mathcal{F}$), the point must be aggregated with its neighbors ($\mathcal{A}$), which in turn requires neighbor search ($\mathcal{N}$). Mesorasi's algorithm breaks this serialized execution chain, allowing $\mathcal{F}$ and $\mathcal{N}$ to be overlapped.

Memory Analysis Point cloud networks have large memory footprints. While the MLP weights are small and are shared across input NFMs (Fig. 3), the intermediate (inter-layer) activations in the MLP are excessive in size.

The "Original" category in Fig. 10 shows the distribution of each MLP layer's output size across the five networks. The data is shown as a violin plot, where the high and low ticks represent the largest and smallest layer output size, respectively, and the width of the violin represents the density at a particular size value (y-axis). The layer output usually exceeds 2 MB, and could be as large as 32 MB, much greater than a typical on-chip memory size in today's mobile GPUs or DNN accelerators. The large activation sizes would lead to frequent DRAM accesses and high energy consumption.

The large activation size is fundamental to point cloud algorithms. This is because an input point usually belongs to many overlapped neighborhoods, and thus must be normalized to different values, one for each neighborhood. Fig. 2b shows a concrete example, where P3 is a neighbor of both P1 and P2; the aggregation operation normalizes P3 to P1 and P2, leading to two different relative values (P3 - P1 and P3 - P2) that participate in feature computation, increasing the activation size. This is in contrast to convolutions, where pixels in overlapped neighborhoods (windows) are directly reused in feature computation (e.g., P4 in Fig. 2a).

We use two networks, DGCNN [43] and PointNet++ [53], to explain the large activation sizes. Fig. 6 shows the distribution of the number of neighborhoods each point is in. Each curve corresponds to an input point cloud, and each (x, y) point on a curve denotes the number of points (y) that occur in a certain number of neighborhoods (x). In PointNet++, over half occur in more than 30 neighborhoods; in DGCNN, over half occurs in 20 neighborhoods. Since the same point is normalized to different values in different neighborhoods, this bloats the MLP's intermediate activations.

Compute Cost The large activations lead to high multiply-and-accumulate (MACs) operations. Fig. 7 compares the number of MAC operations in three classic CNNs with that in

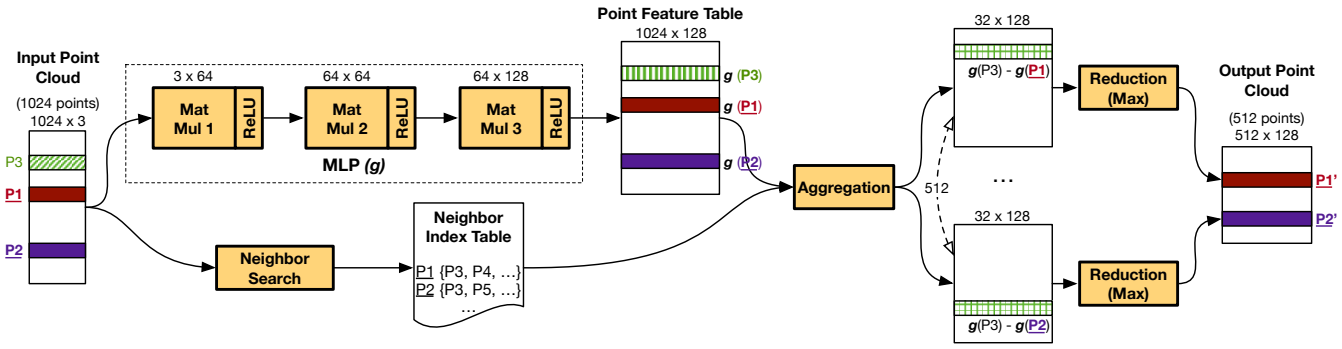


Fig. 8: The delayed-aggregation algorithm applied to the first module in PointNet++. The MLP and neighbor search are executed in parallel, effectively delaying aggregation after feature computation. The input size of the MLP is much smaller (input point cloud as opposed to the aggregated NFMs), which significantly reduces the MAC operations and the intermediate activation sizes. Aggregation now operates on the output feature space (128-D in this case), whereas it previously operates on the input feature space (3-D in this case). Thus, the aggregation time increases and emerges as a new performance bottleneck.

the feature computation of point cloud networks. To use the same "resolution" for a fair comparison, the input point cloud has 130,000 points (e.g., from the widely-used KITTI Odometry dataset [24]) and the CNN input has a similar amount of pixels. In feature computation alone, point cloud networks have an order of magnitude higher MAC counts than conventional CNNs. Mesorasi's algorithm reduces both the memory accesses and MAC counts in feature computation.

Summary Today's point cloud algorithms extract local features of a point by aggregating the point with its neighbors. The aggregation happens *before* feature computation, which leads to two fundamental inefficiencies:

- The two major performance bottlenecks, neighbor search and feature computation, are serialized.
- Feature computation operates on aggregated neighbor points, leading to high memory and compute cost.

IV. Delayed-Aggregation Algorithm

We introduce delayed-aggregation, a primitive for building efficient point cloud networks (Sec. IV-A). Delayed-aggregation improves the compute and memory efficiencies of point cloud networks without degrading accuracy (Sec. IV-B). We show that aggregation emerges as a new bottleneck in new networks, motivating dedicated hardware support (Sec. IV-C).

IV-A. Algorithm

We propose a new framework for building efficient point cloud algorithms. The central idea is to delay aggregation until *after* feature computation so that features are extracted on individual input points rather than on aggregated neighbors. Delayed-aggregation has two benefits. First, it allows neighbor search and feature computation, the two time-consuming components, to be executed in parallel. Second, feature computation operates on input points rather than aggregated neighbors, reducing the compute and memory costs.

Delayed-Aggregation The key insight is that feature extraction ($\mathcal{F}$) is *approximately distributive* over aggregation ($\mathcal{A}$). For an input point $\mathbf{p}_i$ and its corresponding output $\mathbf{p}_o$:

$$\mathbf{p}_o = \mathcal{F}(\mathcal{A}(\mathcal{N}(\mathbf{p}_i),\ \ \mathbf{p}_i)) \approx \mathcal{A}(\mathcal{F}(\mathcal{N}(\mathbf{p}_i)),\ \mathcal{F}(\mathbf{p}_i)) \qquad (2)$$

Fundamentally, Equ. 2 holds because the MLP in $\mathcal{F}$ is approximately distributive over subtraction in $\mathcal{A}$. Specifically, applying an MLP to the difference of two matrices is approximately equivalent to applying an MLP to both matrices and then subtract the two resulting matrices. The approximation is introduced by the non-linearity in the MLP (e.g., ReLU):

$$\phi(\phi(\begin{bmatrix}\mathbf{p}_1-\mathbf{p}_i\\ \dots\\ \mathbf{p}_k-\mathbf{p}_i\end{bmatrix}\times W_1)\times W_2) \approx$$
$$\phi(\phi(\begin{bmatrix}\mathbf{p}_1\\ \dots\\ \mathbf{p}_k\end{bmatrix}\times W_1\times W_2)) - \phi(\phi(\begin{bmatrix}\mathbf{p}_i\\ \dots\\ \mathbf{p}_i\end{bmatrix}\times W_1\times W_2)) \qquad (3)$$

where $\mathbf{p}_1,...,\mathbf{p}_k$ are neighbors of $\mathbf{p}_i$, W_1 and W_2 are the two weight matrices in the MLP (assuming one hidden layer), and ϕ is the non-linear activation function. Without ϕ, the distribution of MLP over subtraction is precise. In actual implementation, the computation on $[\mathbf{p}_i\ \ \dots\ \ \mathbf{p}_i]^\intercal$ is simplified to operating on $\mathbf{p}_i$ once and scattering the result K times.

