FusionArch: A Fusion-Based Accelerator for Point-Based Point Cloud Neural Networks

Xueyuan Liu[1], Zhuoran Song[1*] Guohao Dai[1], Gang Li[1], Can Xiao[2],
Yan Xiang[2], Dehui Kong[2], Ke Xu[2] and Xiaoyao Liang[1]
[1]Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
[2]Sanechips Technology Co., Ltd
[1]{xueyuan_liu, songzhuoran}@sjtu.edu.cn

***Abstract*—Point-based Point Cloud Neural Networks (PCNNs) have attracted much attention for their higher accuracy than voxel-based and multi-view-based PCNNs. Nevertheless, the increasing scale of point cloud data poses a challenge for real-time processing. Numerous previous works focus on accelerating PCNN inference but only optimize specific stages, limiting their generality to different networks with diverse performance bottlenecks.**

In this paper, we take nearly all stages of PCNNs into account, and propose 3 orthogonal algorithms, including Fusion-FPS, Fusion-Computation, and Fusion-Aggregation. We introduce Fusion-FPS to alter the sequential execution flow by reducing the Farthest Point Sampling (FPS) across layers to once and organize all neighbor search stages in parallel. To exclude redundant feature computations of "Filling Points", we propose Fusion-Computation, identifying the presence and locations of "Filling Points" and directly borrowing the nearest neighbor features for them. To eliminate redundant memory accesses caused by shared neighbors in aggregation, we present Fusion-Aggregation, which clusters nearby centroids and coalesces their replicated accesses. In support of our algorithms, we co-design FusionArch, an architecture that implements our strategies and further optimizes memory access via a Local Fusion-Aggregation Table (LFT). We evaluate FusionArch on both server-level and edge-level platforms on 5 PCNNs across 4 applications and show remarkable accuracy and performance gains. On average, FusionArch achieves $2.6\times$, $5.6\times$, $13.0\times$ speedup and $17\times$, $22\times$, $62.4\times$ energy savings over PointAcc.Server, NVIDIA A100 GPU and Intel Xeon CPU, respectively. Moreover, it outperforms PRADA, PointAcc.Edge, Mesorasi and GPU with speedups of $2.4\times$, $2.9\times$, $5.3\times$, $5.5\times$, and energy savings of $4.4\times$, $7.2\times$, $12.4\times$, $11.5\times$, respectively.

I. Introduction

Nowadays, point-based PCNNs play a pivotal role in various cutting-edge technologies, including autonomous driving, Augmented Reality (AR), remote sensing, and more. These networks directly process 3D points, preserving intricate details within point cloud data, thereby achieving impressive accuracy. Point-based PCNNs, exemplified by the groundbreaking PointNet++ [1] and its subsequent derivative networks [2]–[4], are composed of key components including FPS [5], neighbor search, aggregation, and feature computation, all integrated seamlessly along the critical path.

Each stage on the critical path has limitations that impede performance. FPS is hindered by the sequential execution order, extensive computations and irregular memory accesses, resulting in expensive performance costs. The aggregation is memory-bounded, which gathers quite lengthy features in groups of centroids, inducing non-negligible memory access latency. The feature computation includes a series of compute-intensive kernels, and its heavy workload makes PCNNs far from real-time processing. Additionally, we notice that performance bottlenecks vary across different PCNNs. For example, FPS dominates PointNet++, while feature computation has a pronounced impact on F-PointNet++ [4] and PointWeb [2].

Given such factors hampering PCNN's performance, numerous previous works, such as Mersoasi [6], PointAcc [7], PRADA [8], EdgePC [9] and so on, are committed to promoting PCNN's inference. However, most of these approaches tend to target specific stages, which limits their applicability across various PCNNs with diverse bottlenecks. For instance, EdgePC only optimizes FPS, which may achieve remarkable performance on PointNet++, but fails in other PCNNs whose performance is not mainly hindered by FPS. PointAcc mainly focuses on feature computation, resulting in fewer benefits on PointNet++ than other computing-bound PCNNs like PointWeb.

