Specializing CGRAs for Light-Weight Convolutional Neural Networks

Jungi Lee and Jongeun Lee, *Member, IEEE*

***Abstract*—Deep neural network (DNN) processing units, or DPUs, are one of the most energy-efficient platforms for DNN applications. However, designing new DPUs for every DNN model is very costly and time consuming. In this article, we propose an alternative approach: to specialize coarse-grained reconfigurable architectures (CGRAs), which are already quite capable of delivering high performance and high energy efficiency for compute-intensive kernels. We identify a small set of architectural features on a baseline CGRA to enable high-performance mapping of depthwise convolution (DWC) and pointwise convolution (PWC) kernels, which are the most important building block in recent light-weight DNN models. Our experimental results using MobileNets demonstrate that our proposed CGRA enhancement can deliver 8∼18× improvement in area-delay product (ADP) depending on layer type, over a baseline CGRA with a state-of-the-art CGRA compiler. Moreover, our proposed CGRA architecture can also speed up 3-D convolution with similar efficiency as previous work, demonstrating the effectiveness of our architectural features beyond depthwise separable convolution (DSC) layers.**

***Index Terms*—Coarse-grained reconfigurable architecture (CGRA), convolutional neural network (CNN), depthwise separable convolution (DSC), neural processing unit.**

I. Introduction

TODAY deep learning is one of the most important workloads to accelerate, due to its vast set of applications across several different areas, including image, speech, video, language, and game, just to name a few. To provide energy-efficient acceleration beyond those of graphics processing units (GPUs) and field-programmable gate arrays (FPGAs), several deep neural network (DNN) processing units, or DPUs, have been designed [1]. However, since DNNs are still rapidly evolving, there is a certain risk in creating and committing new DPUs.

In this article, we propose to use existing domain-specific accelerators such as coarse-grained reconfigurable architecture (CGRA) with specialization for DNN acceleration. Although the definition of CGRA is so broad as to include many different architectural styles [2]–[7], in this article we consider a very common CGRA architecture family, which is built around a 2-D array of processing elements (PEs), where each PE is a very small processor with a limited set of instructions, and PEs work together in a lock-step manner (e.g., [2] and [3]). CGRAs are known to provide very high energy efficiency beyond that of FPGAs for the right kind of applications [8], [9], and they are also flexible enough to support many different kinds of loops [7]. Since CGRAs are designed to support a range of applications especially in the domain of multimedia and digital signal processing [9], [10], it is possible to utilize CGRAs even when DNN is not used. Also CGRAs can more easily support future application changes in DNNs, such as supporting leaky rectified linear unit (ReLU) [11] and adding skip connections to an existing network.

Previous work on using CGRAs for mapping DNNs includes new architectures [4], [5] and new compilation methods [6], which all target conventional 3-D convolution only. However, for mobile applications, conventional 3-D convolution layers are superseded by light-weight models exploiting depthwise separable convolution (DSC) as exemplified in MobileNets [12], [13], ShuffleNet [14], and EfficientNets [15], due to their significantly higher inference performance and greatly reduced model size and computation complexity. DSC consists of a sequence of depthwise convolution (DWC) and pointwise convolution (PWC) layers. While PWC may account for over 90% multiply accumulate (MAC) operations, DWC can account for up to 40% *in terms of runtime* due to its low computation-to-data-transfer ratio and difficulty of mapping DWC kernels. Hence, it is important to provide optimized mapping for both DWC and PWC.

In this article, we first present our analysis showing that CGRAs are not necessarily slower than DPUs when it comes to machine learning workloads, if a right set of architectural features are provided. Based on our analysis, we present three generic architecture extensions for CGRAs—crossbar-style memory bus, dual-mode MAC unit, and operand reuse network—along with a mapping scheme that can greatly enhance CGRA's performance for DSC kernels. We also present a cross-channel optimization to ameliorate the memory bottleneck problem in DWC stemming from its low computation-to-data-transfer ratio.