Critically, applying this distribution allows us to decouple $\mathcal{N}$ with $\mathcal{F}$. As shown in Equ. 2 and Equ. 3, $\mathcal{F}$ now operates on original input points, i.e., $\mathbf{p}_i$ and $\mathcal{N}(\mathbf{p}_i)$ (a subset of the input points, too) rather than the normalized point values ($\mathbf{p}_k - \mathbf{p}_i$), which requires neighbor search results. As a result, we could first apply feature computation on all input points. The computed features are then aggregated later.

Walk-Through We use the first module in PointNet++ as an example to walk through the new algorithm. This module consumes 1024 (N_{in}) input points, among which 512 undergo neighbor search. Thus, the module produces 512 (N_{out}) output points. The input feature dimension is 3 (M_{in}) and the output feature dimension is 128 (M_{out}). Fig. 8 shows this module implemented with delayed-aggregation.

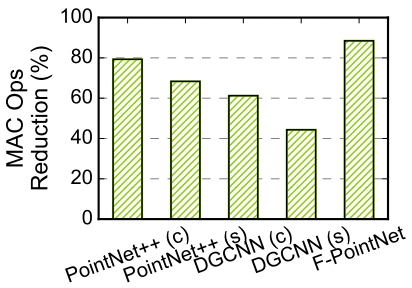


Fig. 9: MAC operation reduction in the MLP by delayed-aggregation. The MAC count reductions come from directly operating on the input points as opposed to aggregated neighbors.

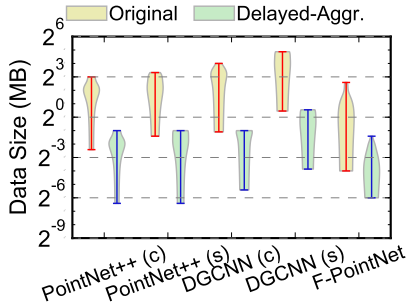


Fig. 10: Layer output size distribution as a violin plot with and without delayed-aggregation. High and low ticks denote the largest and smallest layer outputs, respectively.

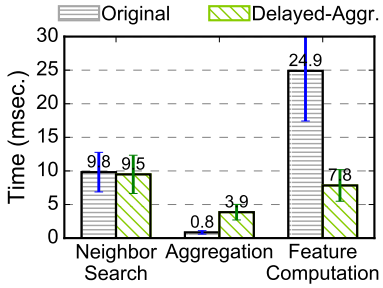


Fig. 11: Time distribution across $\mathcal{N}$, $\mathcal{A}$, and $\mathcal{F}$ in PointNet++ (s) with and without delayed-aggregation. Note that delayed-aggregation would also allow $\mathcal{N}$ and $\mathcal{F}$ to be executed in parallel.

We first compute features ($\mathcal{F}$) from all 1024 points in the input point cloud and store the results in the Point Feature Table (PFT), a 1024×128 matrix. Every PFT entry contains the feature vector of an input point. Meanwhile, neighbor searches ($\mathcal{N}$) are executed in parallel on the input point cloud, each returning 32 neighbors of a centroid. The results of neighbor search are stored in a Neighbor Index Table (NIT), a 512×32 matrix. Each NIT entry contains the neighbor indices of an input point. In the end, the aggregation operation ($\mathcal{A}$) aggregates features in the PFT using the neighbor information in the NIT. Note that it is the features that are being aggregated, not the original points.

Each aggregated matrix (32×128) is reduced to the final feature vector (1×128) of an output point. If reduction is implemented by a max operation as is the common case, aggregation could further be delayed after reduction because subtraction is distributive over max: $max(\mathbf{p}_1-\mathbf{p}_i,\mathbf{p}_2-\mathbf{p}_i)=max(\mathbf{p}_1,\mathbf{p}_2)-\mathbf{p}_i$. This optimization avoids scattering $\mathbf{p}_i$, reduces the subtraction cost, and is mathematically precise.

IV-B. First-Order Efficiency Analysis

Compared with the original implementation of the same module in Fig. 3, the delayed-aggregation algorithm provides three benefits. First, neighbor search and the MLP are now executed in parallel, hiding the latencies of the slower path.

Second, we significantly reduce the MAC operations in the MLP. In this module, the original algorithm executes MLP on 512 32×3 matrices while the new algorithm executes MLP only on one 1024×3 matrix. Fig. 9 shows the MAC operation reductions across all five networks. On average, delayed-aggregation reduces the MAC counts by 68%.

Third, delayed-aggregation also reduces the memory traffic because the MLP input is much smaller. While the actual memory traffic reduction is tied to the hardware architecture, as a first-order estimation Fig. 10 compares the distribution of per-layer output size with and without delayed-aggregation. The data is shown as a violin plot. Delayed-aggregation reduces the layer output sizes from 8 MB˜32 MB to 512 KB˜1 MB, amenable to be buffered completely on-chip.

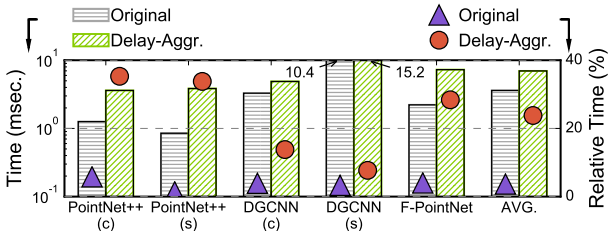


Fig. 12: Both absolute (left y-axis) and relative (right y-axis) aggregation times increase with delayed-aggregation.

By directly extracting features from the input points, our algorithm unlocks the inherent data reuse opportunities in point cloud. Specifically in this example, P3 is a neighbor of both P1 and P2, but could not be reused in feature computation by the original algorithm because P3's normalized values with respect to P1 and P2 are different. In contrast, the MLP in our algorithm directly operates on P1, whose feature is then reused in aggregation, *implicitly* reusing P1.

IV-C. Bottleneck Analysis

While delayed-aggregation reduces the compute costs and memory accesses, it also significantly increases the aggregation time. Using PointNet++ as an example, Fig. 11 compares the execution time distribution across the three operations ($\mathcal{N}$, $\mathcal{A}$, and $\mathcal{F}$) with and without delayed-aggregation. The error bars denote one standard deviation in the measurement. The feature extraction time significantly decreases, and the neighbor search time roughly stays the same — both are expected. The aggregation time, however, significantly increases.

Fig. 12 generalizes the conclusion across the five networks. The figure compares the absolute (left y-axis) and relative (right y-axis) aggregation time in the original and new algorithms. The aggregation time consistently increases in all five networks. Since neighbor search and feature computation are now executed in parallel, aggregation overhead contributes even more significantly to the overall execution time. On average, the aggregation time increases from 3% to 24%.

