2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA) | 978-1-6654-3333-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCA52012.2021.00011

Sparsity-Aware and Re-configurable NPU Architecture for Samsung Flagship Mobile SoC

Industrial Product

Jun-Woo Jang*, Sehwan Lee*, Dongyoung Kim*, Hyunsun Park*, Ali Shafiee Ardestani†, Yeongjae Choi*, Channoh Kim*, Yoojin Kim*, Hyeongseok Yu*, Hamzah Abdel-Aziz†, Jun-Seok Park‡, Heonsoo Lee‡, Dongwoo Lee‡, Myeong Woo Kim*, Hanwoong Jung*, Heewoo Nam*, Dongguen Lim*, Seungwon Lee*, Joon-Ho Song*, Suknam Kwon‡, Joseph Hassoun†, SukHwan Lim‡, and Changkyu Choi*

*Samsung Advanced Institute of Technology, Suwon, Korea
†Samsung Semiconductor, Inc., San Jose, USA
‡Samsung Electronics, Hwaseong, Korea

{*junwoo.jang, sehwan.b.lee, dong-y.kim, h-s.park, ali.shafiee, yj90.choi, channoh.kim, yoojin73.kim, hs617.yu, hamzah.a, js.june.park, heonsoo.lee, dw88.lee, k.myeong-woo, hw7884.jung, hee-woo.nam, dongguen.lim, seungw.lee, joonho71.song, suknam.kwon, j.hassoun, sh19.lim, and changkyu_choi*}*@samsung.com*

***Abstract*—Of late, deep neural networks have become ubiquitous in mobile applications. As mobile devices generally require immediate response while maintaining user privacy, the demand for on-device machine learning technology is on the increase. Nevertheless, mobile devices suffer from restricted hardware resources, whereas deep neural networks involve considerable computation and communication. Therefore, the implementation of a neural-network specialized hardware accelerator, generally called neural processing unit (NPU), has started to gain attention for the mobile application processor (AP). However, NPUs for commercial mobile AP face two challenges that are difficult to realize simultaneously: execution of a wide range of applications and efficient performance.**

In this paper, we propose a flexible but efficient NPU architecture for a Samsung flagship mobile system-on-chip (SoC). To implement an efficient NPU, we design an energy-efficient inner-product engine that utilizes the input feature map sparsity. We propose a re-configurable MAC array to enhance the flexibility of the proposed NPU, dynamic internal memory port assignment to maximize on-chip memory bandwidth utilization, and efficient architecture to support mixed-precision arithmetic. We implement the proposed NPU using the Samsung 5nm library. Our silicon measurement experiments demonstrate that the proposed NPU achieves 290.7 FPS and 13.6 TOPS/W, when executing an 8-bit quantized Inception-v3 model [1] with a single NPU core. In addition, we analyze the proposed zero-skipping architecture in detail. Finally, we present the findings and lessons learned when implementing the commercial mobile NPU and interesting avenues for future work.

***Index Terms*—neural processing unit, neural network, accelerator, sparsity, mixed-precision, re-configurable**

I. Introduction

With the success of deep-learning, applications relying on deep neural networks, such as speech recognition (e.g., Bixby), vision task (e.g., Bixby vision), and personalization, have become popular in mobile devices. As mobile applications generally require real-time response, the always-on feature, and privacy preservation, there has been an explosion in the demand for on-device machine learning technology. However, mobile devices suffer from restricted power budget, memory bandwidth, and computing resources, whereas deep-learning applications necessitate substantial computations and memory communication to execute deep neural networks. Thus, it is challenging to execute deep neural networks efficiently using general purpose mobile processors, which are not optimized for neural networks. Moreover, this problem worsens as the neural networks become deeper to achieve higher accuracy, which is the trend in recent deep learning research [2, 3, 4, 5].

Recently, a dedicated hardware accelerator for neural networks called neural processing unit (NPU), has been extensively studied for the efficient execution of neural networks [6, 7, 8, 9, 10]. For neural network acceleration, neural network characteristics, such as the dataflow [7, 11], sparsity [8, 12], and quantization [13, 14], are mainly utilized. Quantization methods [15] reduce the data bit-width and enable the hardware accelerator to accommodate more computational as well as memory resources within the same chip area. Utilizing the sparsity, the NPU can improve the performance and/or energy efficiency by skipping ineffectual computations caused by zero values [12]. In addition, the application of pruning techniques [16], which increase the portion of zero weights by sacrificing the acceptable accuracy, can maximize sparsity gain. By sacrificing the acceptable accuracy, the NPU can accelerate applications requiring immediate response but are less sensitive to the accuracy, such as camera filters for real-time video communication.

Unfortunately, a commercial mobile application processor (AP) is required to execute a comprehensive range of neural networks while satisfying various user requirements. For instance, two different types of applications might exist in a

This paper is part of the Industry Track of ISCA 2021's program.

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00011

mobile device, where the one is *accuracy-sensitive* and the other is *throughput-sensitive*. By adopting low-precision arithmetic units alone, throughput-sensitive applications in which the throughput requirement is stringent but minor accuracy loss is acceptable can be optimized. However, the performance and quality of an accelerator with only low-precision arithmetic units might degrade when executing general applications, and at the worst, an accuracy-sensitive application might fail to meet the constraint. In addition, as mobile applications are provided by various developers, the application of an optimization method might be restricted because it requires extra effort during the design stage (e.g., pruning). Thus, it is critical to achieve both *flexibility* and *efficiency* in the NPU architecture for a commercial mobile AP.

In this paper, we propose an NPU that aims to achieve both efficiency and flexibility for a commercial mobile AP. The architecture of the proposed NPU is scalable, with three cores including 2k 8-bit MACs each. The novel architecture and methodologies used to achieve flexibility while executing the neural network efficiently in the proposed NPU are summarized below.

Efforts to realize efficiency:

- **Design of an energy-efficient basic computation unit:** Based on our case study, which compares the MAC unit and inner-product engine under the Samsung 5nm library, we design an energy-efficient inner-product engine and utilize it as the basic computation unit.
- **Utilization of the input feature map sparsity.** The proposed sparsity unit takes advantages of the natural sparsity of the input feature map induced by nonlinear functions returning zero values (e.g., ReLU). Based on our observation of a channel-wise pattern, we apply an efficient sparsity unit for the inner-product engine.

Efforts to achieve flexibility:

- **Re-configurable MAC array.** We apply a re-configurable MAC array that can change the input and output channel sizes of the inner-product engine. The proposed NPU maintains high utilization in various types of layers by suitably re-configuring the data path of the MAC array.
- **Dynamic port assignment for complete utilization of the on-chip memory bandwidth.** As the required bandwidth for the input feature map, weight, and partial sum vary among the layers, we apply dynamic port assignment to completely utilize the on-chip memory bandwidth. For instance, to accelerate MobileNet-v2, where the input feature map bandwidth is a performance bottleneck, the proposed dynamic port is assigned to the input feature map for relieving the bandwidth bottleneck.
- **Efficient mixed-precision support:** The proposed NPU adopts an optimized 8-bit integer unit for energy efficiency; however, it can support 16-bit integer arithmetic by consuming multiple cycles without an additional accumulator.

In addition, we present the lessons learned and promising avenues for future work based on our considerable experience in designing NPUs for a commercial mobile AP.

II. Background

A. Energy-efficient NPU

Several approaches have been proposed for developing energy efficient NPUs targeting the mobile environment. Eyeriss [7] classified the conventional dataflows in DNN processing based on the type of reused data, which are stationary in the computing block for multiple cycles. By analyzing dataflows, row-stationary dataflow as well as dedicated accelerator architecture were suggested to improve the energy-efficiency of DNN execution.

Quantization can be effectively exploited in NPU architecture. As the quantization sensitivity varies in each network and even in each layer, the optimal bit-widths after quantization differ for each layer. To fully utilize advantage of quantization, various NPUs supporting mixed-precision arithmetic have been proposed. Stripes [17] used bitwise computing temporally through a series of accumulations and bit-shift operations. On the other hand, Bit fusion [18] proposed that a set of small bit-width processing units which construct an arithmetic unit for larger bit-widths spatially.