In view of this, we take a deeper look at multiple stages of PCNNs and expose the following challenges: i) **FPS is serial executed with low parallelism.** Fig. 1 depicts that each layer's FPS is performed on the results of the previous layer, which means the current layer's FPS cannot be launched before the preceding layer is finished, resulting in long periods of idleness and low parallelism. We conducted an experiment on NVIDIA A100 GPU, observing that FPS accounts for 96% execution time of PointNet++. ii) **Redundant computations in feature computation.** Owing to the nonuniform distribution of points, the searched neighbors are often insufficient. Such a dilemma is solved by employing the nearest neighbor away from the centroid to fill the gap, and we termed it "Filling Points" [1]. The feature computation for these identical "Filling Points" is completely redundant. Fig. 1 reveals that over 64.8% of neighborhoods on PointNet++ contain "Filling Points", and nearly 86.3% of those experience recomputation more than 5 times, indicating the optimization for "Filling Points" feature computations will lead to a considerable performance gains. iii) **Redundant memory accesses in aggregation.** As a consequence of the clustered distribution of points, the adjacent centroids are more likely to share common neighbors, causing huge redundant memory accesses in aggregation. As illustrated in Fig. 1, more than 73% features are repeatedly accessed beyond 4 times on PointNet++.

* Zhuoran Song is the corresponding author.

979-8-3503-4859-0/DATE24/© 2024 EDAA

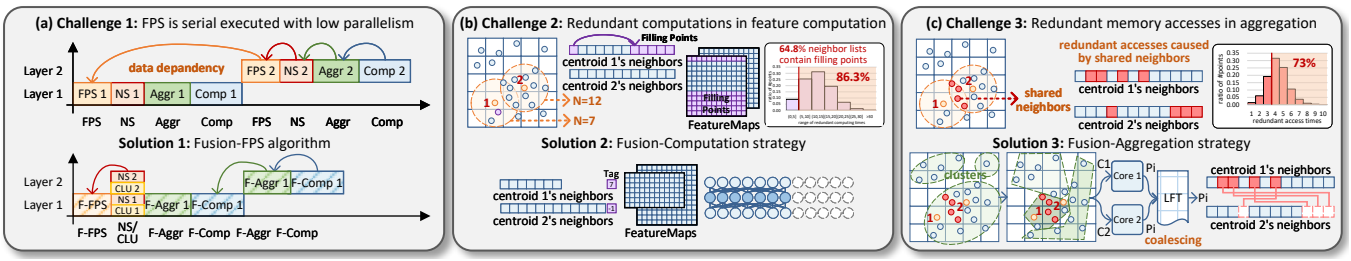

Fig. 1. Challenges across different stages in point-based PCNNs.

To address the aforementioned challenges, we make the following contributions:

- **Fusion-FPS:** To break the sequential order of FPS, we introduce Fusion-FPS. The algorithm reduces all layers' FPS to only once and take the centroids for succeeding layers directly from the 1st layer. It changes network execution flow to parallel stages on the critical path and further alleviates its computational workload.
- **Fusion-Computation:** To exclude the redundant feature computations of "Filling Points", we propose Fusion-Computation, which identifies the presence and locations of "Filling Points" and fuses their computations.
- **Fusion-Aggregation:** To eliminate the redundant accesses of shared neighbors in aggregation, we design Fusion-Aggregation. The strategy employs balanced K-means clustering to enhance the spatial locality of centroids, which provides opportunities to memory coalescing.
- **FusionArch:** In support of our algorithms, we co-design FusionArch, an architecture that hardwareizes algorithmic optimizations and incorporates a Local Fusion-Aggregation Table (LFT) to coalesce redundant memory accesses from parallel cores in a non-blocking manner.

We evaluated our FusionArch across an extensive set of benchmarks encompassing diverse scenarios and point scales, achieving notable accuracy and performance gains.

II. Background and Related Works

A. Point-based PCNN

The paradigm of point-based PCNNs comprises 4 stages: FPS, neighbor search, aggregation and feature computation.

*1) **FPS**:* FPS is the most representative sampling algorithm, following formula 1, where C_i denotes centroid i's index and P_j denotes point j's coordinates. FPS arbitrarily selects an initial centroid and pushes it into the centroid set. During the following iterations before reaching the desired number of centroids, FPS calculates distances between centroids and all points, recording the minimum distance for each point. Afterward, the point with the maximum distance among all points is selected as the next centroid.

$$C_i = \arg \max_{j \in [1,N]} (\min_{k \in [1,i]} (dist(P_j - P_{C_k}))) \quad (1)$$

*2) **Neighbor Search**:* Neighbor search determines neighbors using K-Nearest-Neighbor (KNN) [10] or ball query. Given that ball query is more generalizable than KNN [1], we concentrate on it in this work. Ball query calculates distances between the centroid and all points, trying to find K neighbors within a sphere of radius r. However, such strict constraints may lead to insufficient neighbors. To bridge the gap, the nearest neighbor is utilized as "Filling Point".

*3) **Aggregation**:* Aggregation constructs local features by subtracting the coordinates of the centroid from its neighbors, and then concatenating neighbor features, which consolidate to form input matrics for feature computation.

