SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

November 2019, Vol. 62 212102:1–212102:11
https://doi.org/10.1007/s11432-019-9932-3

Accelerating DNN-based 3D point cloud processing for mobile computing

Bosheng LIU[1,2], Xiaoming CHEN[1*], Yinhe HAN[1*], Jiajun LI[1,2], Haobo XU[1,2] & Xiaowei LI[1]

[1]*State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing* 100190*, China;*
[2]*University of Chinese Academy of Sciences, Beijing* 100190*, China*

Received 13 April 2019/Accepted 3 June 2019/Published online 19 September 2019

Abstract 3D point cloud data, which are produced by various 3D sensors such as LIDAR and stereo cameras, have been widely deployed by industry leaders such as Google, Uber, Tesla, and Mobileye, for mobile robotic applications such as autonomous driving and humanoid robots. Point cloud data, which are composed of reliable depth information, can provide accurate location and shape characteristics for scene understanding, such as object recognition and semantic segmentation. However, deep neural networks (DNNs), which directly consume point cloud data, are particularly computation-intensive because they have to not only perform multiplication-and-accumulation (MAC) operations but also search neighbors from the irregular 3D point cloud data. Such a task goes beyond the capabilities of general-purpose processors in real-time to figure out the solution as the scales of both point cloud data and DNNs increase from application to application. We present the first accelerator architecture that dynamically configures the hardware on-the-fly to match the computation of both neighbor point search and MAC computation for point-based DNNs. To facilitate the process of neighbor point search and reduce the computation costs, a grid-based algorithm is introduced to search neighbor points from a local region of grids. Evaluation results based on the scene recognition and segmentation tasks show that the proposed design harvests 16.4× higher performance and saves 99.95% of energy than an NVIDIA Tesla K40 GPU baseline in point cloud scene understanding applications.

Keywords deep neural network acceleration, point cloud data, neighbor point search, mobile robotics, hardware architecture

Citation Liu B S, Chen X M, Han Y H, et al. Accelerating DNN-based 3D point cloud processing for mobile computing. Sci China Inf Sci, 2019, 62(11): 212102, https://doi.org/10.1007/s11432-019-9932-3

1 Introduction

With the rapid and significant advances of 3D sensors and deep learning techniques, various mobile robotic applications such as autonomous driving and humanoid robots are being built and covered extensively in the news. For example, industry leaders such as Google, Uber, Tesla, and Mobileye [1, 2] are currently building the prototyping of reliable autonomous driving systems self-driving on public roads across the United States. A practical instance is the Boston Dynamics Atlas humanoid robot [3], which has the essential 3D light detection and ranging (LIDAR) sensor components as its eyes and exhibits impressive capabilities in walking outdoors over irregular terrains.

Owing to the reliable depth information representations of the point cloud data for accurate location and shape characteristics, the point cloud data [4–6] produced by 3D sensors such as LIDAR are playing

* Corresponding author (email: chenxiaoming@ict.ac.cn, yinhes@ict.ac.cn)

an increasingly significant role in mobile robotics. Though point cloud data carry accurate depth information, the proper reaction from mobile robotics relies fundamentally on a correct scene understanding of the captured 3D point cloud data. Recently, deep neural networks (DNNs) such as PointNet [5] and PointNet++ [6], which directly consume point cloud data, have achieved significant breakthroughs in scene understanding such as object recognition and semantic segmentation. However, these point-based DNNs are particularly challenging, because they not only need to tackle massive multiplication-and-accumulation (MAC) operations, but also consume a large amount of computation time in neighbor point search from the irregular point cloud data. It goes beyond the capabilities of general-purpose processors in real-time as the scales of both point cloud data and DNNs increase from application to application.

To combat this challenge, hardware solutions such as graphic processing units (GPUs) can be deployed to accelerate the point-based DNNs. Unfortunately, GPUs are power hungry and are not well suited to mobile robotics, which have a limited energy budget. Application-specific integrated circuit based (ASIC) accelerators, such as Diannao [7] and Eyeriss [8], can achieve excellent performance for DNNs [9,10] with regular 2D pixel images. However, these pixel-based DNN accelerators cannot be smoothly deployed for point-based DNNs, as they cannot retrieve well the neighbor points from the irregular point cloud data for convolution calculation. Though Lin et al. [2] provide efficient acceleration for mobile robotics applications such as autonomous driving, their solution is based on a vision-based data representation such as images, instead of the irregular point cloud data. In sum, none of the existing hardware DNN accelerators can provide efficient acceleration for point-based DNNs.

In this paper, we propose a configurable architecture, PointPU, for accelerating the point-based DNNs in scene understanding applications. To facilitate fundamental neighbor point search, a grid-based algorithm is introduced to search neighbor points from a local region of grids to reduce the computation cost. PointPU configures the architecture accordingly to make it well suited to both neighbor point search and MAC computation to achieve high-throughput and energy-efficient processing. The evaluation results show that PointPU achieves 16.4× higher performance and 99.95% energy saving on average when compared with a high-performance NVIDIA Tesla K40 GPU baseline. The key contributions of this study are summarized as follows.