Another method to utilize value characteristics during DNN computation involves the skipping of ineffective operations such as multiplications using either zero input feature map or weights. As the performance improvements through zero-skipping are dependent on the proportion of zero data, several pruning methods have been proposed to increase the number of zero weights [19, 20]. Such weight pruning methods necessitate retraining to complement the accuracy reduction. However, an NPU for a commercial SoC may be used by third parties with in-house neural network models, rendering the adoption of weight pruning in general NPU architecture inappropriate. In terms of the feature map sparsity, ReLU, which is the most popular nonlinear activation function, generates numerous zero input feature maps during processing, and a mobile SoC can exploit the input feature map sparsity to improve the efficiency with adequate hardware support [8, 21].

Although the energy-efficiency of the NPU can be enhanced by exploiting the data sparsity, it complicates DNN execution by replacing the fixed processing order by variable patterns that can be changed depending on the zero-data distribution. For example, the application of zero-skipping for both input feature maps and weights would result in the under-utilization of MACs due to the complex control, and render the NPU performance more sensitive to the network characteristics [21]. Therefore, the tradeoff relationship between the data sparsity and energy-efficiency should be explored in commercial mobile NPUs.

B. Architecture requirements for DNN processing

DNN-based models have achieved remarkable performances in various fields. These models have diverse shapes and model parameters depending on their application and target performance. Crucial model parameters such as the layer

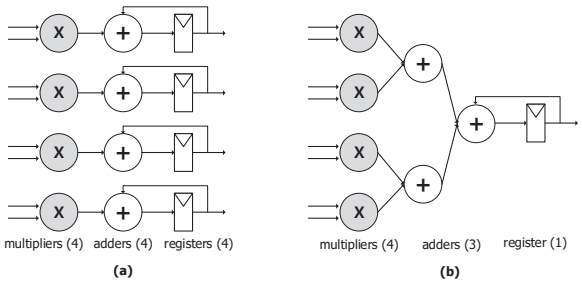


Fig. 1: (a) MAC-based computation engine and (b) adder-tree-based inner-product engine.

depth, number of channels, feature/weight data volumes, and connection patterns between layers could differ among the layers. ResNet [22] introduced residual paths to alleviate the vanishing-gradient problem that occurred during back-propagation in previous models. MobileNet [23] suggested a depthwise-separable convolution technique to effectively reduce model size and operation count. Despite the diversity among models, the principal operation in DNN processing is the inner-product. In addition to the feed-forward network that has gained importance of late as a transformer model, which dominates in several language applications, convolution, which is crucial in image processing applications, can be considered as a type of inner-product with a data reusing scheme. In this context, to support various DNN models in an energy-efficient manner, the NPU architecture should be optimized to perform inner-product-based operations.

Several studies, including [15, 24], have verified that the exploitation of fixed-point integer computing instead of the original floating-point values can achieve similar accuracy in traditional models for image classification, such as AlexNet [25], VGG [26], and MobileNet [23]. On the other hand, several DNN models in more complex fields have exhibited considerable accuracy degradation when using an 8-bit integer system (INT8). Hence, our mobile NPU aims to realize both efficiency and flexibility.

III. Motivations and Design Strategies

In this section, we present our motivations and design strategies for the proposed architecture.

A. Growing Overhead of the Clocking Elements

To accelerate a neural network without sacrificing the quality, various approaches, such as sparsity [19, 27], dynamic voltage and frequency scaling (DVFS) [28], and dynamic quantization [29, 30], have been proposed. These approaches optimize neural-network execution during runtime by providing dynamic control at the cost of additional overhead, particularly the clocking elements. The cost of a clocking element (e.g., flip-flop) increases as the transistor-size reduces. The performance or energy improvement realized through additional features, accompanied by excessive control, may be diluted due to the overhead of the clocking elements. Thus, we focus on achieving considerable performance improvement while minimizing the control overhead. To realize this objective, we propose sparsity-aware architecture, and analyze the tradeoff between the hardware overhead and performance of the proposed sparsity-aware architecture variants. The details are presented in Sections V-A and VII-A.

TABLE I: Energy consumption of the MAC-based computation engine and the adder-tree-based inner-product engine. Each engine has 16 multipliers with 8-bit input data

Component	MAC	Adder-tree
Register (32-bit)	39.2mJ	4.9mJ
Multiplier (8-bit)	71.3mJ	71.3mJ
Adder (16-bit)	52.5mJ	17.5mJ
Total	163.0mJ	93.7mJ

With the increase in the clocking element overhead, the demand for reducing the number of accumulators increases as well. Figure 1 illustrates the adder-tree-based inner-product engine and the MAC-based computation engine. Each multiplier of the MAC engine has its own accumulator that includes flip-flops (Figure 1a), whereas the multipliers in the inner-product engine share an accumulator (Figure 1b). Although the independent accumulators assigned to each multiplier render the MAC engine more flexible, additional area and power overhead compared to the inner-product engine are caused. We implement an efficient inner-product engine as our basic computation unit and propose additional features to achieve flexibility with the inner-product engine. Table I compares the energy consumption between the proposed inner-product engine and the MAC engine, where each engine has the same number of multipliers (e.g., 16). Both are implemented using the Samsung 5nm library. As shown in the table, the energy consumption of the proposed inner-product engine is reduced by 42.5%, compared to the MAC-based engine.

B. Low Resource Utilization

As NPUs for mobile devices execute a wide range of applications, the various layers are required to run efficiently. When executing a neural network using an inner-product engine, each layer is divided into 1D tensors, which are assigned as input to the computation unit. Computational resource utilization depends on a combination of the hardware configuration (e.g., adder-tree size) and tensor shape. Thus, the strategy for mapping the tensors to the computation units is the key for realizing high resource utilization. However, a plain inner-product engine may suffer from low utilization due to lack of flexibility (e.g., static adder-tree size) when executing various tensor shapes. To maximize the resource utilization of the proposed architecture under various tensors, we propose a reconfigurable MAC array, which enables the dynamic changing of the tensor mapping strategy during runtime. Thereby, a compromise between the efficiency and flexibility is possible in the proposed architecture, as described in Section V-B.

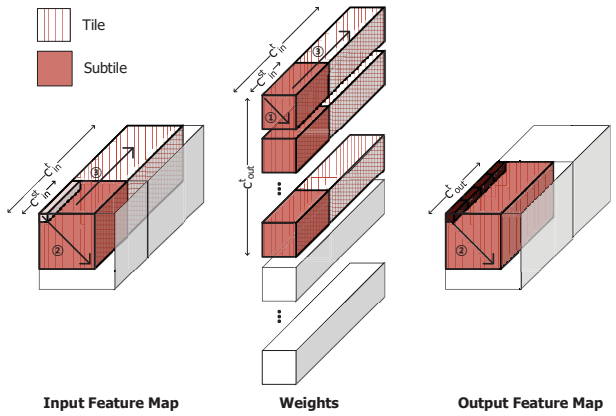


Fig. 2: Work distribution model of the convolutional layer.

Even if we find an excellent strategy for mapping the tensors to the computation unit, the NPU might suffer from utilization degradation if the bandwidth between the computation unit and memory is insufficient. When executing neural networks, the computation unit requires (or generates) different types of data, such as the input feature map, weight, and output feature map, and communicates with the memory. In case each data type has a designated data fetcher and/or on-chip memory, the maximum bandwidth for each data is fixed during design, where their sum is the total on-chip bandwidth. In this case, even if certain data traffic are dominant and the ports for the other data types are idle, each data transfer only occurs through the designated port. As a result, the overall memory bandwidth is wasted. To completely utilize the overall memory bandwidth during runtime, we propose a shared on-chip memory and dynamic data fetcher, which can fetch arbitrary data types, as detailed in Section V-C.

C. Mixed-Precision Requirement

Commercial mobile devices are required to run a comprehensive range of applications while satisfying various requirements. However, it is difficult to simultaneously satisfy some of the representative requirements such as high-throughput and high-accuracy. For instance, although quantization is promising for achieving high throughput using restricted resources, it is often accompanied by accuracy reduction. Thus, there is a growing demand for supporting mixed-precision arithmetic to satisfy the various requirements. Based on the observation that 8-bit integer arithmetic is sufficient for most mobile applications, we optimize the proposed NPU for 8-bit integer arithmetic. In addition, to cope with applications requiring high accuracy, the proposed NPU supports 16-bit integer arithmetic using multiple cycles without an additional accumulator, as described in Section V-D.