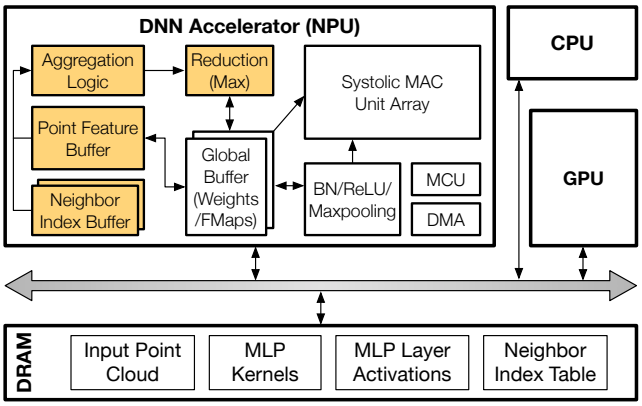


Fig. 13: The MESORASI SoC builds on top of today's SoCs consisting of a GPU and a DNN accelerator (NPU). Neighbor search executes on the GPU and feature extraction executes on the NPU. MESORASI augments the NPU with an aggregation unit (AU) to efficiently execute the aggregation operation. The AU structures are shaded (colored).

Aggregation time increases mainly because aggregation involves irregular gather operations [30], which now operate on a much larger working set with delayed-aggregation. For instance, in PointNet++'s first module (Fig. 8), aggregation originally gathers from a 12 KB matrix but now gathers from a 512 KB matrix, which is much larger than the L1 cache size (48 KB – 96 KB[2]) in the mobile Pascal GPU on TX2.

The working set size increases significantly because aggregation in new algorithms gathers data from the PFT, whose dimension is $N_{in} \times M_{out}$, whereas the original algorithms gather data from the input point matrix, whose dimension is $N_{in} \times M_{in}$. M_{out} is usually several times greater than M_{in} in order to extract higher-dimensional features. In the example above, M_{out} is 128-D whereas M_{in} is 3-D.

V. ARCHITECTURAL SUPPORT

This section describes MESORASI, our hardware design that efficiently executes point cloud algorithms developed using delayed-aggregation. MESORASI extends existing DNN accelerators with minor augmentations while leaving the rest of the SoC untouched. We start from an overview of MESORASI and its workflow (Sec. V-A), followed by a detailed description of the architecture support (Sec. V-B).

V-A. Overall Design

We assume a baseline SoC that incorporates a GPU and an NPU, as with emerging mobile SoCs such as Nvidia Xavier [11], Apple A13 [1], and Microsoft HPU [14]. Point cloud algorithms are a few times faster when an NPU is available to accelerate MLP compared to running only on the GPU (Sec. VII-D). Thus, an NPU-enabled SoC represents the trend of the industry and is a more optimized baseline.

[2]To our best knowledge, Nvidia does not publish the L1 cache size for the mobile Pascal GPU in TX2 (GP10B [9]). We estimate the size based on the L1 cache size per SM in other Pascal GPU chips [13] and the number of SMs in the mobile Pascal GPU [10]

Design Fig. 13 shows how MESORASI augments the NPU in a generic SoC. In MESORASI, the GPU executes neighbor search ($\mathcal{N}$) and the NPU executes feature extraction ($\mathcal{F}$), i.e., the MLP. In addition, MESORASI augments the NPU with an Aggregation Unit (AU) to efficiently execute the aggregation operation ($\mathcal{A}$). As shown in Sec. IV-C, aggregation becomes a bottleneck in our new algorithms and is inefficient on the GPU. AU minimally extends a generic NPU architecture with a set of principled memory structures and datapaths.

MESORASI maps $\mathcal{N}$ to the GPU because neighbor search is highly parallel, but does not map to the specialized datapath of an NPU. Alternatively, an SoC could use a dedicated neighbor search engine (NSE) [31], [59]. We use the GPU because it is prevalent in today's SoCs and thus provides a concrete context to describe our design. We later show that delayed-aggregation could achieve even higher speedups in a futurist SoC where an NSE is available to accelerate neighbor search (Sec. VII-E). In either case, MESORASI does not modify the internals of the GPU or the NSE.

Work Flow Point cloud algorithms with delayed-aggregation work on MESORASI as follows. The input point cloud is initially stored in the DRAM. The CPU configures and triggers the GPU and the NPU simultaneously, both of which read the input point cloud. The GPU executes the KNN search and generates the Neighbor Index Table (NIT), which gets stored back to the DRAM. Meanwhile, the NPU computes features for input points and generates the Point Feature Table (PFT). The AU in NPU combines the PFT with the NIT from the memory for aggregation and reduction, and eventually generates the output of the current module.

In some algorithms (e.g., PointNet++), neighbor searches in all modules search in the original 3-D coordinate space, while in other algorithms (e.g., DGCNN) the neighbor search in module i searches in the output feature space of module $(i-1)$. In the latter case, the current module's output is written back to the memory for the GPU to read in the next module.

Our design modifies only the NPU while leaving other SoC components untouched. This design maintains the modularity of existing SoCs, broadening the applicability. We now describe the AU augmentation in NPU in detail.

V-B. Aggregation Unit in NPU

Aggregation requires irregular gather operations that are inefficient on GPUs. The key to our architectural support is the specialized memory structures co-designed with customized data structure partitioning, which provide efficient data accesses for aggregation with a little area overhead.

Algorithmically, aggregation iterates over the NIT's N_{out} entries until NIT is exhausted. Each NIT entry contains the K neighbor indices of a centroid $\mathbf{p}$. The aggregation operation first gathers the K corresponding entries (feature vectors) from the PFT ($N_{in} \times M_{out}$). The K feature vectors are then reduced to one ($1 \times M_{out}$) vector, which subtracts $\mathbf{p}$'s feature vector to generate the output feature for $\mathbf{p}$.

Fig. 14 shows the detailed design of the aggregation unit. The NIT is stored in an SRAM, which is doubled-buffered in

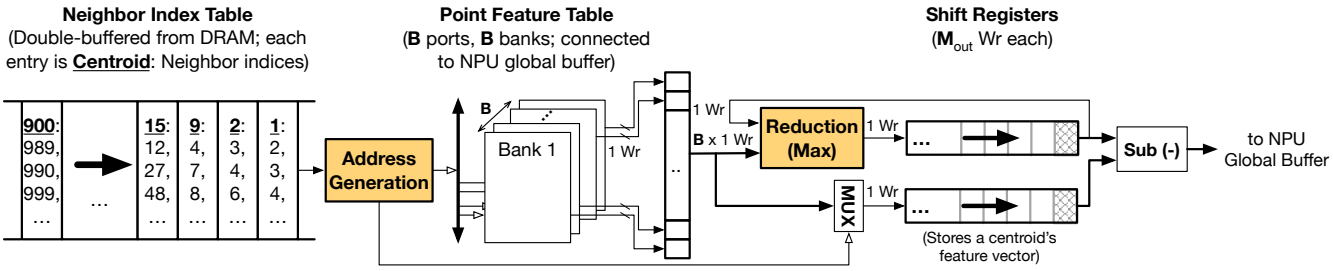


Fig. 14: Aggregation unit. The NIT buffer is double-buffered from the DRAM. The Address Generation logic simply fetches addresses already buffered in the NIT and sends them to the PFT buffer controller. The PFT buffer is organized as B independently addressed single-ported SRAMs. It could be thought of as an optimized version of a traditional B-banked, B-ported SRAM, because it does not need the crossbar that routes data from banks to ports (but does need the crossbar to route an address to the corresponding bank). The PFT buffer is connected to the NPU's global buffer. Each bank produces one word (Wr) per cycle. The shift registers hold up to M_{out} words, where M_{out} is the output feature vector size. The top shift register holds the result of reduction, and the bottom shift register holds the feature vector of a centroid.

order to limit the on-chip memory size. The PFT is stored in a separate on-chip SRAM connected to the NPU's global buffer (which stores the MLP weights and input/output). This allows the output of feature extraction to be directly transferred to the PFT buffer without going through the DRAM. Similarly, the aggregation output is directly written back to the NPU's global buffer, as the aggregation output of the current module is the input to the feature extraction in the next module.