*4) **Feature Computation**:* The feature computation employs the multi-layer perceptron (MLP) to process input matrices. Its output features serve as inputs for subsequent layers.

B. Related Works

Several works have explored optimizations for point-based PCNNs. Mesorasi [6] proposes a "delayed aggregation" algorithm that changes the order of aggregation and feature computation, reducing the workload by performing feature computation on shorter features. PRADA [8] also focuses on feature computation, employing "dynamic approximation" to approximate and eliminate the feature computation for similar local pairs. EdgePC [9] moves its attention to the FPS stage, incorporating Morton Code to structurize the point cloud and utilizing uniform sampling, commonly used in 2D images instead. QuickFPS [11] introduces a bucket-based FPS algorithm that organizes the point cloud into multiple buckets and confines the search scope within each bucket, thus reducing the computation of FPS. Regrettably, all these methods introduce accuracy loss on PCNNs, which is unacceptable for safety-critical applications like autonomous driving. PointAcc [7] unifies FPS and neighbor search on a set of components which reduces data movement costs between heterogeneous architectures without accuracy loss. However, it serializes stages on the critical path due to structural dependencies.

III. Algorithm Design

A. Fusion-FPS

To parallel the execution flow while reducing the workload, we introduce Fusion-FPS, which eliminates FPS across all layers to once and breaks down the layer boundaries to fully-utilize the potential parallelism. Given the observation that the current layer's centroid set is a subset of the preceding layer's in FPS, we propose to perform FPS only once in the 1st layer to obtain a complete centroid set C_0 and inherit the centroids of the subsequent layers from the 1st layer. Concretely, for subsequent $m-1$ layers, their centroid sets $C_i, i \in [1, m)$ are generated directly by truncating the leading n_i centroids from C_0 separately, avoiding the subsequent layers' FPS.

Fusion-FPS changes PCNN execution flow, providing opportunities to parallel neighbor search stages across layers.

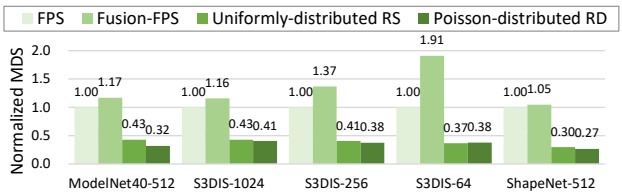

Fig. 2. Sampling quality of different algorithms

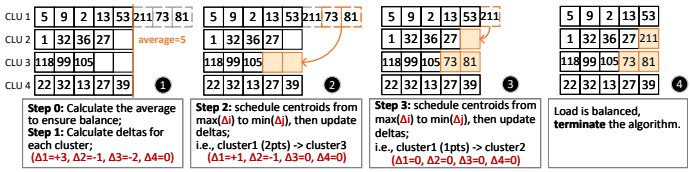

Fig. 3. An example of balanced K-means clustering algorithm.

As depicted in Fig. 1(a), since the neighbor search of each layer depends solely on the current layer's centroid set, it can be launched right after the 1st layer's FPS is finished. As a consequence, neighbor search stages of all layers can be advanced and initiated simultaneously.

$$MDS = \sum_{i=1}^{npoint} \max_{j\in[1,N]} (\min_{k\in[1,i]} (dist(P_j - P_{C_k}))) \quad (2)$$

To eliminate concerns about the accuracy impact of Fusion-FPS, we will discuss it in both intuitive and theory in the following. Intuitively, the leading centroids are farther from each other than the subsequent centroids, which tend to portray the object boundaries, showing no sampling shrinkage trend and exhibiting stronger shape representation capabilities. Moreover, the proposed Fusion-FPS is aligned with the classic FPS, which manifests great potential to achieve even better accuracy than FPS. In theory, we design a metric to quantify the sampling quality – the Maximum Distance Sum (MDS), which is a summary of the maximum distances from the centroid set to points in each iteration. It is deemed that the larger the MDS, the farther the centroids are apart from each other, indicating a more uniform distribution of centroids and thus indicating higher sampling quality. The MDS (shown in Eqn. 2) is essentially a variant of FPS (see Eqn. 1), where the outermost $argmax$ in FPS is replaced by a sum operator. Thus, it is reasonable to adopt MDS to validate the quality of Fusion-FPS. As depicted in Fig. 2, our Fusion-FPS outperforms random sampling (RS) from a uniform distribution, RS from a Poisson distribution and classic FPS.