IV. Architecture

A. Work Distribution Model

We describe our work distribution model using an example, where a single NPU engine computes the convolutional layer (Figure 2). The proposed NPU computes an entire layer by iteratively performing partial computations. The overall execution flow of the proposed NPU is summarized as follows:

1) Given a convolutional layer, we first divide the output feature map along a channel ($\mathbf{c}^t_{out}$ in Figure 2) and the spatial dimension to generate a partial computation unit, which we call *tile*. Each tile can be assigned to a single NPU engine or a set of NPU engines.
2) Similarly, the computation in a tile is divided into multiple subsets called *subtiles*, where the computation result of each subtile is the partial sum of the same output feature map. Each subtile includes chunks of the input feature map and weights divided along the input channel dimension ($\mathbf{c}^{st}_{in}$ in Figure 2)
3) The inputs assigned to each subtile are further divided into 1D tensors. To process a subtile, the NPU engine generates a partial sum tensor (e.g., the partial output feature map tensor in Figure 2 sized $1 \times 1 \times \mathbf{c}^t_{out}$), using an input feature map tensor (e.g., the input feature map tensor in Figure 2 sized $1 \times 1 \times \mathbf{c}^{st}_{in}$) and the $\mathbf{c}^t_{out}$ weights tensors (e.g., the weight tensors in Figure 2 sized $1 \times 1 \times \mathbf{c}^{st}_{in}$). The above process is repeated until all the weight tensors are computed (① in Figure 2), and the results are accumulated to generate the final partial sum tensor. Architectural parameters $\mathbf{c}^{st}_{in}$ and $\mathbf{c}^t_{out}$ are multiples of 16 and 64, respectively.
4) Step 3 is repeated until all the partial sum tensors are computed (② in Figure 2) to complete a subtile.
5) Steps 2–4 are repeated until all the subtiles are computed (③ in Figure 2) to complete a tile.
6) The proposed NPU iterates steps 2–5 until all the tiles in the layer are completed. The generated output feature map is then used as the input feature map for the next layer.

B. Overall Architecture and Computation Process

The proposed NPU includes three cores. Figure 3 illustrates the top-level core design. The core includes two engines called *NPUEs* (e.g., NPUE 0 and NPUE 1); the engines are each equipped with 1k MACs and share an on-chip scratchpad memory. The NPU controller orchestrates data transfer between the on-chip scratchpad memory and an external memory. The NPUE comprises a *data fetcher*, *tensor unit*, and *vector unit*.

As we employ weight-stationary dataflow to maximize the reuse of weights, in order to process each subtile, the *weight fetcher* loads the associated weight tensors and then stores them in a local buffer called *weight buffer*. During each cycle, the *IFM fetcher* loads the associated input feature map tensors from the scratchpad memory. Note that the loaded input feature map tensor is stored as sparse data, which includes zero values. In order to skip the ineffectual computations associated with a zero input feature map, the *sparsity unit* generates a dense tensor, which maximizes nonzero data through zero-skipping. Further, the dense tensors are transferred to four 16×16 MAC arrays (MAAs) for generating a partial sum tensor. The four

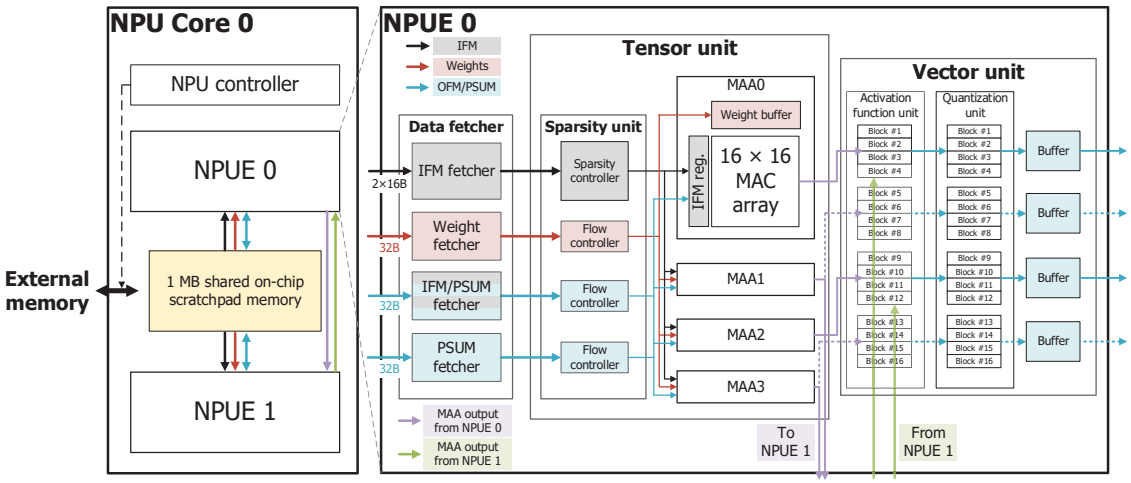


Fig. 3: Top-level architecture of the proposed NPU core comprising two engines (NPUE0 and NPUE1) and an on-chip scratchpad memory. Each engine includes a data fetcher, tensor unit, and vector unit. NPUEs in the same NPU core can communicate directly through the tensor unit output (e.g., to NPUE 1) and vector unit input (e.g., from NPUE 1) to implement a re-configurable MAA.

MAAs receive the same input feature map tensor (e.g., the input feature map tensor in Figure 2 sized $1 \times 1 \times \mathbf{c}_{in}^{st}$), and each MAA generates a partial sum tensor divided along the output channel dimension (e.g., the four small partial output feature map tensors in Figure 2 sized $1 \times 1 \times \mathbf{c}_{out}^{t}/4$). After the accumulation of each partial sum tensor, the result is transmitted to the vector unit, which performs element-wise operations such as nonlinear activation functions.

C. Data Fetcher

The data fetcher includes three static fetchers, the *IFM fetcher*, *weight fetcher*, and *PSUM fetcher*, which handle the input feature maps, weights, and partial sums, respectively. Unlike a static fetcher that handles only a designated type of data, the dynamic fetcher, referred to as the *IFM/PSUM fetcher*, can load an arbitrary type of data from the input feature map or partial sum. All the fetchers have their own buffers, which are sufficiently large to avoid stall due to scratchpad memory delay. Data are loaded through each fetcher in parallel, with a bandwidth of 32 bytes each.

The IFM fetcher is connected to the scratchpad memory through two 16-byte ports. As the proposed NPU utilizes the input feature map sparsity, we set the IFM fetcher bandwidth to twice the data quantity that the MAAs can process per cycle to alleviate utilization drop. To load partial sums with variable bit-width, the PSUM fetcher adopts a time-multiplexing method. We assume that the input data is 8-bit and the corresponding partial sum is 32-bit to cope with the corner case during accumulation. In this case, the IFM and weight fetchers can load 32 elements in a cycle (e.g., 32×1 byte $\times 1$ cycle). To load the corresponding partial sum, which includes 32 4-byte data, the PSUM fetcher loads eight elements per cycle and transfers them to the tensor unit. Two or four cycles are required in total (e.g., 8×4 bytes $\times 2$ fetchers $\times 2$ cycles or 8×4 bytes $\times 1$ fetcher $\times 4$ cycles). The *weight fetcher* preloads the weights of the next subtile during the computations in the current subtile, and the *PSUM fetcher* preloads the partial sums of the next $1 \times 1 \times \mathbf{c}_{out}^{t}$ partial sum tensor during the computations in the current partial sum tensor in order to overcome the load latency of the weight and partial sum through double buffering.

D. Tensor Unit

Sparsity unit. To find the nonzero input feature map, and its corresponding weight and partial sum, the *sparsity controller* first generates nonzero bit flags, which store information on whether each input feature map is zero. Based on our priority-based searching mechanism, the sparsity unit then generates appropriate dense data and transfers them to the MAAs. As the input feature map is shared among the MAAs in the same tensor unit (e.g., MAA0 to MAA3 in Figure 3), the sparsity unit broadcasts the nonzero input feature map to the MAAs, reducing data traffic. Note that the proposed NPU bypasses the sparsity unit when executing the pooling layer. The proposed zero-skipping mechanism is described in Section V-A.