Our experimental results using MobileNet V1 and V2 [16], [17] demonstrate that our proposed features can improve the efficiency of CGRA for DWC and PWC layers

Manuscript received 19 February 2021; revised 9 June 2021 and 27 August 2021; accepted 10 October 2021. Date of publication 26 October 2021; date of current version 20 September 2022. This work was supported in part by IITP through the Artificial Intelligence Graduate School Program under Grant 2020-0-01336, Neuromorphic Computing Software Platform for Artificial Intelligence Systems under Grant 1711080972, and ITRC Support Program under Grant IITP-2021-0-02052, and NRF under Grant 2020R1A2C2015066 all funded by MSIT of Korea; and in part by the Free Innovative Research Fund of UNIST under Grant 1.170067.01. The EDA tool was supported by the IC Design Education Center (IDEC), South Korea. This article was recommended by Associate Editor W. Hung. *(Corresponding author: Jongeun Lee.)*

The authors are with the Department of Electrical Engineering, Ulsan National Institute of Science and Technology, Ulsan 44919, South Korea (e-mail: jlee@unist.ac.kr).

Digital Object Identifier 10.1109/TCAD.2021.3123178

by 8× and 18×, respectively, in terms of area-delay product (ADP) over a compiler approach [7]. Moreover, though not explicitly optimized for, 3-D convolution on our architecture is also quite efficient, generating competitive performance and ADP as a CGRA [5] explicitly optimized for machine learning algorithms including 3-D convolution.

The contributions in this article include the following. First, we analyze the performance bottleneck of CGRAs for DNN acceleration. Second, we propose a small set of generic architecture extensions and a mapping scheme for DWC and PWC kernels. Third, to further increase PE utilization in DWC we propose cross-channel optimization for DWC mapping. Finally, we evaluate our proposed CGRA called neural processing-CGRA (NP-CGRA) for MobileNet models. An earlier version of this work was presented in [18], compared with which this version presents cross-channel optimization as well as more details of the proposed architecture and algorithms.

II. Background and Related Work

A. Baseline CGRA

While CGRA is a generic term encompassing many different architecture families [2]–[7], we consider the ADRES-like CGRA architecture [3] as our baseline, which is one of the most extensively studied. The main datapath of our baseline CGRA consists of a 2-D array of PEs interconnected with a mesh-like network, plus local memory implemented as multibanked static random access memory (SRAM) blocks for high on-chip bandwidth. PEs can perform arithmetic logic unit (ALU) operations and memory operations though details vary depending on architecture instances. A PE has a local register file [3], [19], [20], whose size is implementation-dependent. Some CGRAs [3], [21] assume a global register file (GRF) as well, also called *central register file*, which we include only in our extended architecture (but not in our baseline architecture). The PE operations and inter-PE connections are dynamically reconfigurable with no runtime reconfiguration overhead, thereby supporting pipelining of loops with Initiation Interval (II) greater than 1.

There are two kinds of memory operations on CGRAs in the literature: addressed versus streamed load store. Addressed load store [3] is more common among CGRA compilers as it supports random memory access but requires explicit address computation, which uses PE cycles. Streamed load store [2] does not use PE cycles but requires dedicated address generation units (AGUs), which support only a limited set of access patterns. In either case, it is possible for all connected PEs to simultaneously read data from a memory bus when needed.

B. Application Mapping for CGRA

Loops are the main target of CGRAs, and variants of modulo scheduling have been proposed to schedule loops on CGRAs [8], [9], [21]–[23]. Integer linear programming is also used to find optimal mapping for CGRAs [24]. While most CGRA scheduling algorithms deal with innermost loops only, methods to handle nested loops have been proposed, based on outer-loop parallelism [25] as well as loop flattening [26]. Conditional statements within a loop body pose a challenge in mapping [27], [28]. To go beyond loops and achieve high utilization of CGRAs at the application level, a model-of-computation-based approach has been proposed [10]. CCF [29] is a recent CGRA compiler framework.