- We propose a point-based configurable accelerator that can be dynamically switched to become well suited to the parallelism for both neighbor point search and MAC computation in order to achieve high throughput.
- To facilitate neighbor point search, a grid-based algorithm is developed to search neighbor points from a local region of grids to reduce the large amount of computation.
- We evaluate the proposed architecture by Design Compiler and PrimeTime-PX, and then compare it with a high-performance K40 GPU baseline on 3D scene recognition and segmentation applications. The results reveal the higher performance and better energy saving.

2 Related work

3D data representation. There are mainly three kinds of 3D data representations: multi-view images [11,12], volumes [13,14], and point cloud [5,6]. The multi-view image representation projects 3D shapes with a collection of their rendered views on 2D pixel images. Nevertheless, this kind of data representation not only discards valuable depth information but also incurs a heavy burden of computation to tackle all the rendered images. The volume-based data representation utilizes a regular volumetric model to cover the raw point cloud data into fixed 3D shapes and requires a massive amount of memory for the storage of the volumetric shapes. Conversely, point cloud data are the raw 3D sensor outputs with reliable depth information, implying that they are the most popular and accessible 3D data representation.

DNNs for point cloud data. Until recently, a series of DNNs that directly consume point cloud data achieved breakthroughs in scene understanding [5,6,15]. PointNet [5], a pioneer in this field, adopts a symmetric function (e.g., max pooling) and spatial transformer networks (consisting of fully-connected

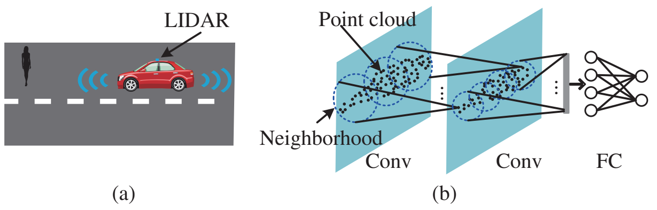

Figure 1 (Color online) Illustrative example of point-based DNNs for point cloud data. (a) An instance of mobile robotic applications; (b) a point-based DNN.

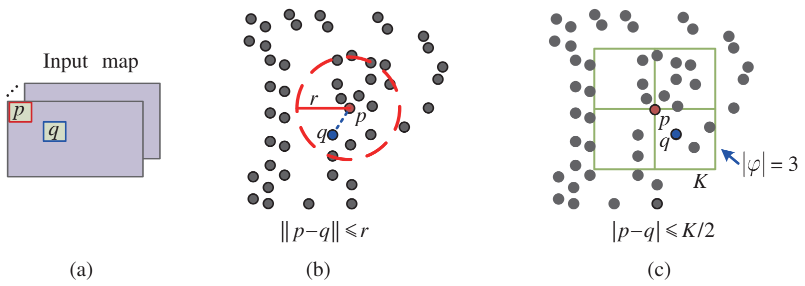

Figure 2 (Color online) Neighbor pixels/points. (a) Neighbor pixels are regular in conventional Conv layers. (b) Irregular neighbors within a radius r in a convolution-like layer. (c) Irregular neighbors within a kernel size K in a pointwise layer. Both (b) and (c) are illustrated for a 2D example and can be smoothly extended to 3D metric space according to the formulations in (b) and (c), respectively.

(FC) layers) to capture the 3D features of point cloud data. PointNet++ [6] extracts the local features of point cloud data by a convolution-like operation and a hierarchical PointNet. Hua et al. [15] build a convolutional neural network (CNN) to achieve promising accuracy improvements by tackling point cloud data with a pointwise convolution operation. Both the convolution-like and pointwise computations include both neighbor point search and MAC operations for irregular point cloud data.

Hardware acceleration. Previous ASIC-based accelerators, such as Diannao [7], Eyeriss [8], and C-Brain [16], provide efficient accelerations to achieve high throughput for DNNs with 2D regular data such as images. For example, C-Brain offers three types of parallelism, inter-kernel, intra-kernel, and hybrid, to enable flexible schedules for different shapes of CNNs with images as inputs. However, ASIC-based accelerators cannot smoothly tackle point-based DNNs, which involve not only MAC operations but also neighbor point search, because of the irregular input point cloud data. Lin et al. [2] focused on accelerating vision-based DNNs, instead of the point-based DNNs, in mobile robotic applications. Until now, there has been no hardware accelerator that is specially designed for 3D point cloud data. The objective of our work is to provide a hardware architecture that can tackle both neighbor point search and MAC operations for point cloud data. It is a configurable point-based architecture that can offer flexible parallelism with the high-throughput and energy-efficient processing for point-based DNNs.

3 Preliminaries

3D point cloud are a set of irregular point data that cover the surfaces of sensed objects in a 3D space. Figure 1 depicts an example of DNNs with point cloud data in mobile robotic applications. The point cloud data, generated by LIDAR, are consumed by DNNs to provide an accurate scene understanding. Here, we introduce the convolutional (Conv) and other major layers of point-based DNNs.

(1) Conv layers. Conv layers are widely deployed in point-based DNNs for local feature extraction and to tackle irregular point cloud data, as shown in Figure 1(b). The point-based Conv layer is different from conventional Conv layers, which consume regular pixels such as images, as shown in Figure 2(a). Typically, a Conv layer comprises both neighbor point search for capturing a local region and MAC operations for convolution computation. We describe two typical Conv layers for point cloud data: convolution-like and pointwise layers. Both are able to capture local features from point cloud data.