MAC array (MAA). Figure 4 illustrates the architecture of the proposed MAA. The tensor unit includes four MAAs, each comprising 16 MUL columns. Each MUL column has 16 multipliers, and two sets including an adder-tree and accumulator each (i.e., four 16x16 MAAs in total). In order to achieve high throughput by reducing the required bandwidth from the sparsity unit, the input feature map is shared across the MUL columns (e.g., the arrow annotated as IFM sharing lane in Figure 4). A set of horizontal MACs, which share an input feature map, is defined as a *lane*. During each cycle, each MAA loads the same input feature map tensor (e.g., $\mathbf{i} \in \mathbb{R}^{1\times1\times16}$)

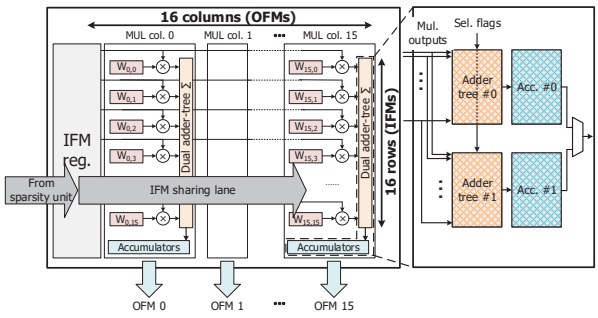


Fig. 4: Architecture of the proposed MAA. The MAA includes 16 MUL columns comprising 16 multipliers each, and two sets including an adder-tree and accumulator each (denoted as *dual adder-tree* and *accumulator*). The MUL columns share the same input feature map.

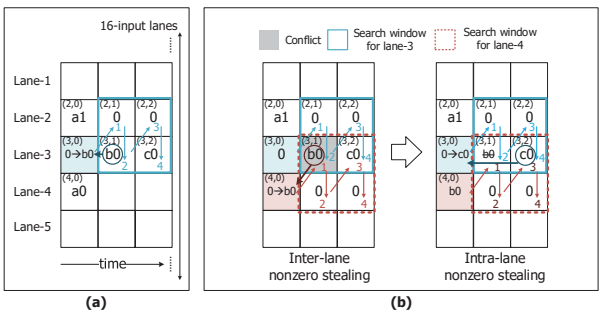


Fig. 5: Priority-based search algorithm for hardware friendly zero-skipping: (a) Mechanism of the priority-based search algorithm when there is no conflict. Lane-3 loads b0 as the next nonzero input. (b) Conflict resolution of the priority-based search algorithm since the values of lane-3 and lane-4 are zero. Assuming that conflict occurs in b0, lane-4 loads b0 as the next nonzero input because it precedes lane-3 in terms of the priority (e.g., the priority of lane-4 is one and that of lane-3 is two). Lane-3 loads c0 as the next nonzero input.

as well as 16 weight tensors (e.g., $\mathbf{w} \in \mathbb{R}^{(1\times1\times16)\times16}$), and performs the inner-product to generate a partial sum tensor (e.g., $\mathbf{i} \cdot \mathbf{w} = \mathbf{p} \in \mathbb{R}^{1\times1\times16}$).

The proposed MAA is re-configurable in order to achieve better utilization and flexibility. Zero-skipping architectures often suffer from load imbalance due to the differences in the number of effectual computations among the computation units, reducing the utilization. To alleviate the load imbalance problem among lanes, a single MUL column equips two 16-input adder-trees and accumulators in the proposed architecture; each accumulates the partial sum for two different output pixels (e.g., Adder-tree #0-#1 and Acc. #0-#1 in Figure 4). In addition, the proposed architecture supports mixed-precision efficiently. Using the MAA optimized for 8-bit arithmetic, the proposed MAA can perform 16-bit operation by consuming multiple cycles without an additional accumulator, as detailed in Section V.

E. Vector Unit

The *activation function unit* receives the output tensors from the tensor unit and performs vector-wise operations, such as element-wise summation between the outputs from different NPUEs. To compute the nonlinear activation functions, the activation function unit also performs basic linear-algebra for piece-wise linear approximation, $aX + b$, where X is the output tensor from the tensor unit, and a and b are scalar values. The proposed NPU can support activation functions ranging from ReLU variants to the sigmoid using the auxiliary arithmetic functions implemented in the vector unit. Each unit in the activation function block (e.g., Block #1 to #16 in Figure 3) is assigned to each output pixel and process the computation in a cycle. Thus, to compute an output feature map in which the size is greater than the number of blocks, the activation function unit operates in a time-multiplexed manner. For instance, 16 blocks in the activation function unit can process 16 output pixels during a cycle, and four cycles are consumed when an output feature map sized $1 \times 1 \times 64$ is received.

As the proposed activation function unit generally receives a 1D output feature map tensor divided along the channel dimension, channel-wise quantization [31] can be applied without reshaping the tensors (e.g., *Quantization unit* in Figure 3). To reduce energy consumption, the proposed NPU bypasses the activation function unit and quantization unit when they receive the partial sum of the output feature map. The vector unit buffers store either the partial sums or final results, and transfer data to the scratchpad memory using double-buffering.

V. Key Features

A. Sparsity Utilization

We employ two methods, *intra-lane search* and *inter-lane search* to skip the zero values in the input feature map. Intra-lane search finds the nonzero values assigned to the following few cycles. Applying intra-lane search, each lane can perform effectual computations until all the assigned nonzero feature maps are consumed. However, as the MUL columns in the MAA generate partial sums for the same output feature map, the MAA proceeds to the next accumulation only after all the lanes in the MAA consume their inputs. Thus, the performance gain achieved by the zero-skipping method relying only on intra-lane search varies based on the number of nonzero values in each lane, and the total execution time of an MAA is determined by the straggler lane. To mitigate this problem, we apply inter-lane search additionally, which expands the search range and steals the nonzero input feature map from the neighboring lane.

The proposed inter-lane search is inspired by the zero-weight skipping proposed in [32]. Unlike skipping zero

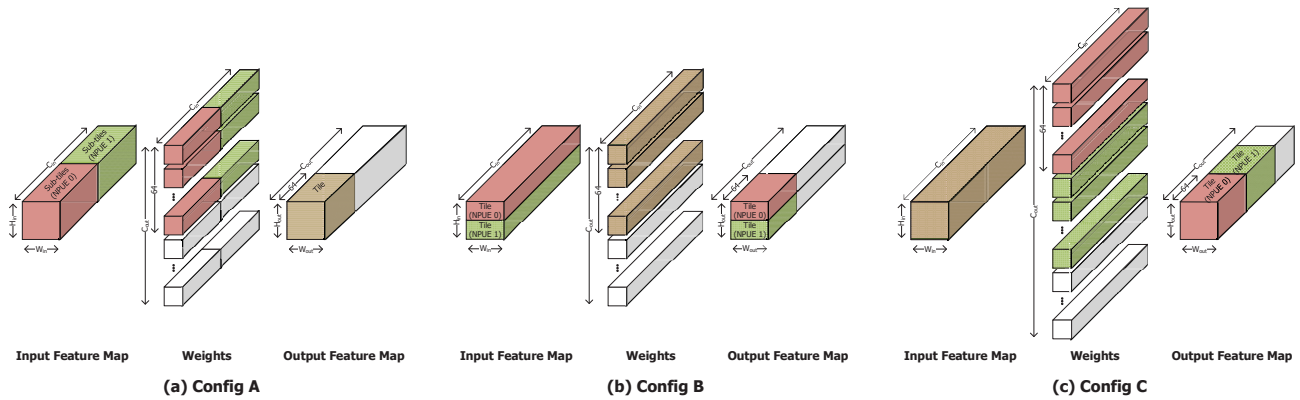


Fig. 6: Three configurations of the two NPUEs through which a tile or subtile is assigned: (a) config A: Two groups of subtiles in the same tile, (b) config B: two independent tiles in the spatial dimension, and (c) config C: two independent tiles in the channel dimension. The red, green, and brown tensors are assigned to NPUE 0, NPUE 1, and are common to both NPUEs, respectively.

weights [32], we aim to utilize the input feature map sparsity because of the commercial mobile-device environment. Skipping zeros in weights is beneficial when accompanied by pruning; however, many models running on commercial mobile devices are provided by third-party developers without pruning. Thus, to take advantage of the sparsity in practical usage, the proposed methods utilize the natural sparsity in the input feature map generated by nonlinear functions, such as ReLU, which return zero for any negative input value.