A compiler approach has been proposed to optimize mapping of convolutional neural networks (CNNs) to CGRAs [6]. It uses an existing modulo-scheduling-based compiler infrastructure, but aims to find the best set of loop transformations (such as loop interchange and loop unrolling) for given layer parameters. However, being a pure compiler approach, this approach has limited efficiency as shown in our comparison (see Table VI). Also it has yet to be applied to DWC layers, where the lack of interchannel data reuse is very likely to limit its efficiency further. Outside of CNNs, compilation strategies to map other types of networks (e.g., recurrent neural networks [30]) have also been proposed. However, the problem with architectures or CGRAs supporting fully connected layers only, when it comes to mapping DWC, is that they have very low PE utilization because there is no way to transform DWC into matrix multiplication. DWC can only be transformed into matrix-vector multiplication, which can utilize only one row (or column) of PEs at most.

C. CGRA Architecture Exploration

CGRA architecture exploration has been performed in [31] and [32], which however does not take into account DNN workload or specific mapping schemes. While single-cycle MAC operation is common in DPUs, it is rarely supported on CGRAs by default. Our dual-mode MAC is configurable at the application granularity to minimize the cycle time impact of operation chaining. An extreme version of operation chaining has been proposed [33] in order to accelerate narrow acyclic subgraphs at subcycle granularity, which however complicates datapath, control, and compiler scheduling significantly. Our operand reuse network is an input-to-input network, whereas operand networks in [34] and [35] generally refer to output-to-input networks. Xiong *et al.* [36] proposed a reconfigurable cache architecture (shared versus private) for a tile-based architecture, which may be useful in supporting DNNs with different characteristics.

D. DPU Optimization

DSC computation has been targeted by both hard DPUs [17] and soft DPUs, but not by CGRAs. Some previous work [4], [5] proposes CGRAs optimized for DNNs. While these architectures may be called CGRAs as they are organized as arrays of simple PEs, they represent new architectures designed from scratch for DNNs, rather than exploiting existing CGRAs for DNN workloads. In this work we do not consider pruning [37] directly, but DSC already has a form of sparsity at a coarse (i.e., channel wise) granularity [38] while being much more amenable to hardware parallelization than fine-grained (i.e., element-wise) sparsity. Also we do not consider aggressive quantization, but the width of datapath is trivially configurable at design time.

TABLE I
THEORETICAL MIN LATENCY (MS, SUM OF SEVEN DWC LAYERS)

Architecture	Compute time	L1 transfer	Layer latency
CGRA baseline (4×4)	1.68	0.75∼4.10	1.68∼4.10
CGRA enhanced (8×8)	0.21	0.19	0.21
Eyeriss (168 PEs)	0.20	0.23	0.23

III. NP-CGRA ARCHITECTURE

A. CGRA Performance Bottleneck Analysis

To analyze performance bottleneck of CGRAs, we compare a baseline CGRA [6] with Eyeriss [16], a reference hard DPU, in terms of minimum theoretical latency, using seven DWC layers from MobileNet V2, one from each bottleneck (we see similar results with other layers as well). The result is summarized in Table I. The baseline CGRA has 4×4 PEs running at 500 MHz with 4-byte word size, and Eyeriss has 168 PEs running at 200 MHz with 2-byte word size.

We calculate the minimum theoretical latency simply by taking the max of compute time (assuming 100% PE utilization), L1 transfer time (i.e., on-chip memory access latency), and external memory direct memory access (DMA) time, the last of which is very small for all the cases compared, and not shown. To estimate L1 transfer time for the baseline CGRA, we assume all four load-store units (one per row) are 100% utilized, and consider two scenarios: 1) the least and 2) most data reuse of input feature map (IFM). For Eyeriss we assume 32 load-store units, and most data reuse.

Our result suggests that there is ∼8× compute time difference between the baseline CGRA and Eyeriss DPU even if we assume 100% PE utilization, which may be harder to realize for CGRA. The difference would grow if CGRA fails to reuse IFM data optimally.