(i) Convolution-like layers. A convolution-like layer (utilized in Pointnet++ [6]) includes two key operations: searching neighbors from the input points and extracting features from the neighbor points with few FC layers. The overlapped neighbor points for different outputs provide opportunities for local feature extraction, as shown in the Conv layer of Figure 1(b). PointNet++ utilizes the Ball Query algorithm, as shown in Figure 2(b), for neighbor point search.

The Ball Query algorithm searches k neighbor points within a radius size r for an input point p. A point q with a Euclidean distance shorter than r is selected as one of the neighbors. Ball Query repeats the search process of the neighbor points if the total number of the found neighbor points within r is less than k, such that it can capture a fixed number of neighbors.

(ii) Pointwise layers. A pointwise layer [15] performs convolution operations according to a kernel size K, as shown in Figure 2(c). The kernel partitions the neighbor points into blocks (e.g., 4 blocks in Figure 2(c)). The points in each block are regarded as the neighbors (comparison is based on Manhattan distance) of a block of the kernel and share the same weight parameters in convolution operations. The details of the convolution calculation from the l-th to $(l+1)$-th layers are formulated as

$$\begin{cases} 1: \varphi_i(k) = \text{Neighbor}(P, K), \\ 2: x_i^{(l+1)} = \sum\limits_k w_k \dfrac{1}{|\varphi_i(k)|} \sum\limits_{p_j \in \varphi_i(k)} x_j^l, \end{cases} \tag{1}$$

where the Neighbor function finds the neighbor points for each block of the kernel. P is the input point cloud data, K is the kernel size, and $\varphi_i(k)$ denotes the neighbor points of the k-th block of the kernel centered at point i $(1 \leqslant i \leqslant \text{Count}(P))$. Line 2 of (1) exhibits the MAC operation. x_j^l is the characteristic of an input point j at the l-th layer and p_j is the coordinate of j. The neighbor points in the k-th block $(p_j \in \varphi_i(k))$ share the same weight parameter (w_k). $|\varphi_i(k)|$ is the number of neighbor points in the k-th block.

(2) FC and pooling layers. Both the FC and pooling layers for point cloud data are similar to the conventional FC and pooling layers for pixel images, because the coordinates of point cloud data can be regarded as a portion of the input channels. The pooling operation retrieves the max characteristics of the input point cloud data [5, 6, 15].

We focus on accelerating the above layers in point-based DNNs. A grid-based algorithm is provided to search the neighbor points from a local region of grids, so that we can avoid searching neighbors from all input points to reduce the computation cost. We further introduce a configurable architecture that can flexibly suit both neighbor point search and MAC operations and thereby achieve efficient acceleration for point-based DNNs.

4 Grid-based neighbor point search

In this section, we describe a grid-based algorithm for neighbor point search in both convolution-like and pointwise layers, as shown in Algorithm 1, which includes two key operations: initialization and retrieval. The initialization process partitions the input points into equal-length grids. The retrieval process discovers the neighbor points from a local region of grids.

The initialization process builds grids based on a grid size g, as shown in lines 1–5. The process first finds the boundaries of the input points, and then partitions the input points into equal-length grids based on the boundaries and g (lines 3 and 4). Finally, it records the points within each grid and their counts. Details of the initialization results are illustrated in Figure 3(a).

The grid-based retrieval process for neighbor point search of convolution-like layers (Ball Query) is depicted in lines 9–20. The local regions of the grids between $-r/g$ and r/g centered to the given point p are enumerated (line 10). The neighbor grids within r are indexed according to the address addr in line 13. The retrieval operation (line 13) returns the points and their count of a neighbor grid. The operations in lines 14–18 can eliminate the points located within the neighbor grids but out of the neighbor distance r. The grid-based neighbor point search for pointwise layers is shown in lines 21–36. The local region

Algorithm 1 Grid-based neighbor point search

```
1: //Initialization
2: Inputs: $P$: input points, $g$: a grid size;
3: Find the minmun/maximum boundaries of $P$ ($\langle Mn_x, Mx_x\rangle, \langle Mn_y, Mx_y\rangle$);
4: Build grids based on $P$, $Mn/Mx$, and $g$;
5: Store grids of points, the start address and count of each grid;
6: //Retrieval
7: Inputs: $G$: grid based points; $\langle S, C\rangle$: start address and count of grids; $p(p_x,p_y)$: a center point; $g$: a grid size; $r$: a
   radius size; $K(K_x,K_y)$: a kernel size;
8: Outputs: out: neighbor results (out = 0), cnt: their count (cnt = 0);
9: if (Ball Query) then
10:    for $x_i = (-r/g) : r/g$; $y_i = (-r/g) : r/g$ do
11:       $p_{gx} = p_x + g.x_i$, $p_{gy} = p_y + g.y_i$;
12:       if $Mn_x \leqslant p_{gx} \leqslant Mx_x$, $Mn_y \leqslant p_{gy} \leqslant Mx_y$ then
13:          addr = $(p_{gx} - Mn_x)/g + ((p_{gy} - Mn_y)/g).(Mx_x - Mn_x)/g$; $t$_out = Retrieve(addr, $S$, $C$, $G$);
14:          for $i$=1 : Count($t$_out) do
15:             if $\|p - t\_\text{out}(i)\| < r$ then
16:                out+ = $t$_out($i$); cnt++;
17:             end if
18:          end for
19:       end if
20:    end for
21: else if (Pointwise) then
22:    for $k_x$= 1 : $K_x$; $k_y$ = 1 : $K_y$ do
23:       $p_{gx} = p_x - K_x/2 + k_x$, $p_{gy} = p_y - K_y/2 + k_y$;
24:       for $x_i = (-1/(2g)) : (1/(2g))$, $y_i = (-1/(2g)) : (1/(2g))$ do
25:          if $Mn_x \leqslant p_{gx} + x_i.g \leqslant Mx_x$, $Mn_y \leqslant p_{gy} + y_i.g \leqslant Mx_y$ then
26:             $addr = (p_{gx} + x_i.g - Mn_x)/g + ((p_{gy} + y_i.g - Mn_y)/g).(Mx_x - Mn_x)/g$;
27:             $t$_out = Retrieve (addr, $S$, $C$, $G$); out+ = $t$_out; cnt+ = Count($t$_out);
28:             for $i$ = 1 : Count($t$_out) do
29:                if $|p - t\_\text{out}(i)| \leqslant K/2$ then
30:                   out+ = $t$_out($i$); cnt++;
31:                end if
32:             end for
33:          end if
34:       end for
35:    end for
36: end if
```