To process each NIT entry, the Address Generation Unit (AGU) uses the K indices to generate K addresses to index into the PFT buffer. Due to the large read bandwidth requirement, the PFT buffer is divided into B independently addressable banks, each of which produces 1 word per cycle.

Each cycle, the PFT buffer produces B words, which enters the reduction unit. In our current design, the reduction unit implements the max logic as is the case in today's point cloud algorithms. The output of the max unit, i.e., the max of the B words, enters a shift register (the top one in Fig. 14). Ideally, the number of banks B is the same as the number of neighbors K and the K addresses fall into different banks. If so, the shift register is populated with the $1 \times M_{out}$ vector after M_{out} cycles. The AGU then reads **p**'s feature vector from the PFT buffer and stores it in another shift register (the bottom one in Fig. 14). The two shift registers perform an element-wise subtraction as required by aggregation. The same process continues until the entire NIT is exhausted.

Multi-Round Grouping In reality, reading the neighbor feature vectors takes more than M_{out} cycles because of two reasons. First, K could be greater than B. The number of banks B is limited by the peripheral circuits overhead, which increases as B increases. Second, some of the K addresses could fall into the same bank, causing bank conflicts. We empirically find that an LSB-interleaving reduces bank conflicts, but it is impossible to completely avoid bank conflict at runtime, because the data access patterns in point cloud are irregular and could not be statically calculated – unlike conventional DNNs and other regular kernels.

We use a simple multi-round design to handle both non-ideal scenarios. Each round the AGU would attempt to identify as many unconflicted addresses as possible, which is achieved by the AGU logic examining each address modulo B. The unconflicted addresses are issued to the PFT buffer, whose output enters the max unit to generate a temporary output stored in the shift register. The data in the shift register would be combined with the PFT output in the next round for reduction. This process continues until all the addresses in an NIT buffer entry are processed.

An alternative way to resolve bank-conflict would be to simply ignore conflicted banks, essentially approximating the aggregation operation. We leave it to future work to explore this optimization and its impact on the overall accuracy.

PFT Buffer Design One could think of the PFT buffer as a B-banked, B-ported SRAM. Traditionally, heavily ported and banked SRAMs are area inefficient due to the crossbar that routes each bank's output to the corresponding issuing port [54]. However, our PFT buffer is much simplified *without* the crossbar. This is by leveraging a key observation that the outputs of all the PFT banks are consumed by the max unit, which executes a *commutative* operation, i.e., $max(a,b) = max(b,a)$. Thus, the output of each bank need not be routed to the issuing port so long as the requested data is correctly produced. This design optimization greatly reduces the area overhead (Sec. VII-A).

One might be tempted to reuse the NPU's global buffer for the PFT buffer to save chip area. After all, the PFT *is* MLP's output, which is stored in the global buffer. However, physically sharing the two SRAM structures is difficult, mainly because of their different design requirements. Global buffer contains MLP weights and layer inputs, accesses to which have regular patterns. As a result, NPU global buffers are usually designed with very few ports (e.g., one) [3], [29] while using a wide word. In contrast, accesses to the PFT are irregular as the neighbors of a centroid could be arbitrary spread in the PFT. Thus, the PFT buffer must be heavily-ported in order to

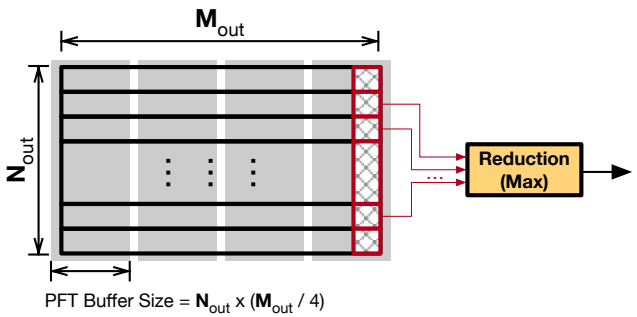


Fig. 15: Column-major partitioning of PFT to reduce PFT buffer size (4 partitions in this example). Each time the PFT buffer is filled with only one partition. Since reduction (max) is applied to each column independently, the column-major partitioning ensures that all the neighbors of a centroid are present in the PFT buffer for aggregation.

sustain a high bandwidth requirement.

PFT Partitioning To hold the entire PFT, the buffer must hold $N_{out} \times M_{out}$ features, which could be as large as 0.75 MB in some networks (e.g., DGCNN). Since the PFT buffer adds area overhead, we would like to minimize its size.

We partition the PFT to reduce the PFT buffer size. Each time, the PFT buffer is filled with only one partition. One straightforward strategy is the row-major partitioning, where the PFT buffer holds only a few rows of the PFT. However, since a centroid's neighbors could be arbitrarily spread across different PFT rows, row-major partitioning does not guarantee that all the neighbors of a centroid are present in the PFT buffer (i.e., in the same partition) for aggregation.

Instead, our design partitions the PFT column-wise, where each partition contains several columns of the PFT. Fig. 15 illustrates the idea with 4 partitions. In this way, aggregation of a centroid is divided into four steps, each step aggregating only one partition. The column-major partitioning ensures that, within each partition, the neighbors of a centroid are available in the PFT buffer. Since reductions (max) of different columns are independent, the four intermediate reduction results can simply be concatenated in the end.

With column-wise partitioning, each NIT entry is accessed multiples times—once per aggregation step. Thus, a smaller PFT buffer, while reducing the area overhead, would also increase the energy overhead. We later quantify this resource vs. energy trade-off (Sec. VII-F).

VI. Experimental Setup

Hardware Implementation We develop RTL implementations for the NPU and its augmentations for the aggregation unit (AU). The NPU is based on the systolic array architecture, and consists of a 16×16 PE array. Each PE consists of two input registers, a MAC unit with an accumulator register, and simple trivial control logic. This is identical to the PE in the TPU [29]. Recall that MLPs in point cloud networks process batched inputs (Fig. 3), so the MLPs perform matrix-matrix product that can be efficiently implemented on a systolic array.

TABLE I: Evaluation benchmarks.

Application Domains	Algorithm	Dataset	Year
Classification	PointNet++ (c)	ModelNet40	2017
	DGCNN (c)		2019
	LDGCNN		2019
	DensePoint		2019
Segmentation	PointNet++ (s)	ShapeNet	2017
	DGCNN (s)		2019
Detection	F-PointNet	KITTI	2018

The NPU's global buffer is sized at 1.5 MB and is banked at a 128 KB granularity.

The PFT buffer in the AU is sized at 64 KB with 32 banks. The NIT buffer is doubled-buffered; each buffer is implemented as one SRAM bank sized at 12 KB and holds 128 entries. The NIT buffer produces one entry per cycle. Each entry is 98 Bytes, accommodating 64 neighbor indices (12 bits each). Each of the two shift registers is implemented as 256 flip-flops (4-byte each). The datapath mainly consists of 1) one 33-input max unit and 256 subtraction units in the reduction unit, and 2) 32 32-input MUXes in the AGU.