To fill the gap, we consider CGRA enhanced, which is the same CGRA but with 8×8 PEs and 2-byte word size. Also, the PEs of CGRA enhanced can do MAC operation in a single cycle like Eyeriss (CGRA baseline can do either MUL or ADD, not both). These changes can bring compute time to Eyeriss level, but layer performance would still suffer due to L1 transfer bottleneck. To make it compute-bound, CGRA enhanced needs to have 16 load-store units, or *one per row and column*, and the most-data-reuse scenario.

To summarize, our analysis suggests that CGRAs can be made to deliver hard DPU-level performance, but need a few major changes: single-cycle MAC, larger array size, at least 2× on-chip memory bandwidth, and very high PE utilization. We observe similar trends with 3-D convolution layers and PWC layers though bottleneck is not always the same. Next, we present such an architecture.

B. Our Proposed Architecture Extension

Our driving application is PWC, which is also known as 1×1 convolution and algorithmically equivalent to matrix multiplication. While one can use a CGRA compiler (e.g., [29] and [39]) to compile matrix multiplication for a CGRA, it would yield a vastly suboptimal schedule. In the case of matrix multiplication, it is straightforward to find an optimal schedule manually, if one is allowed to modify the architecture slightly. The most critical architectural change is crossbar-type memory busses, as opposed to parallel busses.

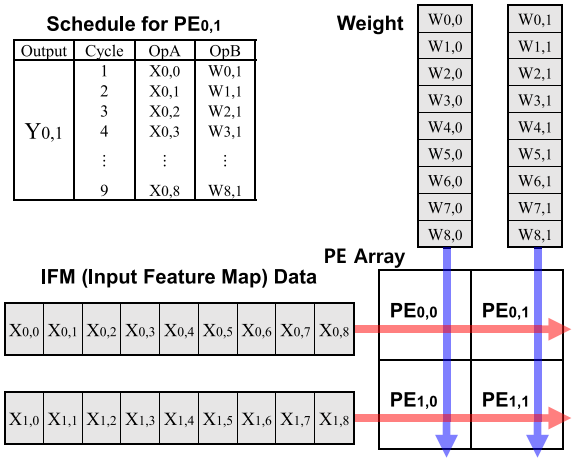


Fig. 1. Mapping PWC (or matrix mult.) to a 2×2 CGRA.

1) Crossbar-Style Memory Bus: Fig. 1 illustrates our proposed mapping for a 2×2 CGRA. In this article when referring to a PE of a CGRA, we use a zero-based subscript to specify its position (e.g., $PE_{0,1}$ denotes the PE at the first row and second column). Similarly, zero-based subscripts are used for elements of a tensor (e.g., $X_{i,j} = X[i][j]$). In the example mapping we use the first two rows of one source matrix (X) and the first two columns of the other source matrix (W), to generate the top-left 2×2 submatrix of the result matrix. The result submatrix is generated on the 2×2 PEs through a series of MAC operations (thus output stationary), as indicated by the schedule.

To provide the four PEs with correct operands, all we need is two horizontal busses and two vertical busses. Note that the data on a bus can be accessed by all connected PEs, and we add only vertical busses; horizontal busses already exist. For instance, $PE_{0,0}$ and $PE_{0,1}$ can access the same $X(0, i)$ at cycle i ($0 \le i \le 8$) through a horizontal bus (called H-bus), and $PE_{1,0}$ and $PE_{1,1}$ can share $X(1, i)$ through another H-bus. Similarly, $PE_{0,0}$ and $PE_{1,0}$ share $W(i, 0)$ through a vertical bus (V-bus), and $PE_{0,1}$ and $PE_{1,1}$ share $W(i, 1)$ through another V-bus. To use all PEs for MAC operations, streamed load store is necessary. This mapping achieves 100% PE utilization, each PE performing MUL (multiplication) *and* ADD (addition) operations every cycle, given dual-mode MAC units, explained next.