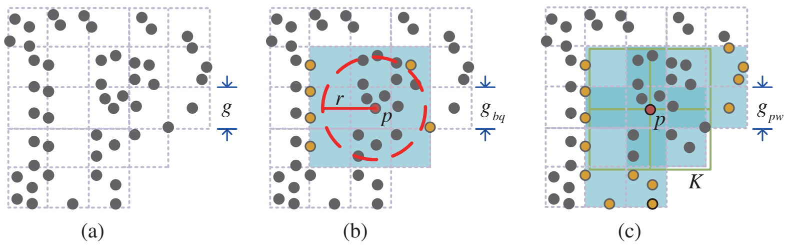

Figure 3 (Color online) Illustrative examples of grid-based neighbor point search. (a) Initialization; (b) grid-based neighbor point search for convolution-like layers; (c) grid-based neighbor point search for pointwise layers.

of the grids for neighbor point search can be enumerated by the loops of both the blocks of the kernel (line 22) and the grids in each block (line 24). The points within each block are picked up as neighbors in lines 28–32. In sum, the grid-based algorithm searches the neighbors from a local region of grids instead of all input points, so that it can effectively reduce the computation cost.

Figures 3(b) and (c) illustrate the grid-based retrieval results for neighbor point search in both convolution-like and pointwise layers. g_{bq} and g_{pw} are the grid sizes for the two layers, respectively. The grid-based algorithm searches the neighbor points from local region grids (colored blue). The points (colored orange) located in the neighbor grids (blue region) but out of the neighbor boundaries can be eliminated during the distance comparisons in Algorithm 1 (lines 14–18 and 28–32). Therefore, the grid-based neighbor point search does not affect the accuracy of the point-based DNNs. Nevertheless, the larger sizes of both g_{bq} and g_{pw} imply a larger local region for the retrieval, incurring more redundant neighbor points (colored orange) in comparison. In other words, the smaller grid size is effective for reducing the comparison. According to the above analysis, we use a small grid size in our implementation,

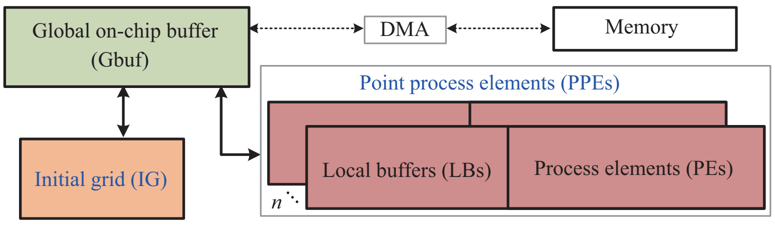

Figure 4 (Color online) PointPU architecture with n tiles of PPEs.

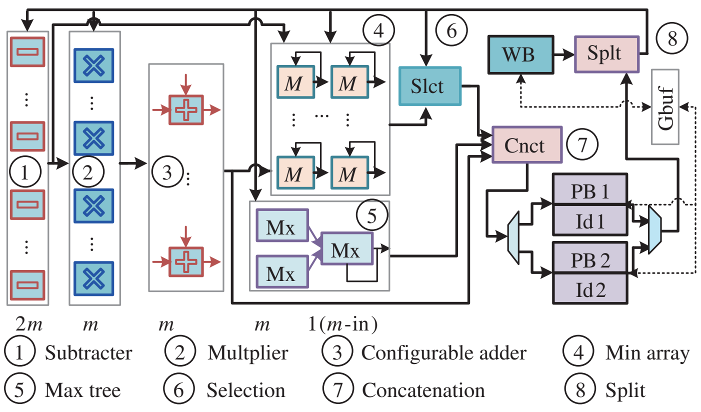

Figure 5 (Color online) One tile of PPEs with local buffers and processing elements.

where $g_{bq} = 0.1 \times r$ and $g_{pw} = 0.1$.