B. Fusion-Computation

To filter redundant feature computations of "Filling Points", we introduce Fusion-Computation. In this strategy, we initiate the process by determining the start position of the "Filling Point". Then, the amount of "Filling Points" can be derived by subtracting from the length of the neighbor list (referred to as k in KNN). Subsequently, feature computation exclusively targets those "non-Filling Points", whereas the output features of "Filling Points" are replicated multiple times from the 1st neighbor when being written back to memory. As depicted in Fig. 1(b), the Fusion-Computation shortens the length of the neighbor list, reducing the input size of MLPs and eliminating the redundant feature computations caused by "Filling Points".

C. Fusion-Aggregation

To alleviate redundant accesses in aggregation, we propose Fusion-Aggregation (Fig. 1(c)), which packs adjacent centroids into a cluster for simultaneous memory accessing and coalesces redundant accesses for shared neighbors. To be aware of adjacent centroids, we leverage K-means clustering [12], which organizes the dataflow of aggregation and feature computation cluster-by-cluster, rather than being disordered as in the previous method. However, Fusion-Aggregation allows multiple clusters to be processed in parallel, while the unaligned size of clusters results in a disastrous load imbalance. In terms of this, we propose a greedy ***balanced K-means clustering*** to rebalance the clustering results produced by K-means.

First, we compute the expected number of centroids, which is defined as the average number to allocate to each cluster to ensure absolute workload balance. Next, we calculate deltas (Δ) between the expected number and cluster sizes (see Fig. 3 ❶). We then schedule points from the cluster with the maximum delta to the cluster with the minimum delta (❷ and ❸). The number of points moved at each step is determined by: $min(\Delta_{max}, \Delta_{min})$. After multiple steps, all clusters reach balance (❹). This max-min greedy strategy tries to move the bulk of points from the same cluster to preserve the spatial locality while satisfying load balance.

IV. FusionArch Architecture

A. Overview

In this section, we propose FusionArch, an architecture to support our algorithms. Fig. 4(a) provides its overview, which comprises a Fusion-FPS engine, a Fusion-Aggregation engine, a neighbor search engine, a Fusion-Computation engine, several on-chip buffers and a system controller. The on-chip buffers are distributed around engines, storing data fetched from external memory and feeding them to consumer engines. The neighbor search engine is built upon a set of parallel distance calculators (DCs), each accompanied by a sorter array, employing a fixed modular design as Point-X [13]. Considering the DC is reused in multiple engines, we augment its architecture with 2 functions: distance calculation for 3D points and delta calculation in balanced K-means clustering. When the DC performs distance calculation, it takes point coordinates as inputs, producing the squared Euclidean distances between them. While the DC calculates deltas, it accepts each cluster size and the expected number of centroids to generate deltas in 3-way parallelism.

B. Fusion-FPS Engine

The Fusion-FPS engine is a hardware implementation of the Fusion-FPS algorithm, as illustrated in Fig. 4(a), which consumes point coordinates and outputs centroid indices. The Fusion-FPS engine incorporates a group of DCs for distance calculation, followed by minimal units selecting the minimum distance for each point. A distance buffer is integrated to accommodate distances, while a maximal unit identifies the maximum value among them. We utilize a counter to accumulate the generated centroid count to determine layer boundaries.

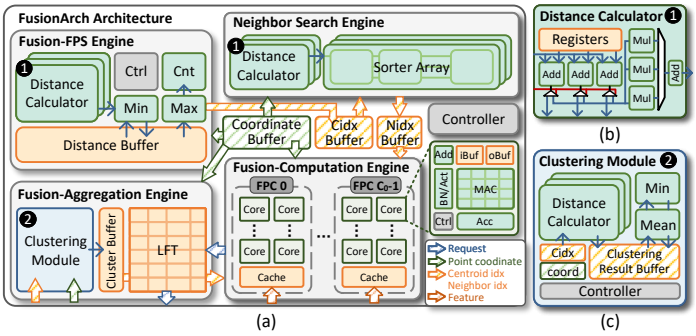

Fig. 4. FusionArch architecture. (a) overview, (b) DC, (c) clustering module.

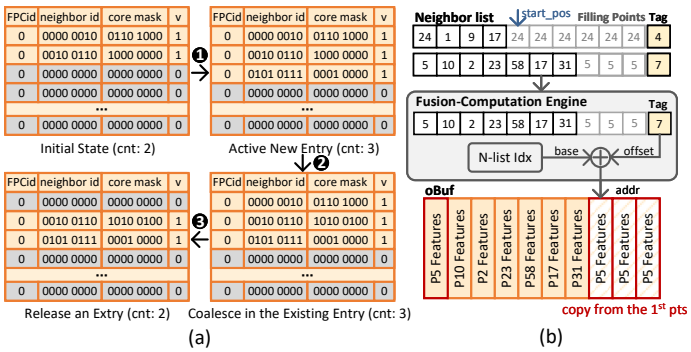

Fig. 5. An example of (a) Fusion-Aggregation, (b) Fusion-Computation.