As the distribution pattern of a nonzero feature map is unpredictable before execution, the sparsity unit has to search for nonzero values during runtime. We implement a specific hardware unit, which finds the nonzero values of each lane within the search window. Unfortunately, there are overlaps among the search windows when performing inter-lane search, rendering hardware-based searching complicated. To tackle this problem, we propose a *priority-based search algorithm*, which enables the skipping of a zero input feature map in a hardware-friendly manner.

We describe the proposed priority-based search algorithm using an example in which the intra- as well as inter-lane search window size is two (Figure 5). Figure 5a illustrates the working of the proposed priority-based search algorithm when there is no conflict. Each lane has its own search window for which the size is *intra-lane search window size* $\times$ *inter-lane search window size* (e.g., 2×2 blue box in Figure 5a). If the next value for lane-3 is zero (the zero value at coordinate **(3,0)** in Figure 5a), the search order is as follows: coordinate **(2,1)**, **(3,1)**, **(2,2)** and **(3,2)**. More precisely, assume that the time step is **t** and lane-**x** is given. If the value at position $(\mathbf{x}, \mathbf{t})$ is zero, where the coordinate represents (*lane number*, *time step*), the search order of the proposed method is **(x-1, t+1)**, **(x, t+1)**, **(x-1, t+2)** and **(x, t+2)**. Following the search order, the sparsity controller finds the first nonzero data for each lane and utilizes it as the next data (e.g., lane-3 utilizes b0 as the next data instead of zero as shown in Figure 5a).

Figure 5b illustrates the working of the proposed priority-based search algorithm when conflict occurs. Assume that the next values of lane-3 and lane-4 are zero (the zero values at coordinate **(3,0)** and **(4,0)** in Figure 5b). When searching for nonzero data, conflict occurs because the first nonzero data in lane-3 as well as lane-4 is b0 (denoted by the gray box in Figure 5b). In case of conflict, the lane with higher priority (i.e., the lane preceding in the searching order) owns the conflicted value. b0 is the second value in the search window of lane-3 but the first value in that of lane-4. Thus, lane-4 steals the nonzero b0, which had been assigned to its neighbor (i.e., inter-lane search). Lane-3 owns the nonzero c0, which had been assigned to the next cycle, instead of b0 (i.e., intra-lane search).

Inter-lane search requires extra MUX circuits to handle the overlap between search windows and the overhead increases with the increase in the window size of inter-lane search. We optimized the performance and cost of the proposed sparsity unit based on our experimental results. Detailed analysis of the performance and cost under various intra- and inter-lane window sizes is presented in Section VII-A.

B. Re-configurable MAC Arrays

To further improve the MAC utilization of the proposed inner-product engine, we propose re-configurable MAC arrays. By dynamically controlling the connection between a tensor unit and vector unit, we implement re-configurable MAC arrays. During execution, the proposed NPU can connect the tensor and vector units within the same NPUE or across different NPUEs through inter-engine connections (e.g., denoted as “To NPUE1” and “From NPUE1” in Figure 3). As one subtile is assigned to an NPUE at a time, we implement three operating mode configurations by assigning a tile (or subtile) to multiple NPUEs in different ways. Figure 6 depicts examples of the operation of a re-configurable MAC array,

where three different methods of assigning a tile (or subtile) to two NPUEs are illustrated.

Config A. Different NPUEs are responsible for processing different subtiles in the same tile, and the result of each is the partial sum of the same output (Figure 6a). Assuming that NPUE 0 accumulates the output feature maps, the partial sums generated from the subtiles assigned to NPUE 1 are loaded to NPUE 0 through a wire connected to NPUE 1 (e.g., denoted as "From NPUE 1" in Figure 3), after which all the partial sums are summed at the vector unit. As two 16-input adder-trees in different NPUEs run simultaneously operating like a 32-input adder-tree, MAC utilization might degrade if the number of input channels is not a multiple of 32 in this mode.

Config B. Different NPUEs are responsible for processing different tiles, which are divided along the spatial dimension of the input and output feature maps (Figure 6b). Each NPUE holds the same weights and generates an output feature map divided along the spatial dimension. Thus, the output from the tensor unit of each NPUE is transferred to their own vector unit and processed in parallel. As each NPUE generates an output feature map divided along the spatial dimension in parallel, MAC utilization might degrade if either the width or height is not a multiple of two (i.e., the number of NPUEs in each core).

Config C. Different NPUEs are responsible for processing different tiles along the output channel dimension (Figure 6c). Each NPUE performs computations for different weights using the same input feature map; hence, each NPUE performs its own execution in parallel without any communication. As each NPUE generates an output feature map divided along the output channel dimension and each NPUE generates $1 \times 1 \times 64$ output feature maps at a time, MAC utilization might degrade if the number of output channels is not a multiple of 128 (i.e., 2×64).

During compilation, the proposed NPU selects the operating mode for each layer among the three configurations described above. This enables the proposed NPU to achieve higher MAC utilization.

C. Dynamic Port Assignment

The proposed NPU has designated data fetchers for each data type (e.g., input, weight, partial sum, and output) and a shared on-chip memory on which they are stored. As certain data ports may be busy when the ports for the other data types are idle, naively assigning an on-chip memory port to each data type may result in inefficiency. To mitigate such a problem and completely utilize the on-chip memory bandwidth, we propose dynamic port assignment, which redirects data suffering from lack of bandwidth to an idle port. In particular, we focus on partial sums, which often require low bandwidth during runtime. A partial sum requires low bandwidth mainly because of the following: (i) it is only loaded when multiple subtiles exist in a tile and (ii) it is written back to the memory only after accumulation is complete. Thus, we implement dynamic port assignment using the IFM/PSUM fetcher, which can load

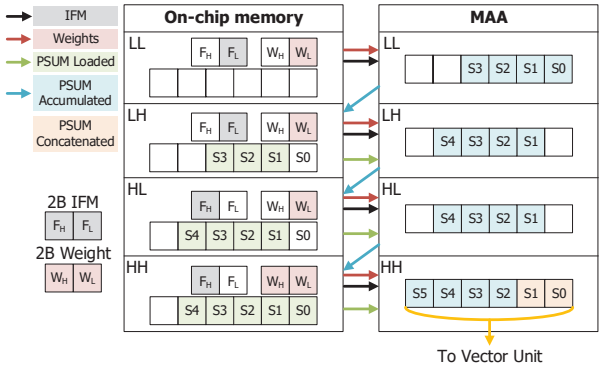


Fig. 7: Simplified operation diagram of the computation of 16-bit fixed-point arithmetic by the proposed architecture using time-multiplexing without additional computational resources in a data path originally designed for 8-bit fixed-point arithmetic.

an arbitrary data type among the input feature maps and partial sum (shown in Figure 3).

The depth-wise convolutional layer, widely utilized in mobile applications, is a representative use case that benefits from the proposed dynamic port assignment. In depth-wise convolution, the input feature map is reused only along the spatial dimension unlike conventional convolution where the input activation is reused among the kernel weights (i.e., channel dimension of the output feature map). Hence, to execute depth-wise convolution, each MAA requires distinct input feature maps, unlike conventional convolution where the input activation is broadcast to every MAA in the NPUE. Even though four 16×16 MAAs require distinct $1 \times 1 \times 16$ input tensors to process depth-wise convolution, the IFM fetcher of the proposed NPU loads only two $1 \times 1 \times 16$ input tensors in a cycle. Thus, without dynamic port assignment, MAA utilization degrades due to lack of input feature map bandwidth. However, the proposed IFM/PSUM fetcher enables the loading of two extra input feature map tensors in a cycle, thereby fully utilizing the four MAAs.