5 PointPU architecture

In this section, we describe the proposed PointPU architecture. We first outline the overall architecture, and then describe the components in detail. Finally, we exhibit the detailed schedules for neighbor point search for both convolution-like and pointwise layers.

5.1 PointPU architecture overview

Figure 4 describes the overall architecture of PointPU for point-based DNN inference accelerations and includes three main components: point processing elements (PPEs) with n tiles, a global on-chip buffer (Gbuf), and an initial grid (IG). Each tile of PPEs comprises not only the processing elements (PEs) for computation but also local buffers for data storage. Gbuf contains banks of on-chip memory, and exchanges data with the off-chip memory through a direct memory access (DMA) component and transfers data with both IG and the local buffers for initialization and computation, respectively. IG builds grids on-the-fly for the initialization of grid-based neighbor point search.

5.2 PointPU components

5.2.1 *Point process elements*

Figure 5 depicts the block diagram of a tile of PPEs, which includes both local buffers for data storage and PEs for computation. The local buffers comprise the point buffers (PB1 and PB2), index buffers (Id1 and Id2), and a weight buffer (WB). PB1 and PB2 alternately store the input/output points. In each grid, the points are accommodated consecutively. Id1 and Id2 alternately accommodate both the start addresses and their counts of grids for input/output points. Splt and Cnct respectively split the inputs and concatenate the outputs, so that the data exchange between local buffers and PEs can be performed smoothly. Slct mainly selects the neighbor points and generates their counts. Slct enables (1) the selection of the neighbor points and generation of the counts based on the distance comparison

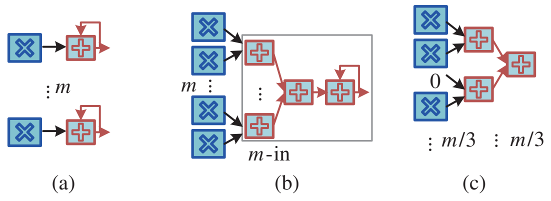

Figure 6 (Color online) Three MAC architectures are delivered by constructing the configurable adder array to three shapes. (a) One-to-one, (b) m-to-one, and (c) three-to-one.

results from the Min array, and (2) the repetition of the search of neighbor points when the total number is less than k. The latter function is utilized in Ball Query.

Processing elements. The PEs in a tile of PPEs include $2m$ subtractors, m multipliers, m configurable adders, a Min array with m Min elements, and an m-input (denoted by m-in) max-tree. The m multipliers, together with the m configurable adders, can be configured for the computation of both neighbor point search and MAC operations. The Min array is organized as an $(m/3) \times 3$ array (m is a multiple of 3). The Min array can perform (a) at most m Min operations for neighbor point search in pointwise layers or (b) Min operations with $m/3$ inputs from the adder components for neighbor point search in convolution-like layers. The max-tree component generates the maximum results for the max pooling operation.

(i) Configurable MAC architecture. Figure 6 depicts three shapes of the configurable MAC architecture, which are developed by constructing the interconnection of both the multipliers and adders. First, the MAC architecture can be organized by configuring each multiplier to an adder, as shown in Figure 6(a), to calculate all the convolutions of an output point data in the same MAC unit, which suits the parallelism of computing inside a kernel step-by-step, such as the intra-kernel parallelism of C-Brain [16]. Second, the adder array can be built as an m-input adder-tree for the MAC architecture, as shown in Figure 6(b). In this case, the m outputs of the multipliers are added together to generate the final outputs. This architecture is effective for the parallelism, such as the inter-kernel parallelism of C-Brain, in computing DNNs with deep channels. Third, the adder array can be configured to $m/3$ groups of MAC operators with a 4-input adder-tree in each, as shown in Figure 6(c), which are utilized for the distance comparison in Ball Query. In sum, the configurable MAC architecture can flexibly suit different parallelisms in both convolution calculation and neighbor point search.

(ii) Local buffers. To supply sufficient inputs for the PEs of a tile of PPEs, both PB1 and PB2 provide at most $2m$ point data in a cycle, whereas Id1 and Id2 need to provide at most one output. WB offers at most m weight parameters. In our design, the local buffers in each tile of the PPEs occupy 4 KB. Specifically, PB1 and PB2 are 1 KB, both Id1 and Id2 are 0.5 KB, and WB occupies 1 KB. The local buffers, together with Gbuf, are organized in a hierarchical fashion for the point/weight storages to facilitate the local buffers to provide sufficient inputs for the tiles of the PPEs. Because the local buffers are private to the PPEs, different tiles can perform computations independently to avoid performance decline, which is caused by the different counts of the neighbors (i.e., $|\varphi|$ in (1)).

(iii) Configurable inputs and outputs. Splt and Cnct enable to smoothly exchange data between the PEs and local buffers. After taking at most $2m$ input points from the local buffers, Splt allows (1) directly transferring inputs to subtractors and (2) providing half of the $2m$ input points sequentially to the PEs, such as the Min array and multipliers. On the other hand, for m input weights, Splt enables sending all or one of them to the multipliers for MAC calculations. In contrast, Cnct gathers the outputs before storing them back in the local buffers. Cnct can (a) combine one output from the Max array, (b) collect m outputs from the configurable adder array, and (c) gather at most m outputs from the Slct component. Cnct also adopts a small look up table for the division operation in line 2 of (1).