The design is clocked at 1 GHz. The RTL is implemented using Synposys synthesis and Cadence layout tools in TSMC 16nm FinFET technology, with SRAMs generated by an Arm memory compiler. Power is simulated using Synopsys PrimeTimePX, with fully annotated switching activity.

Experimental Methodology The latency and energy of the NPU (and its augmentation) are obtained from post-synthesis results of the RTL design. We model the GPU after the Pascal mobile GPU in the Nvidia Parker SoC hosted on the Jetson TX2 development board [10]. The SoC is fabricated in a 16 nm technology node, same as our NPU. We directly measure the GPU execution time as well as the kernel launch time. The GPU energy is directly measured using the built-in power sensing circuity on TX2.

The DRAM parameters are modeled after Micron 16 Gb LPDDR3-1600 (4 channels) according to its datasheet [7]. DRAM energy is calculated using Micron's System Power Calculators [8] using the memory traffic, which includes: 1) GPU reading input point cloud, 2) NPU accessing MLP kernels and activations each layer, and 3) GPU writing NIT and NPU reading NIT. Overall, the DRAM energy per bit is about 70× of that of SRAM, matching prior work [23], [61].

The system energy is the aggregation of GPU, NPU, and DRAM. The overall latency is sum of GPU, NPU, and DRAM minus: 1) double buffering in the NPU, and 2) parallel execution between neighbor search on GPU and feature computation on NPU. Due to double-buffering, the overall latency is dominated by the compute latency, not memory.

Software Setup Tbl. I lists the point cloud networks we use, which cover different domains for point cloud analytics including object classification, segmentation, and detection. The networks cover both classic and recent ones (2019).

For classification, we use four networks: PointNet++ [43], DGCNN [53], LDGCNN [65], and DensePoint [34]; we use

the ModelNet40 [58] dataset. We report the standard overall accuracy metric. To evaluate segmentation, we use the variants of PointNet++ and DGCNN specifically built for segmentation, and use the ShapeNet dataset [19]. We report the standard mean Intersection-over-Unit (mIoU) accuracy metric. Finally, we use F-PointNet [41] as the object detection network. We use the KITTI dataset [24] and report the geometric mean of the IoU metric (BEV) across its classes.

We optimize the author-released open-source version of these networks to obtain stronger software baselines. We: 1) removed redundant data duplications introduced by `tf.tile`; 2) accelerated the CPU implementation of an important kernel, 3D Interpretation (`three_interpolate`), with a GPU implementation; 3) replaced the Farthest Point Sampling with random sampling in PointNet++ with little accuracy loss; 4) replaced the Grouping operation (`group_point`) with an optimized implementation (`tf.gather`) to improve the efficiency of grouping/aggregation. On TX2, our baseline networks are 2.2× faster than the open-source versions.

Baseline We mainly compare with a generic NPU+GPU SoC without any Mesorasi-related optimizations. Compared to the baseline, our proposal improves both the software, i.e., the delayed-aggregation algorithm as well as hardware, i.e., the aggregation unit (AU) augmentations to the NPU.

Variants To decouple the contributions of our algorithm and hardware, we present two different Mesorasi variants:

- Mesorasi-SW: delayed-aggregation without AU support. Neighbor search and aggregation execute on the GPU; feature computation executes on the NPU.
- Mesorasi-HW: delayed-aggregation with AU support. Neighbor search executes on the GPU; aggregation and feature computation execute on the NPU.

VII. Evaluation

We first show Mesorasi adds little hardware overhead (Sec. VII-A) while achieving comparable accuracy against original point cloud networks (Sec. VII-B). We then demonstrate the efficiency gains of Mesorasi on different hardware platforms (Sec. VII-C – Sec. VII-E), followed by sensitivity studies (Sec. VII-F).

VII-A. Area Overhead

Mesorasi introduces only minimal area overhead with the minor AU augmentations. The main overhead comes from the 88 KB additional SRAM required for the PFT buffer and the NIT buffer. Compared to the baseline NPU, the additional hardware introduces less than 3.8% area overhead (0.059 mm^2), which is even more negligible compared to the entire SoC area (e.g., 350 mm^2 for Nvidia Xavier [12] and 99 mm^2 for Apple A13 [1]).

Our custom-designed PFT buffer avoids the crossbar connecting the banks to the read ports by exploiting the algorithmic characteristics (Sec. V-B). Since our PFT buffer is heavily banked (32) and ported (32) and each bank is small in size (2 KB), the additional area overhead introduced by the crossbar would have been high. Specifically, the area of the PFT buffer now is 0.031 mm^2, but the crossbar area would be 0.064 mm^2, which is now avoided.

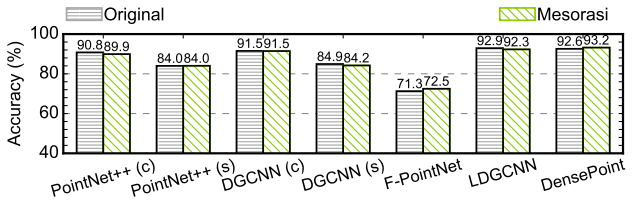


Fig. 16: The accuracy comparison between networks trained with delayed-aggregation and the original networks.

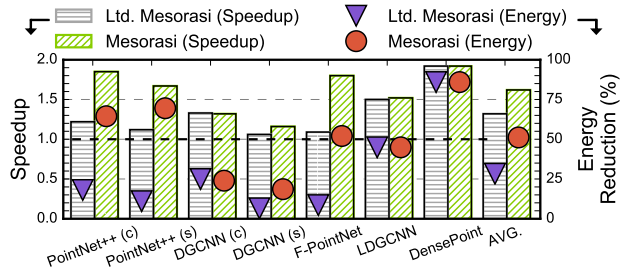


Fig. 17: Speedup and energy reduction of the delayed-aggregation algorithm and the limited version of the algorithm on the mobile Pascal GPU on TX2.

VII-B. Accuracy

Overall, Mesorasi matches or out-performs the original algorithms. We train all seven networks with delayed-aggregation from scratch until the accuracy converges. Fig. 16 compares our accuracy with that of the baseline models, which we choose the better of the reported accuracies in the original papers or accuracies from training their released code. Overall, Mesorasi leads to at most 0.9% accuracy loss in the case of PointNet++ (c) and up to 1.2% accuracy gain in the case of F-PointNet. This shows that, while delayed-aggregation approximates the original algorithms, the accuracy loss could be recovered from training. Delayed-aggregation could be used as a primitive to build accurate point cloud algorithms.

We find that fine-tuning the model weights trained on the original networks has similar accuracies as retraining from scratch. However, directly using the original weights without retraining leads to a few percentages of accuracy loss, which is more significant when the non-linear layers use batch normalization, which perturbs the distributive property of matrix multiplication over subtraction more than ReLU.

VII-C. Results on GPU

We first show that our delayed-aggregation algorithm readily achieves significant speedups on today's GPU *without* hardware support. Fig. 17 shows the speedup and energy reduction of Mesorasi on the Pascal GPU on TX2.