D. Mixed-precision

Inference using only 8-bit integer inputs (i.e., INT8 format) improves the latency and energy consumption by sacrificing the acceptable accuracy (e.g., 1∼2%) in well-known image classification models [24]. However, to cope with the requirement of precise results without compromising on the accuracy, the proposed NPU also supports inference using 16-bit integer inputs (i.e., INT16 format). To handle INT16 input data with minimum overhead, we propose a time-multiplexing method to support INT16 arithmetic using a data path originally designed for INT8 input data. By including suitable muxing operations in the data fetcher and vector unit, the proposed time-multiplexing approach can handle INT16 input data without the addition of a multiplier or accumulators.

Figure 7 illustrates an example of the proposed NPU operation in a time-multiplexing manner using the INT8 data path when both the input feature map and weight are in INT16 format. To perform INT16 arithmetic using the proposed time-multiplexing method, 2-byte input data are decomposed into two 1-byte data (e.g., F_H, F_L, W_H and W_L) and multiple subcomputations (e.g., *LL*, *LH*, *HL* and *HH*) are performed. Note that each rectangle represents a 1-byte space. At each subcomputation, the data fetcher loads the appropriate 1-byte data and feeds them into the tensor unit to perform INT8 arithmetic. The tensor unit generates a new partial sum and accumulates it to the most significant 3-byte or 4-byte data of the previous partial sum. The above process is repeated until each subcomputation is complete.

For example, four iterations are required when the input feature map and weight are in INT16 format. In the first iteration, the lower 1-byte data of the input feature map and weight, F_L and W_L, are loaded to the tensor unit and INT8 multiplication is performed (*LL* in Figure 7). The last 4-byte data of the partial sum, S[3:0], is valid and stored in the on-chip memory. In the second iteration (*LH* in Figure 7), F_L and W_H are loaded together to the tensor unit with an appropriate previous partial sum, i.e., S[3:1]. The multiplication result is accumulated to the previous partial sum and a new 4-byte partial sum result, S[4:1], is generated and stored in the on-chip memory. The third iteration is performed similarly. In the last iteration, F_H and W_H are loaded, and the last multiplication and accumulation are performed. As the last iteration generates the final output, every valid byte of the previous partial sum result (i.e., S[4:0]) is loaded. Even though 5-byte data of the previous partial sum are loaded, the 32-bit accumulator is sufficient for computation because only 3-byte data (i.e., S[4:2]) is accumulated with the newly computed result (i.e., S[5:2]). Finally, the last 2-byte data of the previous partial sum, S[1:0], are concatenated with the accumulator output and transferred to the vector unit.

VI. Experimental Setup

Neural network models. We developed RTL implementations as well as an in-house cycle-accurate simulator and verified the advantages of the proposed NPU using three DNNs: Inception-v3 [1], ResNet-50 [22], and MobileNet-v2 [33]. In addition, we measured the end-to-end performance on a silicon product using three more DNNs: MobileNet [23], U-Net [34], and DeepLab-v3 [35]. We trained the models on the ImageNet [36] dataset and randomly selected 32 images as the input for measurement. We applied a channel-wise quantization method [37], in which each feature map, weight, and bias were 8-bit integers. No pruning techniques were applied.

Simulation. We built an in-house cycle-accurate simulator, implemented in C++, to model the performance of the proposed NPU. The simulator models a single NPU core, which comprises two NPUEs and an on-chip scratchpad memory. The details of the architectural parameters used in our simulation are listed in Table II. We utilized the simulator for the architectural exploration of various sparsity methods. The baseline is the proposed NPU, with a fixed config A (described in Section V-B) and without feature map sparsity (described in Section V-A) and dynamic port assignment (described in Section V-C).

TABLE II: Architectural parameters for simulation.

	The proposed NPU
NPU engines	2
IFM fetchers	2
Weight fetchers	2
PSUM fetchers	2
IFM/PSUM fetchers	2
Sparsity units	2
MAC arrays	8
8-bit multipliers	2048
Adder-trees	128
Accumulators	128

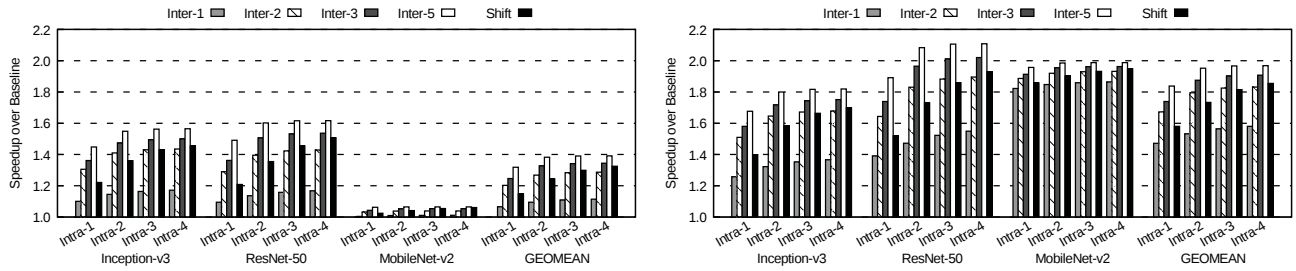
Synthesis and fabrication. We synthesized and fabricated proposed NPU at a 5-nm technology node using the Samsung cell library and Synopsys Design Compiler (O-2018-06-SP4). We measured the area breakdown for each hardware component through the design compiler report.

VII. Evaluation

A. Performance

Speedup on simulator. Figure 8a shows the speedup of the various configurations for feature map zero-skipping compared to the baseline. The MAC array configuration is fixed as config A, which is described in Section V-B. Application of intra-lane zero-skipping along with inter-lane zero-skipping improves the performance significantly by skipping inefficient computation. According to our experimental results on Inception-v3, ResNet-50, and MobileNet-v2, the combined application of intra-lane zero-skipping and inter-lane zero-skipping realizes a geomean speedup of 1.39× with a maximum speedup of 1.62× on ResNet-50, where the window size of intra-lane zero-skipping and inter-lane zero-skipping is four. On applying intra-lane zero-skipping alone with a window size of one, the geomean speedup is 1.07× with a maximum speedup of 1.10× on Inception-v3.

The proposed re-configurable MAC array improves the performance significantly by increasing MAC utilization. Figure 8b demonstrates the benefit of a re-configurable MAC array and dynamic port assignment. On applying the re-configurable MAC array and dynamic port assignment, the proposed NPU achieves a geomean speedup of 1.97× with a maximum speedup of 2.11× on ResNet-50, where the window size of inter- and intra-lane zero-skipping is four. Even if zero-skipping is not applied, a speedup of 1.27× speedup is realized compared to the baseline. MobileNet-v2 , in particular, derives significant advantage from the re-configurable MAC array and dynamic port assignment mainly because dynamic port assignment improves the MAC utilization of depth-wise convolution, which is prevalent in MobileNet-v2. In addition, the re-configurable MAC array increases the



(a) (b)

Fig. 8: Overall speedup for various zero-skipping window sizes (a) without and (b) with a re-configurable MAC array and dynamic port assignment. `Inter-N` and `Intra-N` denote the window sizes of inter- and intra-lane zero-skipping, respectively. `Shift` denotes the feature map shifting algorithm described in Section VIII-A.

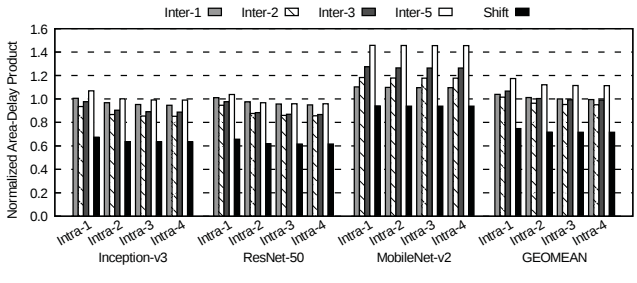


Fig. 9: Normalized area-delay product (ADP) over the baseline.

TABLE III: Performance on a silicon product.