5.2.2 *Initial grid*

IG initializes the grids for neighbor point search, then takes the input points from Gbuf and the grid size as inputs, and outputs the grouped points and their count for each grid. The key operations of IG

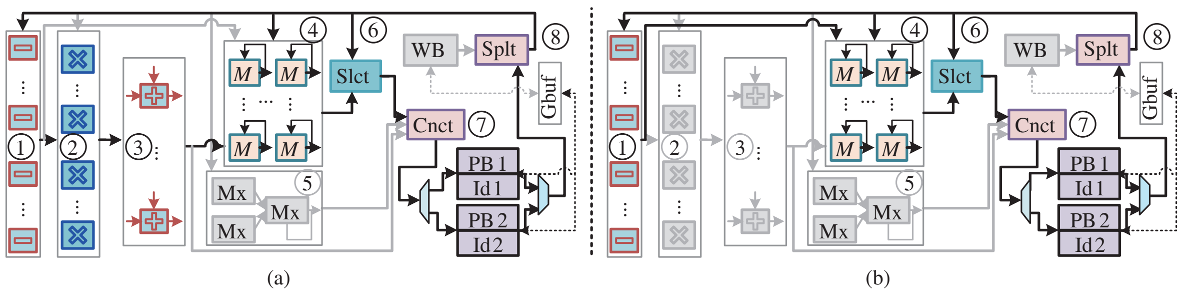

Figure 7 (Color online) Illustrative schedules of PPEs. (a) Neighbor point search in convolution-like layers; (b) neighbor point search in pointwise layers.

include: (a) finding the minimum and maximum coordinates by scanning all input points, (b) partitioning the input points into grids according to the found minimum/maximum boundaries and the grid size, and (c) storing the grouped points and the point counts in the grids back to Gbuf, which then sends the prepared data to the PPEs for computation.

As the output points are generated layer by layer, the proposed architecture needs to build the grids on-the-fly before searching the neighbors for the next layer. As the output channels of a Conv layer can be computed independently, we make full use of the period time in computing the channels of the output points to avoid the computation overhead in building grids. The process of building grids for each output channel is overlapped with the computation of another output channel, so that the process of building grids can be performed in a pipeline-like manner. Consequently, the computation overhead of building grids can be fully hidden to achieve efficient accelerations.

5.3 Schedule of PPEs

Figure 7 illustrates the schedules of the grid-based neighbor point search for both convolution-like and pointwise layers (highlighted in black arrows). Figure 7(a) depicts the schedule of the neighbor point search for convolution-like layers. First, the local buffers provide a grid of input points to both the Slct component (⑥) for selection and the subtractors (①) for computation. Subsequently, the combination of the subtractors and the MAC architecture (①–③) can calculate the Euclidean distances to neighbor points to generate at most $m/3$ results. The Min array (④) compares the results with the radius r (r^2 is used in comparison to avoid the division operation for the Euclidean distance), and then Slct is activated to pick up the neighbor points and generate their counts. Cnct enables the collection of the output points and their counts. Finally, the neighbor points are stored back in PB2/PB1 and their count is sent to Id2/Id1.

Figure 7(b) depicts the schedule of neighbor point search for pointwise layers. The Manhattan distance (Figure 2(c)) of the neighbor points can be calculated by the subtractors (①). This calculation is different from the Euclidean distance one in Figure 7(a) with ①–③. Slct identifies the neighbors by using the distance comparison results from the Min array, then generating their counts. Cnct concatenates the neighbor points and their count, and then stores them back in PB2/PB1 and Id2/Id1, respectively. In sum, the schedule in the PPEs can effectively perform the grid-based neighbor point search for both the convolution-like and pointwise layers.

6 Evaluation

6.1 Measurements

Benchmarks. We use three popular point-based DNNs, PointNet [5], PointNet++ [6], and pointwise CNN [15], and two scene understanding applications, object recognition and semantic segmentation, as the benchmarks, whose characteristics of the benchmarks are described in Table 1 [5, 6, 15, 17, 18]. Specifically, PointNet is the first point-based DNN that can directly consume point cloud data with

Table 1 Characteristics of benchmarks

Description	Network abbreviation	Total layer	Neighbor search layers	Input points	Dataset
PointNet in scene recognition [5]	PN_r	7	0	1024	ModelNet40 [17]
PointNet in semantic segmentation [5]	PN_s	8	0	2048	ShapeNet [17]
PointNet++ in scene recognition [6]	PNpp_r	7	2	1024	ModleNet40
PointNet++ in semantic segmentation [6]	PNpp_s	10	2	2048	ShapeNet
Pointwise CNN in scene recognition [15]	Pw_r	6	4	2048	ModelNet40
Pointwise CNN in semantic segmentation [15]	Pw_s	5	5	4096	S3DIS [18]

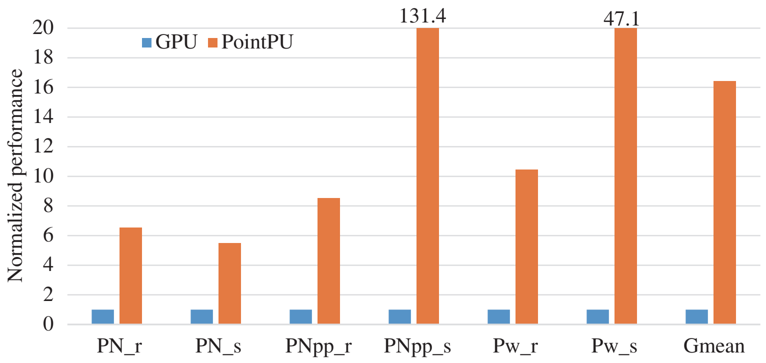

Figure 8 (Color online) Performance compared to GPU baseline.

promising accuracy and mainly contains both FC and pooling layers without neighbor point search or Conv layers. In contrast, PointNet++ enables the capture of local features by convolution-like operations. Pointwise CNN can effectively perform convolution operations to capture local features by pointwise operations.