To implement the control logic, we embed a local controller that handles the generation of centroid sets. Particularly, after FPS generates a complete centroid set, the local controller truncates leading centroids based on specific numbers held in registers to create centroid sets for subsequent layers. Eventually, these centroid sets are stored in the cidx buffer.

C. Fusion-Computation Engine

The Fusion-Computation engine, as shown in Fig. 4(a), follows our Fusion-Computation strategy. It comprises a set of Feature Processing Clusters (FPCs), with each incorporating multiple cores for parallel feature computation and caches for features fetched from external memory. Within each core, an adder is utilized to perform subtractions between coordinates of centroids and their neighbors. The feature computation module includes a lightweight Multiply-Accumulate (MAC) array, a batch normalization-activation unit and an iBuf for feature partial sums (fpsums). In terms of scheduling, as we have transitioned dataflow to a cluster-by-cluster mode, we schedule centroids to FPCs at the cluster-level granularity. Internal FPC, we initially schedule centroids to cores in a round-robin manner, and then assign them to available cores.

To determine the start position of "Filling Points", a tag is attached to each neighbor list to record it while generating from the neighbor search engine. It is readily implemented by detecting the empty slot in the sorter array. Upon the neighbor list loaded onto a core, its tag is parsed by the controller. As a consequence, the feature computation is exclusively performed on those "non-Filling Points", while that of the "Filling Points" are directly borrowed from the 1st neighbor. All features are eventually written back to memory following the tag value, which is utilized as an offset in this context.

Fig. 5(b) presents an example of the Fusion-Computation. Assuming a neighbor list length of 10 is dispatched to a core, with the "Filling Point" starting from index 7. Upon receiving the neighbor list, the feature computation is executed on the first 7 "non-Filling Points" and their output features are written back to memory, utilizing addresses generated on the neighbor list index (N-list Idx) and point locations within the neighbor list. For "Filling Point" 5, the offset address refers to the tag value and the replicated number is determined by subtracting from the neighbor list length. Eventually, the corresponding locations in external memory are populated by copying the features of neighbor 5, located at offset 0. By adopting this strategy, a significant number of redundant computations and memory accesses are bypassed, further improving PCNNs' performance.

D. Fusion-Aggregation Engine

The Fusion-Aggregation engine, illustrated in Fig. 4(a), comprises 3 main components: clustering module, cluster buffer and LFT. The clustering module performs balanced K-means clustering, with its architecture depicted in Fig. 4(c). We leverage DCs to compute distances between points and K cluster centers. Each point is assigned to the nearest center using a minimal unit, and their mapping relationships are stored in the cluster buffer. After each iteration, centers are updated through a mean unit. The K-means algorithm is considered to have converged when assignments change rarely. Notably, the controller handles rebalancing by swapping centroids among clusters within the cluster buffer. It reuses the mean unit to calculate the expected number and employs DCs to compute deltas. In this context, the DC acts as a 3-way subtractor by enabling the pertinent signal, highlighted in red in Fig 4(b), which facilitates direct output of the subtraction results.

Given that parallel cores send numerous requests simultaneously, and redundant memory accesses caused by shared neighbors are common in aggregation. To boost the Fusion-Aggregation in hardware, enabling cores to access features with fewer requests in a non-blocking manner, we introduce LFT. As depicted in Fig. 4(a), LFT is a table with 4 fields in each entry: FPC index, neighbor index, core mask and a valid bit. When a core loads neighbor features by index, LFT intercepts the index to check if it exists. If found, it implies the request is redundant and memory coalescing should be performed. Then, the core mask is updated based on the core indices associated with this request. Otherwise, a new entry is activated, and the request is sent to the next level of memory. After memory response, the entry is released by toggling the valid bit to zero. Accordingly, the returned data is scattered to cores via a chain NoC [13], following the core mask. This approach effectively mitigates redundant memory accesses of shared neighbors, and directs them to cores that are transparent to external memory.