As a comparison, we also show the results of a limited version of delayed-aggregation, where only the matrix-vector multiplication (MVM) part of an MLP is hoisted before aggregation (Ltd-Mesorasi). The limited delayed-aggregation

algorithm is inspired by certain Graph Neural Network (GNN) implementations such as GCN [4], [22], [52] and GraphSage [5]. Note that by hoisting only the MVM rather than the entire MLP, Ltd-Mesorasi is precise since MVM is linear. We refer interested readers to the wiki page of our code repository [6] for a detailed comparison between our delayed-aggregation and GNN's limited delayed-aggregation.

On average, Mesorasi achieves 1.6× speedup and 51.1% energy reduction compared to the original algorithms. In comparison, the limited delayed-aggregation algorithm achieves only 1.3× speedup and 28.3% energy reduction. Directly comparing with Ltd-Mesorasi, Mesorasi has 1.3× speedup and 25.9% energy reduction. This is because the limited delay-aggregation, in order to be precise, could be applied to only the first MLP layer. By being approximate, Mesorasi does not have this constraint and thus enables larger benefits; the accuracy loss could be recovered through fine-tuning (Fig. 16). Mesorasi has similar performance as Ltd-Mesorasi on DGCNN (c), LDGCNN, and DensePoint, because these three networks have only one MLP layer per module.

Although delayed-aggregation allows neighbor search and feature extraction to be executed in parallel, and our implementation does exploit the concurrent kernel execution in CUDA, we find that neighbor search and feature extraction in actual executions are rarely overlapped. Further investigation shows that this is because the available resources on the Pascal GPU on TX2 are not sufficient to allow both kernels to execute concurrently. We expect the speedup to be even higher on more powerful mobile GPUs in the future.

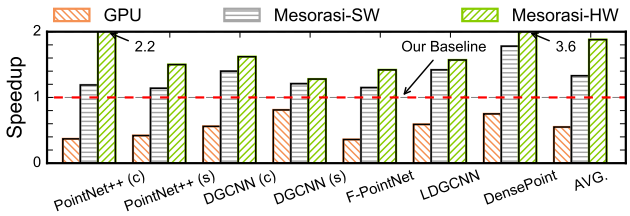
Overall, networks in which feature computation contributes more heavily to the overall time, such as PointNet++ (c) and F-PointNet (Fig. 5), have higher MAC operation reductions (Fig. 9), and thus have higher speedups and energy reductions. This confirms that the improvements are mainly attributed to optimizing the MLPs in feature computation.

VII-D. Speedup and Energy Reduction

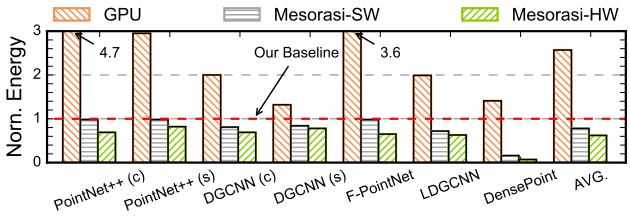
Mesorasi also improves the performance and energy consumption of emerging mobile SoCs with a dedicated NPU. Fig. 18a and Fig. 18b show the speedup and the normalized energy consumption of the two Mesorasi variants over the NPU+GPU baseline, respectively.

Software The delayed-aggregation algorithm alone without AU support, i.e., Mesorasi-SW, has a 1.3× speedup and 22% energy saving over the baseline. The main contributor of the improvements is optimizing the MLPs in feature computation. Fig. 19a shows the speedups and energy savings of the delayed-aggregation algorithm on feature computation. On average, the feature computation time is reduced by 5.1× and the energy consumption is reduced by 76.3%.

The large speedup on feature computation does not translate to similar overall speedup, because feature computation time has already been significantly reduced by the NPU, leaving less room for improvement. In fact, our GPU+NPU baseline is about 1.8× faster (Fig. 18a) and consumes 70% less energy compared to the GPU (Fig. 18b). The increased workload of

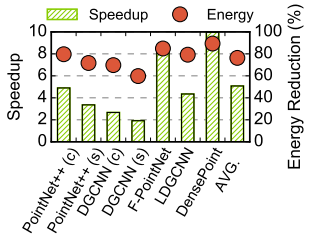


(a) Speedup. Higher is better.

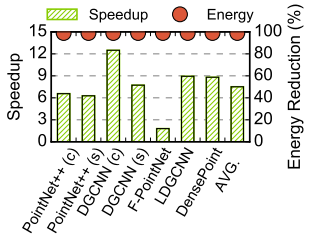


(b) Normalized energy. Lower is better.

Fig. 18: Speedup and energy reduction of Mesorasi-SW and Mesorasi-HW over the baseline (GPU+NPU), which is twice as fast and consumes one-third of the energy compared to the GPU, indicating an optimized baseline to begin with.



(a) Feature computation.



(b) Aggregation.

Fig. 19: Speedup and energy savings on feature computation and aggregation.

aggregation also adds to the overhead, leading to overall lower speedup and energy reduction than on GPU.

Hardware With the AU hardware, Mesorasi-HW boosts the speedup to 1.9× (up to 3.6×) and reduces the energy consumption by 37.6% (up to 92.9%). DGCNN (s) has the least speedup because it has the least aggregation time (Fig. 12), thus benefiting the least from the AU hardware.

Fig. 19 shows the speedup and energy reduction of aggregation over the baseline (which executes aggregation on the GPU). Overall, Mesorasi-HW reduces the aggregation time by 7.5× and reduces the energy by 99.4%. The huge improvements mainly come from using a small memory structure customized to the data access patterns in aggregation.

On average, 27% (max 29%) of PFT buffer accesses are to serve previous bank conflicts. The total time spent on PFT buffer accesses is 1.5× of the ideal case without bank conflicts. Empirically we do not observe pathological cases.

The AU's speedup varies across networks. For instance, the speedup on PointNet++ (c) is over 3× higher than that of F-PointNet. This is because the speedup decreases as bank conflict increases; bank conflicts occur more often

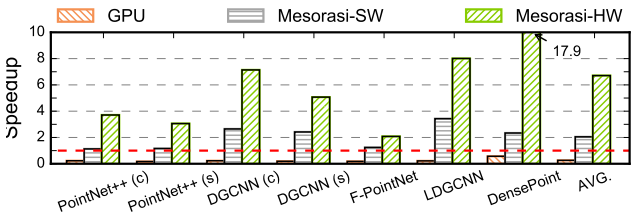


Fig. 20: Mesorasi-SW and Mesorasi-HW speedup over an NSE-enabled SoC (GPU+NPU+NSE), which is 4.0× faster than the GPU by accelerating both MLP and neighbor search.

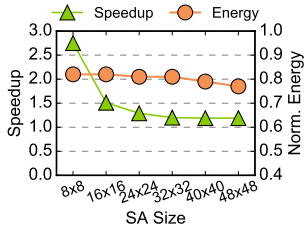


Fig. 21: Sensitivity of the speedup and energy to the systolic array size.

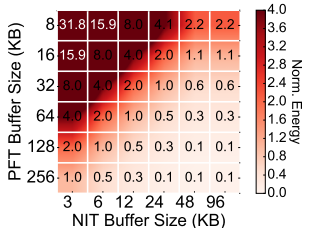


Fig. 22: Sensitivity of AU energy consumption to the NIT/PFT buffer sizes.

when neighbor search returns more neighbors. The neighbor searches in PointNet++ (c) mostly return 32 neighbors, whereas neighbor searches in F-PointNet return mostly 128 neighbors, significantly increasing the chances of bank conflicts. This also explains why PointNet++ (c) has overall higher speedup than F-PointNet (Fig. 18a).