Neural Network Models	Frames per Second (FPS)
Inception-v3 [1]	290.70
MobileNet [23]	1052.26
MobileNet-v2 [33]	622.54
ResNet-50 [22]	131.13
U-Net [34]	41.48
DeepLab-v3 [35]	36.70

utilization of convolutions where the input channel depth is narrow. We discuss the effect of the re-configurable MAC array in Section VII-C.

We observed an imbalance in the number of nonzero values of the feature map among different input channels. Empirically, we found that the application of intra-lane zero-skipping alone resulted in considerable performance improvement if the given input feature map had been shuffled across the channel-wise direction before execution (denoted as `Shift` in Figure 8). The proposed shuffle method achieves a geomean speedup of 1.85× with a maximum speedup of 1.95× on MobileNet-v2. The shuffle method is analyzed in detail in Section VIII-A.

Performance on a silicon product. Table III depicts the end-to-end performance of a single NPU core executing six representative neural network models for mobile devices. We highly optimized the fabricated SoC using various circuit optimization techniques. In addition, we applied specific compiler optimization techniques for our architecture, such as determining the configurations of the re-configurable MAC arrays (Section V-B) and decomposing stages for mixed-precision support (Section V-D) as well as common optimization techniques including tiling and layer fusion [38]. Our experimental results with the silicon product established that the proposed NPU could execute representative neural networks for mobile devices in real-time.

TABLE IV: Area breakdown.

	Ratio[%]
NPU Core	100.0
├ NPUE0	23.4
├ NPUE1	20.3
│ ├ Data fetcher	3.4
│ ├ Tensor unit	68.5
│ ├ Vector unit	17.0
│ └ Etc.	11.1
├ SRAM	50.8
└ Etc.	5.5

	Ratio[%]
Tensor Unit	100.0
├ Sparsity unit	3.51
├ Weight buffer	57.10
├ Multiplier	18.73
├ Adder-tree	7.56
├ Accumulator	7.78
└ Etc.	5.33

B. Area and Energy Efficiency

Figure 9 compares the normalized area-delay product (ADP) with the baseline of the proposed NPU. Area efficiency is critical in implementing an efficient mobile SoC. To measure the ADP, we implemented synthesizable RTL models of the tensor unit with various window sizes and utilized the area reported by the Synopsys design compiler. The proposed NPU with intra- and inter-lane search window sizes of four and two respectively, achieves a minimum geomean ADP of 0.950. When implementing the silicon product, we set the intra- and inter-lane search windows as two in order to minimize the ADP while satisfying the bandwidth limitation of the on-chip memory, realizing a minimum geomean ADP of 0.964.

The proposed NPU was fabricated using Samsung 5nm technology [39]. Figure 10 illustrates the die layout of the proposed NPU, which comprises three NPU cores and a control unit. Each NPU core includes two NPU engines and a 1-MB on-chip scratchpad memory. The NPU core achieves

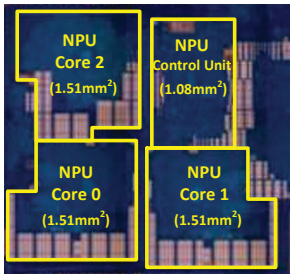


Fig. 10: SoC die layout of the proposed NPU fabricated using Samsung 5nm technology [39].

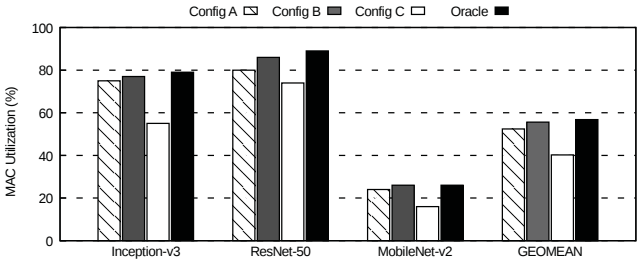


Fig. 11: MAC utilization of each MAC array configuration.

an energy efficiency of 13.6 TOPS/W. Table IV reports the detailed area breakdown of a single NPU core. The tensor unit occupies most of the NPUE area at 68.5%. The data fetcher, which fetches data from the on-chip memory with dynamic port assignment, occupies 3.4% area. The vector unit occupies 17.0% in order to support both diverse operations and the re-configurable MAC array. In particular, our re-configurable tile/subtile scheme demands double the number of processing blocks in the vector unit to handle vector processing of the independent output tensors. The sparsity unit, which is a logic for feature map zero-skipping, occupies 2.4% of an NPUE. In the tensor unit, 57.1% is the weight buffer. Arithmetic circuits, including multipliers and adders, occupy 23.4% of an NPUE.

C. Effect of the Re-configurable MAC Array

Figure 11 shows the MAC array utilization for all the configurations introduced in Section V-B. To demonstrate the advantages of the re-configurable MAC array, we assume that there is no stall due to lack of memory bandwidth. The first three bars in this figure depict the MAC utilization for fixed Config A, B, and C, respectively. In oracle, the optimal MAC array configuration is decided during offline compilation. The re-configurable MAC array achieves a geomean utilization of 56.8% with a maximum utilization of 89% on ResNet-50. In the case of MobileNet-v2, the overall MAC utilization is lower than those of the other models due to depth-wise convolution.

VIII. Analyses and Lessons

In this section, we discuss the learning from the experimental results and our suggestions for the next NPU. Our observations indicate that compromise between the performance and area is necessary for mobile NPUs.

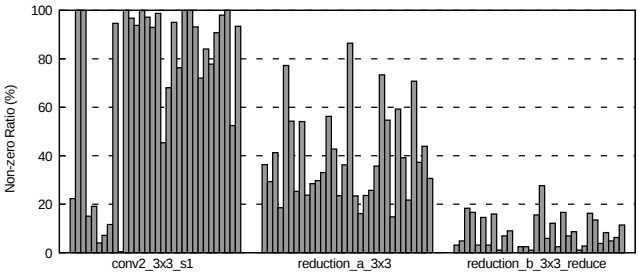


Fig. 12: Nonzero portion of the input feature map, extracted from Inception-v3.

A. Sparsity: Simple Shift Scheme

The proposed NPU achieves significant performance gain by skipping ineffectual computations using the sparsity unit (Section V-A). inter-lane zero-skipping, in particular, drastically enhances the performance by enhancing the computational load balance among lanes. However, as shown in Figure 9, inter-lane zero-skipping is expensive due to the inclusion of several multiplexers to search for nonzeros and select the weight of the other lanes.

We observed the dispersion of the nonzero values, based on the input channels. Figure 12 depicts an example of the feature map distribution extracted from Inception-v3. The nonzero ratio for 32 random channels from three sampled layers of Inception-v3 is depicted. It can be easily observed that the zero ratios among the channels are unpredictably imbalanced.

Therefore, instead of using inter-lane search, we suggest a *shift scheme*, which shuffles the input feature map across the channel dimension during run-time. This shift scheme mitigates the load imbalance problem among lanes to enable the efficient skipping of zero values using only intra-lane search. Even though the shift scheme shows inferior performance compared to the best configuration that applies intra- and inter-lane search, it deserves consideration because it reduces the logic complexity by eliminating multiplexers.

Figure 13 depicts an example of the proposed shift scheme. If a sparsity unit utilizes only intra-lane search, it requires four cycles to consume all the input tensors since lane-4 has four nonzero data. On the other hand, if the shift scheme is applied with intra-lane search, the proposed NPU requires two cycles to consume all the input tensors. We assume that the given kernel size is 2×2. In the proposed architecture, the input feature maps are flattened into a $1\times1\times4$ tensor and assigned to each lane. The amount of shift is determined by the kernel weight coordinates in ascending order from 0–*size of kernel*-1. For example, the shift amount is set as 0–3 in ascending order, in the case of a 2×2 kernel (e.g., Shift-0 to Shift-3 in Figure 13). There is no additional hardware logic to select the corresponding weights because the weight fetcher loads the weights to be aligned with the shifted input feature map.

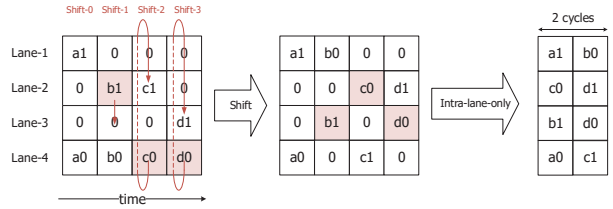


Fig. 13: Shift scheme example (four $1 \times 1 \times 4$ tensors on four lanes). In this example, intra-lane zero-skipping with the shift scheme requires two cycles, whereas four cycles are required without the shift scheme.