Fig. 5(a) gives an illustration of our Fusion-Aggregation. Suppose there are 2 valid entries in the LFT. In event ❶, core 3 of FPC 0 accesses neighbor 87's features, whose FPC and neighbor indices are then relayed to the CAM. After a bit-by-bit matching, none of matchlines are activated, indicating the absence of neighbor 87 in the LFT. Notably, we employ

TABLE I
EVALUATION BENCHMARKS AND DATASETS

Benchmarks	Dataset	Task	Scene	Point Scale
PointNet++(c)	ModelNet40	Classification	Object	1K
PointNet++(ps)	ShapeNet	Part Seg	Object	2K
PointNet++(s)	S3DIS	Semantic Seg	Indoor	4K
PointWeb	S3DIS	Segmentation	Indoor	4K
F-PointNet++	KITTI	Detection	Outdoor	2K

a counter to track active entries, which currently stands at 2, below LFT's capacity. Thus, we create a new entry for neighbor 87, setting the 3rd core mask bit to 1 and the valid bit to 1, increasing the counter by 1. In event ❷, both core 2 and core 5 of FPC 0 request the same neighbor 38. Following a comparison in the CAM, the second matchline is enabled, which means neighbor 38 exists in the LFT. We only update the core mask by toggling the 2nd and 5th bits to 1. Event ❸ refers to a memory response where neighbor 2's features are fetched from external memory. The first matchline, corresponding to neighbor 2, is activated. The core mask is extracted to route data to core 1, 2, and 4 of FPC 0. Simultaneously, the entry is released by setting the valid bit to 0, and the counter decreases by 1.

V. EVALUATION

A. Evaluation Setup

1) ***Benchmarks****:* We adopt 5 representative point-based PCNNs across diverse applications and scenarios as benchmarks shown in Table I. The networks are well-acknowledged and widely used in the field and the corresponding datasets cover various sizes with assorted scenes.

2) ***Architectural Modeling****:* We develop a cycle-accurate simulator to model the behavior of our FusionArch. To obtain precise and reliable memory access latency, we integrate DRAMSim3 [14] to model external memory, which is configured as 8GB 3200Mhz DDR4 with a bandwidth of 25.6GB/s. The area and power of on-chip buffers are modeled through CACTI [15]. To evaluate FusionArch under different resource conditions, we propose 2 implementations: a full version named Fusion.Server with 4096 MACs and 334.6KB SRAMs and a stripped-down version named FusionArch.Edge with only 256 MACs and 83.8KB SRAMs. Our design is verified under 28nm technology on 1GHz frequency. ASIC design parameters are compared in Table II.

3) ***Baselines****:* We select 2 kinds of hardware platforms as baselines, each representing different scales of computational power and bandwidth constraints: server-level and edge-level. For server-level, we compare FusionArch.Server against Intel Xeon Gold 6226R, NVIDIA A100 PCIe 80GB and PointAcc.Server. Regarding edge-level platforms, we compare FusionArch.Edge with NVIDIA Jetson AGX Xavier, Mesorasi, PointAcc.Edge and PRADA.

B. Accuracy

To validate whether Fusion-FPS affect the accuracy of PCNNs, we evaluate the PCNNs that integrate Fusion-FPS using PyTorch on GPU across various benchmarks. As shown in Fig. 6, Fusion-FPS outperforms the classic FPS 2.02% at most, while also achieving a notable reduction in computations ranging from 20% to 50%. Specially, F-PointNet++ exhibits the most significant improvement in accuracy, which indicates that Fusion-FPS can achieve better accuracy as the scale of points increases. Moreover, Fusion-FPS reduces half of the sampling operations in DensePoint, a PCNN characterized by its extensive layers, indicating that FusionArch achieves substantial performance gains when PCNNs confront a large number of layers and desired centroids.

TABLE II
HARDWARE CONFIGURATIONS

Accelerator	PointAcc.Server	**FusionArch.Server**	PointAcc.Edge	PRADA	**FusionArch.Edge**
MACs	64×64=4096	8×16×32=4096	16×16=256	16×16=256	2×8×16=256
SRAM (KB)	776	334.6	274	98	83.8
Frequency	1GHz	1GHz	1GHz	1GHz	1GHz
DRAM Bandwidth	256GB/s	256GB/s	17GB/s	256GB/s	25.6GB/s
Technology	28nm	28nm	28nm	28nm	28nm
Area (mm2)	14.5	14.1	1.5	2.0	1.3
Effective Throughput	8TOPS	20.8TOPS	512GOPS	625GOPS	1.5TOPS

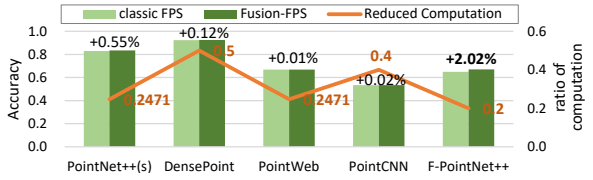

Fig. 6. Accuracy and computation reduction of Fusion-FPS.