Baseline. To evaluate our design, we implement a Verilog-based DNN accelerator for 3D point cloud data. We synthesize the Verilog code with a Synopsys Design Compiler by using the 45-nm Nangate technology library. Then, we perform a power analysis through Synopsys PrimeTime-PX. The power consumptions of both the on-chip buffers and off-chip memory are evaluated by Cacti [19]. As there is no existing point-based accelerator, we use a high-performance NVIDIA Tesla K40 GPU as the baseline. To evaluate the performance and energy of the GPU baseline, six workloads are performed in the GPU baseline on basis of the Tensorflow framework and cuDNN Library.

Experimental setting. PointPU suits two types of parallelism, inter-kernel and intra-kernel (first defined in C-Brain [16]), to provide flexible computation for both the neighbor point search and MAC operations in order to achieve efficient point-based DNN accelerations. Specifically, inter-kernel parallelism computes the data from the same kernel position but different input channels in the PEs. Intra-kernel parallelism transfers one or several kernel windows from the same channel to the PEs. The capacity of the local buffers in each tile of PointPU is 4 KB and the Gbuf is 64 KB. The PEs of the PPEs are implemented in 16-bit fixed-point arithmetic with $n = 16$ and $m = 24$.

6.2 Evaluation results

Performance. Figure 8 shows the performance comparison. PointPU on average achieves 16.4× higher performance than the GPU baseline. The higher performance of PointPU comes from two folds. First, PointPU enables the dynamic configuration of the architecture to suit different parallelisms smoothly for high throughput. Second, the grid-based neighbor point search algorithm can search neighbor points from a local region of the grids to reduce the computation cost. PointPU achieves the highest speedup for the PointNet++ benchmark in scene recognition (PNpp_s) at 131.4×, because the proposed grid-based algorithm is able to search neighbor points from a local region of grids, instead of all input points. PointPU gains slightly lower performance for the PointNet benchmarks (PN_r and PN_s at 6.5× and

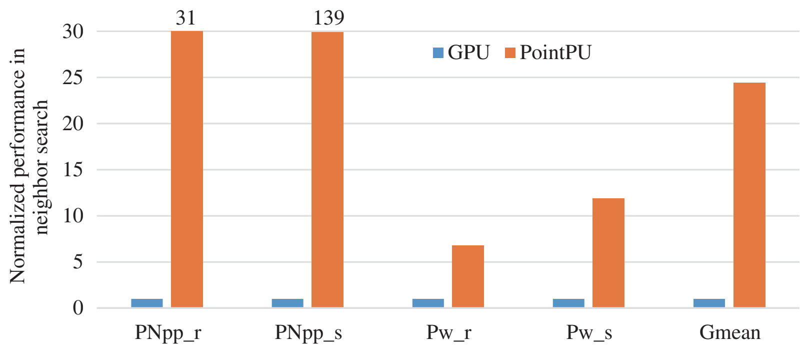

Figure 9 (Color online) Normalized performance against GPU for neighbor search.

Table 2 Normalized energy consumption compared with GPU

	PN_r	PN_s	PNpp_r	PNpp_s	Pw_r	Pw_s	Gmean
GPU	1	1	1	1	1	1	1
PointPU	0.1%	0.1%	0.1%	6.3E−5	0.1%	1.5E−5	0.05%

Table 3 Detailed characteristics of PointPU against to the GPU baseline

Platform	NVIDIA Tesla K40	PointPU
Technology	28 nm	45 nm, 1.1 V
Frequency (MHz)	745	700
Average power (W)	89	0.726
Area	–	2.67 mm^2

5.5×, respectively) than others, because the PointNet benchmarks include only the MAC operations without neighbor point search. In sum, the proposed design can effectively boost the performance for both neighbor point search and MAC computation.

We further evaluate the acceleration for neighbor point search, as shown in Figure 9. The PointNet workloads are excluded because there is no neighbor point search in PointNet. On the geometric mean, the proposed design achieves 24.4× higher performance than the GPU baseline for neighbor point search. PointPU achieves slightly lower performance in neighbor point search for the pointwise layers (Pw_r and Pw_s) than for the convolution-like ones (PNpp_r and PNpp_s), because the GPU baseline can avoid searching neighbors from all input points by reordering the input points in the pointwise layers. For neighbor point search in the convolution-like layers, the performance speedup of PointPU against the GPU baseline for PNpp_s is 4.5× higher than that of PNpp_r as the input points increase to 2× (from 1024 to 2048, as shown in Table 1), because GPU cannot tackle neighbor search from all inputs effectively as the scale of point cloud data increases, where neighbor point search of the GPU baseline reaches 34.8% of the total execution time for PNpp_s. This further confirms that accelerating neighbor point search is critical for point-based DNNs as the scale of point cloud data increases from application to application. To sum it up, PointPU can provide efficient performance boost in neighbor point search for point-based DNNs.