Although dynamic detection realizes better performance than static shuffling, as shown in Figure 8, we demonstrate that the area-efficient shift scheme with inter-lane zero-skipping shows the most optimized result in terms of the area-delay product in Figure 9. The increase in the intra-lane window size, which has smaller area overhead than the increase in the inter-lane window size, renders the shift scheme more effective. Its performance is comparable with the zero-skipping method used in our silicon product (2×2 window-size), when the intra-lane search window size is four for all the networks (Figure 8).

B. MAC configuration: Re-configurability vs. Area

There is unnecessary hardware logic, which is not enabled for all the configurations, to enhance MAC utilization with the MAC re-configuration feature described in Section V-B. The adders in the vector unit add the outputs from the two NPUEs only for Config A, and half the data-path in the vector unit for the feature map activation is enabled only for Config B and C. Moreover, the weights are replicated by the controller in the weight buffers of each NPUE for Config B.

As shown in Table IV, most of the area of a tensor unit is occupied by the weight buffer. Thus, we suggest the MAC array configuration to be fixed as Config B with weight buffer sharing between the two NPUEs. There is a tradeoff between MAC utilization and the area overhead mentioned above. However, the performance degradation on sacrificing the re-configurable MAC array is negligible, whereas the area is significantly reduced. The weight buffer sharing is more effective as increasing parallelism in the spatial dimension.

C. Dynamic On-chip Memory Assignment

In practice, the NPU cannot use the full memory bandwidth due to bank conflict, which is difficult to predict when multiple requests for different data type access to the shared on-chip memory occur simultaneously. To address this problem, we suggest an additional small-sized memory that can be dynamically assigned to the input-feature map, partial sums, or weights. The decision method for assignment is similar to the dynamic port assignment described in Section V-C.

In order to maintain MAC utilization, the amount of each data that should be supplied to the MAC differs for each layer because of the variety of layers. For example, a convolution layer with a small width and height feature map has small weight reuse, and the MAC requires larger weight bandwidth than that of a layer with a large width and height feature map. On the other hand, as the layer does not need considerable storage for the partial sum, an additional small-sized memory, which stores the partial sum, enables the main shared on-chip memory to provide a larger weight bandwidth. The performance of a fully-connected layer is generally limited by the memory bandwidth for loading weights from the on-chip memory. Using an additional small-sized memory to store the feature map allows the main memory to provide a large bandwidth for loading weights.

IX. CONCLUSION

In this study, we discussed the crucial prerequisites for developing NPU architecture for a commercial mobile SoC. In addition to possessing processing efficiency in mobile environments, the NPU needs to be flexible to cope with various applications and target performances. Therefore, when implementing our flagship mobile NPU, we employed several features such as an adder-tree-based inner-product engine with mixed-precision, zero-skipping based on IFM sparsity, dynamic port assignment, and re-configurable MAC datapath. Evaluation on a silicon die indicated that our NPU achieved a peak energy-efficiency of 13.6 TOPS/W and 131.13 FPS on ResNet-50. In addition, extensive simulations verified that our optimization boosted the speed by a factor of 1.79 compared to the dense baseline.

Based on the implementation experience and evaluation results, additional analysis was performed to further the enhance the NPU performance, and three findings were presented: (i) Inter-lane static shuffling can be a satisfactory compromise between the hardware complexity and performance due to the uneven distribution pattern of nonzero IFM. (ii) Limiting the re-configurability can be reduce the performance slightly; however, it can reduce the area significantly. (iii)It would be advantageous to provide an extra memory block to offer more flexibility in loading the IFM, weights, or PSUM, considering bank conflict in the on-chip scratchpad memory. These findings can contribute to improving the NPU architecture and will be applied in our future development. The practical issues dealt with in this study and the lessons learned can be beneficial for the advancement of mobile NPUs.

REFERENCES

[1] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 2818–2826.

[2] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry,

A. Askell *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[4] C. Rosset, "Turing-NLG: A 17-billion-parameter language model by Microsoft," *Microsoft Blog*, 2019.

[5] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 4780–4789.

[6] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 42, no. 1, pp. 269–284, 2014.

[7] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[8] D. Kim, J. Ahn, and S. Yoo, "ZeNA: Zero-aware neural network accelerator," *IEEE Design & Test*, vol. 35, no. 1, pp. 39–46, 2018.

[9] J. Sim, J.-S. Park, M. Kim, D. Bae, Y. Choi, and L.-S. Kim, "A 1.42TOPS/W deep convolutional neural network recognition processor for intelligent IoE systems," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, 2016, pp. 264–265.

[10] J. Song, Y. Cho, J.-S. Park, J.-W. Jang, S. Lee, J.-H. Song, J.-G. Lee, and I. Kang, "An 11.5 TOPS/W 1024-MAC butterfly structure dual-core sparsity-aware neural processing unit in 8nm flagship mobile SoC," in *2019 IEEE International Solid-State Circuits Conference (ISSCC)*, 2019, pp. 130–132.

[11] J. Jo, S. Cha, D. Rho, and I.-C. Park, "DSIP: A scalable inference accelerator for convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 53, no. 2, pp. 605–618, 2017.

[12] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, vol. 44, no. 3, pp. 1–13, 2016.

[13] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 688–698.

[14] Y. Choi, D. Bae, J. Sim, S. Choi, M. Kim, and L.-S. Kim, "Energy-efficient design of processing element for convolutional neural network," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 64, no. 11, pp. 1332–1336, 2017.

[15] D. Zhang, J. Yang, D. Ye, and G. Hua, "LQ-Nets: Learned quantization for highly accurate and compact deep neural networks," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 365–382.

[16] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[17] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[18] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 764–775.

[19] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," *Advances in Neural Information Processing Systems*, vol. 28, pp. 1135–1143, 2015.

[20] S. Lin, R. Ji, C. Yan, B. Zhang, L. Cao, Q. Ye, F. Huang, and D. Doermann, "Towards optimal structured cnn pruning via generative adversarial learning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 2790–2799.

[21] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, vol. 45, no. 2, pp. 27–40, 2017.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[23] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[24] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 2704–2713.

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[26] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[27] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," *Advances in Neural Information Processing Systems*, vol. 29, pp. 2074–2082, 2016.

[28] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm FDSOI," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, 2017, pp. 246–247.

[29] P. Gysel, J. Pimentel, M. Motamedi, and S. Ghiasi, "Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 11, pp. 5784–5789, 2018.

[30] R. Zhao, Y. Hu, J. Dotzel, C. De Sa, and Z. Zhang, "Improving neural network quantization without retraining using outlier channel splitting," *arXiv preprint arXiv:1901.09504*, 2019.

[31] R. Banner, Y. Nahshan, and D. Soudry, "Post training 4-bit quantization of convolutional networks for rapid-deployment," in *Advances in Neural Information Processing Systems*, 2019, pp. 7950–7958.

[32] A. Delmas Lascorz, P. Judd, D. M. Stuart, Z. Poulos, M. Mahmoud, S. Sharify, M. Nikolic, K. Siu, and A. Moshovos, "Bit-Tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 749–763.

[33] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[34] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional networks for biomedical image segmentation," *arXiv preprint arXiv:1505.04597*, 2015.

[35] L. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking atrous convolution for semantic image segmentation," *arXiv preprint arXiv:1706.05587*, 2017.

[36] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[37] J. H. Lee, S. Ha, S. Choi, W. Lee, and S. Lee, "Quantization for rapid deployment of deep neural networks," *arXiv preprint arXiv:1810.05488*, 2018.

[38] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[39] J.-S. Park, J.-W. Jang, H. Lee, D. Lee, S. Lee, H. Jung, S. Lee, S. Kwon, K. Jeong, J.-H. Song, and I. Kang, "A 6K-MAC feature-map-sparsity-aware neural processing unit in 5nm flagship mobile SoC," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, 2021, pp. 152–153.