C. Speedup and Energy Savings

Fig. 7 presents the speedup and energy savings achieved by the FusionArch.Server compared to PointAcc.Server, GPU, and CPU. On average, FusionArch offers 2.6×, 5.6×, 13× speedup and 17×, 22×, 62.4× energy savings, respectively. Notably, the most significant performance improvement is observed in PointNet++(s), which is intrinsically dominated by aggregation and feature computation. Therefore, it fully exhibits the benefits of our Fusion-Aggregation and Fusion-Computation. For edge-level evaluation, our FusionArch achieves speedups of 2.4×, 2.9×, 5.3×, 5.5×, along with energy savings of 4.4×, 7.2×, 12.4×, 11.5× when compared to PRADA, PointAcc.Edge, Mesorasi, and GPU, respectively. As depicted in Fig. 8, FusionArch exhibits the most notable improvement over Mesorasi. This can be attributed to the fact that Mesorasi only reduces the feature computation latency while increasing the cost of aggregation, which becomes more crucial in resource-limited edge-level devices. In comparison to PointAcc, our FusionArch gains nearly 3× speedup, this is because PointAcc lacks optimization for the sequential execution flow and neglects redundant feature accesses which leads to heavy memory costs. The experiments demonstrate that our FusionArch delivers strong performance across different PCNNs in both server and edge conditions, which is primarily attributed to the optimization of multiple stages along the critical path.

D. Ablation Study

Fig. 9 demonstrates the ablation study conducted on a baseline accelerator modeling from FusionArch.Edge, showcasing the performance gains of the Fusion-FPS, Fusion-Computation and Fusion-Aggregation across various benchmarks. Given that the performance of PCNNs is severely constrained by limited

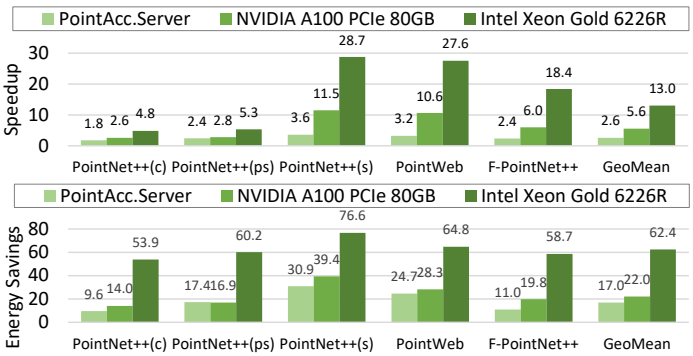

Fig. 7. Performance gain over server-level platforms.

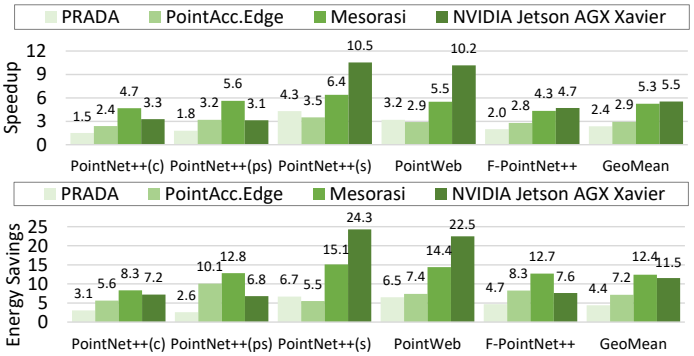

Fig. 8. Performance gain over edge-level platforms.

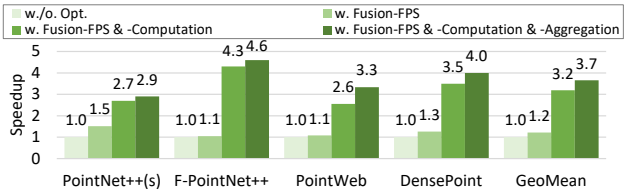

Fig. 9. Ablation study with various optimizations.