VII-E. Results with Neighbor Search Engine (NSE)

From the evaluations above, it is clear that the improvements of Mesorasi will ultimately be limited by the neighbor search overhead, which Mesorasi does not optimize and becomes the "Amdahl's law bottleneck."

To assess the full potential of Mesorasi, we evaluate it in a futuristic SoC that incorporates a dedicated neighbor search engine (NSE) that accelerates neighbor searches. We implement a recently published NSE built specifically for accelerating neighbor searches in point cloud algorithms [59], and incorporate it into our SoC model. On average, the NSE provides over 60× speedup over the GPU. Note that the NSE is *not* our contribution. Instead, we evaluate the potential speedup of Mesorasi if an NSE is available.

The speedup of Mesorasi greatly improves when neighbor search is no longer a bottleneck. Fig. 20 shows the speedups of Mesorasi-SW and Mesorasi-HW on the NSE-enabled SoC. On average, Mesorasi-SW achieves 2.1× speedup and Mesorasi-HW achieves 6.7× speedup. The two DGCNN networks have particularly high speedups because neighbor search contributes heavily to their execution times (Fig. 5).

VII-F. Sensitivity Study

The evaluations so far are based on one hardware configuration. We now study how the improvements vary with different hardware resource provisions. In particular, we focus on two types of hardware resources: the baseline NPU and the AU augmentation. Due to the page limit, we show only the results of PointNet++ (s). The general trend holds.

NPU We find that Mesorasi has higher speedups when the NPU is smaller. Fig. 21 shows how the speedup and normalized energy of Mesorasi-HW over the baseline vary as the systolic array (SA) size increases from 8×8 to 48×48. As the SA size increases, the feature extraction time decreases, and thus leaving less room for performance improvement. As a result, the speedup decreases from 2.8× to 1.2×.

Meanwhile, the energy reduction increases from 17.7% to 23.4%. This is because a large SA is more likely throttled by memory bandwidth, leading to overall higher energy.

AU We find that the AU energy consumption is sensitive to the NIT and PFT buffer sizes. Fig. 22 shows the AU energy under different NIT and PFT buffer sizes. The results are normalized to the nominal design point described in Sec. VI (i.e., a 64 KB of PFT and a 12 KB NIT).

The energy consumption increases as the PFT and NIT buffer sizes decrease. In an extremely small setting with an 8 KB PFT buffer and a 3 KB NIT buffer, the AU energy increases by 32×, which leads to a 5.6% overall energy increase. This is because a smaller PFT buffer leads to more PFT partitions, which increases NIT buffer energy since each NIT entry must be read once per partition. Meanwhile, a smaller NIT requires more DRAM accesses, whose energy dominates as the PFT buffer becomes very small. On the other extreme, using a 256 KB PFT buffer and a 96 KB NIT buffer reduces the overall energy by 2.0% while increasing the area overhead by 4×. Our design point balances energy saving and area overhead.

VIII. Related Work

Point Cloud Analytics Point cloud has only recently received extensive interests. Unlike conventional image/video analytics, point cloud analytics requires considerably different algorithms due to the unique characteristics of point cloud data. Most of the recent work focuses on the accuracy, exploring not only different data representation (e.g., 2D projection [42], voxelization [45], [58], and raw points [43], [53]), but also different ways to extract features from points [43], [49], [53], [57]. Our delayed-aggregation primitive can be thought of as a new, and efficient, way of extracting features from raw points.

Mesorasi focuses on improving the efficiency of point cloud algorithms while retaining the high accuracy. In the same vein, PVCNN [35] combines point-based and voxel-based data representations in order to boost compute and memory efficiency. Different but complementary, Mesorasi focuses on point-based neural networks. While PVCNN is demonstrated on GPUs, Mesorasi not only directly benefits commodity GPUs, but also incorporates systematic hardware support that improves DNN accelerators.

Prior work has also extensively studied systems and architectures for accelerating neighbor search on GPU [25], [44], FPGA [27], [31], [56], and ASIC [59]. Neighbor search

contributes non-trivial execution time to point cloud networks. MESORASI hides, rather than reduces, the neighbor search latency, and directly benefits from faster neighbor search.

GNNs Point cloud applications bear some resemblance to GNNs. After all, both deal with spatial/geometric data. In fact, some point cloud applications are implemented using GNNs, e.g., DGCNN [53].

However, existing GNN accelerators, e.g., HyGCN [60], are insufficient in accelerating point cloud applications. Fundamentally, GNN does not require explicit neighbor search (as a vertex's neighbors are explicitly encoded), but neighbor search is a critical bottleneck of all point cloud applications, as points are arbitrarily spread in 3D space. Our design hides the neighbor search latency, which existing GNN accelerators simply do not optimize for. In addition, MESORASI minimally extends conventional DNN accelerators instead of being a new accelerator design, broadening its applicability in practice.

From Fig. 5, one might notice that $\mathcal{A}$ in point cloud networks is much faster than $\mathcal{F}$, which is the opposite in many GNNs [60]. This is because $\mathcal{F}$ in point cloud applications does much more work than $\mathcal{A}$, opposite to GNNs. In point cloud application, $\mathcal{A}$ simply gathers neighbor feature vectors, and $\mathcal{F}$ operates on neighbor feature vectors (MLP on each vector). In contrast, $\mathcal{A}$ in GNNs gathers and reduces neighbor feature vectors to one vector, and $\mathcal{F}$ operates on the reduced vector (MLP on one vector).

Domain-Specific Accelerator Complementary to improving generic DNN accelerators, much of recent work has focused on improving the DNN accelerators for specific application domains such as real-time computer vision [18], [21], [67], computational imaging [28], [36], and language processing [46]. The NPU in the MESORASI architecture is a DNN accelerator specialized to point cloud processing. MESORASI also extends beyond prior visual accelerators that deal with 2D data (images and videos) [32], [37], [38], [63], [64], [66] to 3D point clouds.

To keep the modularity of existing SoCs, MESORASI relies on the DRAM for inter-accelerator communication. That said, MESORASI could benefit from more direct accelerator communication schemes such as VIP [39] and Short-circuiting [62]. For instance, the NIT could be directly communicated to the NIT buffer from the GPU through a dedicated on-chip link, pipelining neighbor search with aggregation.

IX. CONCLUSION

With the explosion of 3D sensing devices (e.g., LiDAR, stereo cameras), point cloud algorithms present exciting opportunities to transform the perception ability of future intelligent machines. MESORASI takes a systematic step toward efficient point cloud processing. The key to MESORASI is the delayed-aggregation primitive that decouples neighbor search with feature computation and significantly reduces the overall workload. Hardware support maximizes the effectiveness of delayed-aggregation. The potential gain is even greater in future SoCs where neighbor search is accelerated.