Energy. Table 2 compares the normalized energy with the GPU baseline. On average, PointPU harvests 99.95% energy saving compared with the GPU baseline. The huge energy benefits come from the higher performance and lower power consumption for point-based DNN accelerations.

Table 3 shows the detailed characteristics of PointPU compared with the GPU baseline. The power consumption of PointPU is only 0.8% of that of the GPU baseline while we can achieve higher performance. This substantial amount of energy savings further verifies that ASIC-based accelerators are more suitable than the GPU baseline to mobile robotics with limited power supplies. We further compare PointPU with the state-of-the-art pixel-based accelerator, Diannao [7]. PointPU takes slightly more power (1.50×) because its hierarchical buffer architecture also consumes slightly more hardware overhead, which takes 58.8% of the total power consumption. Nevertheless, PointPU provides efficient neighbor point search and MAC computation accelerations for the point-based DNNs with raw point cloud data.

7 Conclusion

We presented a configurable point-based DNN accelerator to boost the throughput and save energy for mobile robotics in scene understanding applications. The accelerator provides a configurable architecture that can flexiblely adapt itself to different parallelisms for both neighbor point search and MAC computation. To accelerate neighbor point search, a grid-based algorithm is proposed to search the neighbor points from a local region of grids for high throughput and energy saving. Our evaluation shows that the proposed design achieved 16.4× higher performance and 99.95% energy saving compared to a high-performance NVIDIA Tesla K40 GPU baseline.

Acknowledgements This work was supported in part by China Scholarship Council (CSC), in part by National Natural Science Foundation of China (Grant Nos. 61804155, 61522406, 61834006, 61532017, 61521092), in part by Beijing Municipal Science & Technology Commission (Grant Nos. Z171100000117019, Z181100008918006), in part by Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDPB12), in part by the Youth Innovation Promotion Association CAS, in part by the Young Elite Scientists Sponsorship Program by CAST (Grant No. 2018QNRC001), and in part by the Innovative Project of ICT, CAS (Grant No. 5120186140).

References

1 Gallardo N, Gamez N, Rad P, et al. Autonomous decision making for a driver-less car. In: Proceedings of IEEE System of Systems Engineering Conference (SoSE), Waikoloa, 2017. 1–6
2 Lin S C, Zhang Y, Hsu C H, et al. The architectural implications of autonomous driving: constraints and acceleration. In: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems, 2018. 751–766
3 Kuindersma S, Deits R, Fallon M, et al. Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot. Auton Robot, 2016, 40: 429–455
4 Wang X J, Zhou Y F, Pan X, et al. A robust 3D point cloud skeleton extraction method (in Chinese). Sci Sin Inform, 2017, 47: 832–845
5 Qi C R, Su H, Mo K, et al. Pointnet: deep learning on point sets for 3D classification and segmentation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017. 1: 4
6 Qi C R, Yi L, Su H, et al. Pointnet++: deep hierarchical feature learning on point sets in a metric space. In: Proceedings of Neural Information Processing Systems, 2017. 5099–5108
7 Vazou N, Seidel E L, Jhala R, et al. Refinement types for Haskell. SIGPLAN Not, 2014, 49: 269–282
8 Chen Y H, Emer J, Sze V. Eyeriss: a spatial architecture for energy-efficient dataflow for convolutional neural networks. In: Proceedings of ACM SIGARCH Computer Architecture News, 2016. 367–379
9 Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks. In: Proceedings of Advances in Neural Information Processing Systems, 2012. 1097–1105
10 He K, Zhang X, Ren S, et al. Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016. 770–778
11 Su H, Maji S, Kalogerakis E, et al. Multi-view convolutional neural networks for 3D shape recognition. In: Proceedings of IEEE International Conference on Computer Vision, 2015. 945–953
12 Soltani A A, Huang H, Wu J, et al. Synthesizing 3D shapes via modeling multi-view depth maps and silhouettes with deep generative networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017. 1511–1519
13 Qi C R, Su H, Niessner M, et al. Volumetric and multi-view cnns for object classification on 3D data. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016. 5648–5656
14 Zhou Y, Tuzel O. Voxelnet: end-to-end learning for point cloud based 3D object detection. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018. 4490–4499
15 Hua B S, Tran M K, Yeung S K. Pointwise convolutional neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018. 984–993
16 Song L, Wang Y, Han Y, et al. C-brain: a deep learning accelerator that tames the diversity of CNNs through adaptive data-level parallelization. In: Proceedings of Design Automation Conference (DAC), 2016. 1–6
17 Wu Z, Song S, Khosla A, et al. 3D shapenets: a deep representation for volumetric shapes. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015. 1912–1920
18 Armeni I, Sener O, Zamir A R, et al. 3D semantic parsing of large-scale indoor spaces. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016. 1534–1543
19 Muralimanohar N, Balasubramonian R, Jouppi N P. CACTI 6.0: a tool to model large caches. HP Laboratories, 2009, 22–31