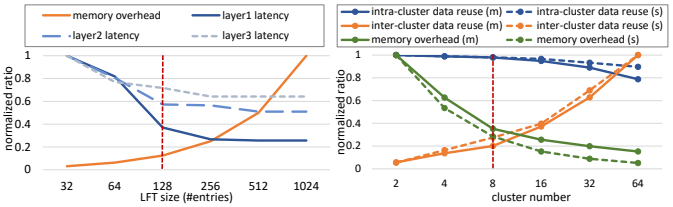

Fig. 10. Exploration of (a) LFT entries, and (b) clusters.

bandwidth and computing powers of the edge platform, the bottlenecks has shifted to aggregation and feature computation. This is the reason why Fusion-FPS individually offers only $1.2\times$ speedup on average, which shows fewer performance gains compared to that expected on GPU. While Fusion-Computation and Fusion-Aggregation manifest a sharp speedup of $3.7\times$, owning to both altered performance bottlenecks and the intrinsic heavy workload of these stages.

E. Exploration

*1) **The size of LFT**:* We leverage LFT to coalesce redundant memory accesses in aggregation. LFT size is a critical parameter that may affect performance and needs to be carefully determined. We carry out this experiment by varying the size of LFT from 32 to 1024 across different layers in PointNet++. As illustrated in Fig. 10(a), the most significant reduction in latency occurs within the range of (64,128] for nearly all layers. Conversely, the memory overhead exhibits a sharp increase beyond a size of 128. To balance the latency and memory overhead, we determine the optimal LFT size to 128.

*2) **The number of clusters**:* We utilize balanced K-means clustering to enhance data reuse within each FPC. The number of clusters along with that of FPCs are critical to the performance. We determine the optimal number of clusters by varying the number of clusters and evaluating the clustering quality. The experiment is conducted on 2 datasets (i.e., ModelNet40 and S3DIS) under different point-scale, as depicted in Fig. 10(b). We tend to maximize intra-cluster data reuse while minimizing inter-cluster data sharing, and simultaneously ensuring an acceptable memory overhead. The results indicate that setting C_0 to 8 enables us to strike a balance between these metrics.

F. Conclusion

This paper first alters PCNN's sequential execution flow by proposing a Fusion-FPS algorithm. Then, to bypass redundant feature computations of "Filling Points", we introduce Fusion-Computation. Furthermore, we optimize memory accessing by leveraging the spatial locality of centroids and coalescing redundant accesses of their common neighbors. In support of algorithms, we propose FusionArch architecture, which outperforms the latest CPU, GPU and start-of-the-art PCNN accelerators. Our proposal exhibits excellent generality by optimizing various stages to eliminate potential bottlenecks that exist across different PCNNs.

VI. ACKNOWLEDGEMENTS

This work is partly supported by the National Natural Science Foundation of China (Grant No. 62202288, 61972242). We thank Sanechips's support.

REFERENCES

[1] Q. et al, "Pointnet++: Deep hierarchical feature learning on point sets in a metric space," *NeurIPS*, vol. 30, 2017.
[2] Z. et al, "Pointweb: Enhancing local neighborhood features for point cloud processing," in *CVPR*, 2019, pp. 5565–5573.
[3] L. et al, "Densepoint: Learning densely contextual representation for efficient point cloud processing," in *CVPR*, 2019, pp. 5239–5248.
[4] Q. et al, "Frustum pointnets for 3d object detection from rgb-d data," in *CVPR*, 2018, pp. 918–927.
[5] E. et al, "The farthest point strategy for progressive image sampling," *TIP*, vol. 6, no. 9, pp. 1305–1315, 1997.
[6] F. et al, "Mesorasi: Architecture support for point cloud analytics via delayed-aggregation," in *MICRO*. IEEE, 2020, pp. 1037–1050.
[7] L. et al, "Pointacc: Efficient point cloud accelerator," in *MICRO*, 2021, pp. 449–461.
[8] S. et al, "Prada: Point cloud recognition acceleration via dynamic approximation," *DATE*, vol. 30, 2023.
[9] Y. et al, "Edgepc: Efficient deep learning analytics for point clouds on edge devices," in *ISCA*, 2023, pp. 1–14.
[10] P. et al, "K-nearest neighbor," *Scholarpedia*, vol. 4, no. 2, p. 1883, 2009.
[11] H. et al, "Quickfps: Architecture and algorithm co-design for farthest point sampling in large-scale point clouds," *TCAD*, 2023.
[12] H. J. A. et al, "Algorithm as 136: A k-means clustering algorithm," *J R Stat Soc Ser A Stat Soc*, vol. 28, no. 1, pp. 100–108, 1979.
[13] Z. et al, "Point-x: A spatial-locality-aware architecture for energy-efficient graph-based point-cloud deep learning," in *MICRO*, 2021, pp. 1078–1090.
[14] L. et al, "Dramsim3: A cycle-accurate, thermal-capable dram simulator," *CAL*, vol. 19, no. 2, pp. 106–109, 2020.
[15] M. et al, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.