REFERENCES

[1] "Apple says its new A13 Bionic chip brings hours of extra battery life to new iPhones." [Online]. Available: https://en.wikichip.org/wiki/apple/ax/a13

[2] "ARCore." [Online]. Available: https://developers.google.com/ar

[3] "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

[4] "GCN in PyTorch Geometric." [Online]. Available: https://github.com/rusty1s/pytorch_geometric/blob/master/examples/gcn.py

[5] "GraphSage in TensorFlow." [Online]. Available: https://github.com/williamleif/GraphSAGE

[6] "Mesorasi wiki." [Online]. Available: https://github.com/horizon-research/Efficient-Deep-Learning-for-Point-Clouds/wiki

[7] "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

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

[9] "Nvidia GP10B Spec." [Online]. Available: https://www.techpowerup.com/gpu-specs/nvidia-gp10b.g856

[10] "NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge." [Online]. Available: https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/

[11] "Nvidia reveals xavier soc details." [Online]. Available: https://www.forbes.com/sites/moorinsights/2018/08/24/nvidia-reveals-xavier-soc-details/amp/

[12] "NVIDIA's Xavier System-on-Chip, HotChips 30." [Online]. Available: https://www.hotchips.org/hc30/1conf/1.12_Nvidia_XavierHotchips2018Final_814.pdf

[13] "Pascal (microarchitecture)." [Online]. Available: https://en.wikipedia.org/wiki/Pascal_(microarchitecture)

[14] "Second Version of HoloLens HPU will Incorporate AI Coprocessor for Implementing DNNs." [Online]. Available: https://www.microsoft.com/en-us/research/blog/second-version-hololens-hpu-will-incorporate-ai-coprocessor-implementing-dnns/

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

[16] P. Alliez, F. Forge, L. De Luca, M. Pierrot-Deseilligny, and M. Preda, "Culture 3d cloud: A cloud computing platform for 3d scanning, documentation, preservation and dissemination of cultural heritage," 2017.

[17] J. Behley, M. Garbade, A. Milioto, J. Quenzel, S. Behnke, C. Stachniss, and J. Gall, "Semantickitti: A dataset for semantic scene understanding of lidar sequences," in *Proceedings of the 13th IEEE International Conference on Computer Vision*, 2019.

[18] M. Buckler, P. Bedoukian, S. Jayasuriya, and A. Sampson, "Eva^2: Exploiting temporal redundancy in live computer vision," in *Proceedings of the 45th ACM/IEEE Annual International Symposium on Computer Architecture*, 2018.

[19] 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.

[20] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 367–379.

[21] Y. Feng, P. Whatmough, and Y. Zhu, "Asv: Accelerated stereo vision system," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[22] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[23] 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.

[24] 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.

[25] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel, "Buffer k-d trees: Processing massive nearest neighbor queries on gpus," 2014.

[26] M. Gross and H. Pfister, *Point-based graphics*. Elsevier, 2011.

[27] 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.

[28] C.-T. Huang, Y.-C. Ding, H.-C. Wang, C.-W. Weng, K.-P. Lin, L.-W. Wang, and L.-D. Chen, "ecnn: A block-based and highly-parallel cnn accelerator for edge inference," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[29] 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 *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

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

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

[32] Y. Leng, C.-C. Chen, Q. Sun, J. Huang, and Y. Zhu, "Energy-efficient video processing for virtual reality," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[33] M. Levoy and T. Whitted, *The use of points as a display primitive*. Citeseer, 1985.

[34] 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.

[35] Z. Liu, H. Tang, Y. Lin, and S. Han, "Point-voxel cnn for efficient 3d deep learning," in *Advances in Neural Information Processing Systems*, 2019, pp. 963–973.

[36] M. Mahmoud, K. Siu, and A. Moshovos, "Diffy: A déjà vu-free differential deep neural network accelerator," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[37] M. Mahmoud, B. Zheng, A. D. Lascorz, F. H. Assouline, J. Assouline, P. Boucher, E. Onzon, and A. Moshovos, "Ideal: Image denoising accelerator," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[38] A. Mazumdar, T. Moreau, S. Kim, M. Cowan, A. Alaghi, L. Ceze, M. Oskin, and V. Sathe, "Exploring computation-communication tradeoffs in camera systems," in *Proceedings of the 9th IEEE International Symposium on Workload Characterization*, 2017.

[39] N. C. Nachiappan, H. Zhang, J. Ryoo, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "Vip: virtualizing ip chains on handheld platforms," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 655–667, 2016.

[40] H. Pfister, M. Zwicker, J. Van Baar, and M. Gross, "Surfels: Surface elements as rendering primitives," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, 2000, pp. 335–342.

[41] 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.

[42] C. R. Qi, H. Su, M. Nießner, A. Dai, M. Yan, and L. J. Guibas, "Volumetric and multi-view cnns for object classification on 3d data," in *Proceedings of the 29th IEEE conference on computer vision and pattern recognition*, 2016.

[43] 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.

[44] 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.

[45] G. Riegler, A. Osman Ulusoy, and A. Geiger, "Octnet: Learning deep 3d representations at high resolutions," in *Proceedings of the 30th IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[47] S. Rusinkiewicz and M. Levoy, "Qsplat: A multiresolution point rendering system for large meshes," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, 2000, pp. 343–352.

[46] M. Riera, J.-M. Arnau, and A. González, "Computation reuse in dnns by exploiting input similarity," in *Proceedings of the 45th IEEE Annual International Symposium on Computer Architecture*, 2018.

[48] R. B. Rusu, N. Blodow, and M. Beetz, "Fast point feature histograms (fpfh) for 3d registration," in *Proceedings of the 22nd IEEE International Conference on Robotics and Automation*, 2009.

[49] M. Simonovsky and N. Komodakis, "Dynamic edge-conditioned filters in convolutional neural networks on graphs," in *Proceedings of the 30th IEEE conference on computer vision and pattern recognition*, 2017.

[50] 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.

[51] F. Tombari, S. Salti, and L. Di Stefano, "Unique signatures of histograms for local surface description," in *European conference on computer vision*. Springer, 2010.

[52] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma *et al.*, "Deep graph library: Towards efficient and scalable deep learning on graphs," *arXiv preprint arXiv:1909.01315*, 2019.

[53] 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.

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

[55] 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.

[56] 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.

[57] W. Wu, Z. Qi, and L. Fuxin, "Pointconv: Deep convolutional networks on 3d point clouds," in *Proceedings of the 32nd IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[58] 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.

[59] 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.

[60] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Hygcn: A gcn accelerator with hybrid architecture," in *Proceedings of the 26th International Symposium on High Performance Computer Architecture*, 2020.

[61] 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.

[62] P. Yedlapalli, N. C. Nachiappan, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "Short-circuiting memory traffic in handheld platforms," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 166–177.

[63] H. Zhang, P. V. Rengasamy, S. Zhao, N. C. Nachiappan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "Race-to-sleep+ content caching+ display caching: a recipe for energy-efficient video streaming on handhelds," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[64] H. Zhang, S. Zhao, A. Pattnaik, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "Distilling the essence of raw video to reduce memory usage and energy at edge devices," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[65] 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.

[66] S. Zhao, H. Zhang, S. Bhuyan, C. Mishra, Z. Ying, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "Déjà-view: Spatio-temporal compute reuse for energy-efficient 3600 vr video streaming," in *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture*, 2020.

[67] Y. Zhu, A. Samajdar, M. Mattina, and P. Whatmough, "Euphrates: Algorithm-soc co-design for low-power mobile continuous vision," in *Proceedings of the 45th ACM/IEEE Annual International Symposium on Computer Architecture*, 2018.