2) Dual-Mode MAC: In most CGRAs a PE performs only one operation per cycle, either MUL or ADD, which is fine if they are used intermittently. We propose configurable chaining of MUL and ADD operations, which can reduce PWC latency to half, though it may also increase cycle time. We make chaining configurable at the application granularity, so that higher clock speed is selected if the application does not use MAC chaining. We call this dual-mode MAC.

Fig. 2 illustrates the datapath of our dual-mode MAC. In the MAC mode, the multiplication and accumulation operations are chained together to realize $A \times B + C$, where operands A and B are provided through MUX A and MUX B, respectively,

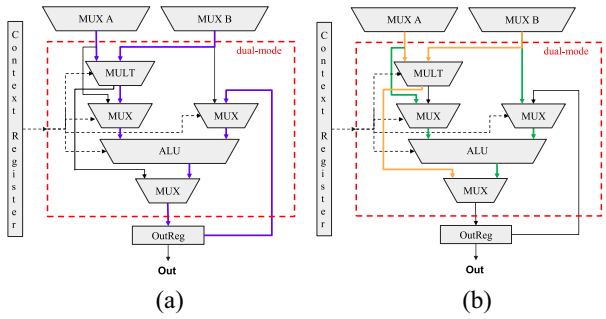


Fig. 2. Datapath of our dual-mode MAC. In (b), orange lines show the datapath for the MUL operation, green lines show the datapath for the ALU operation. (a) MAC mode. (b) MUL/ALU mode.

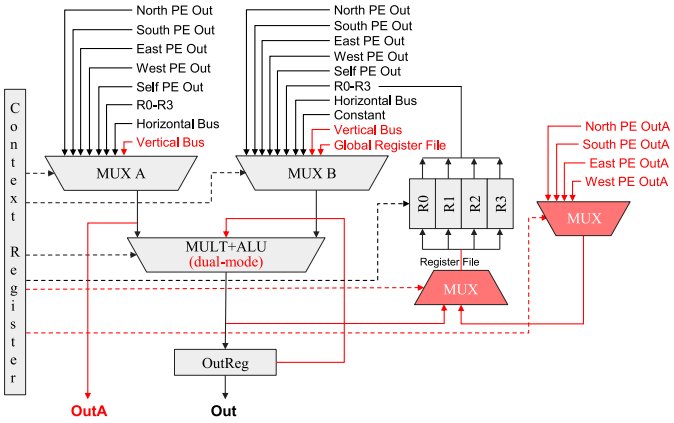


Fig. 3. Proposed PE architecture (our extension shown in red).

and operand C is provided by the output register (OutReg). In the MUL/ALU mode, on the other hand, a PE can choose either a multiplication or an ALU operation for each cycle. Note that even in the MAC mode, a PE can still perform the MUL and ALU operations individually, but enabling the MAC mode reduces the operating frequency of the entire CGRA, and thus should be done judiciously.

3) Operand Reuse Network: To make it easy to realize spatial data reuse on CGRAs we propose *operand reuse network*, which enables input-to-input routing as opposed to output-to-input routing. Consider an finite impulse response (FIR) filter example: $y_i \leftarrow w_0 x_i + w_1 x_{i+1} + w_2 x_{i+2}$, where i is the index variable of a loop that is pipelined. One way to map this loop to a CGRA is to place output variables y_i to different PEs (i.e., y_i to PE_i), called *output stationary*, and route input and coefficients to PEs. In this scheme, the same input data is used by multiple PEs at different cycles (e.g., x_2 is used by PE_0, PE_1, and PE_2 at consecutive cycles). Thus, the operand reuse network allows one of the source operands of a PE to be passed to neighbor PEs without affecting other computation that the PEs may be doing. As illustrated in Fig. 3, a PE has another output called OutA, which is the output of MUX A.

While a weight stationary scheme could realize spatial data reuse without an operand reuse network, it cannot easily utilize more PEs than the number of weight parameters. Also, the output stationary scheme is more amenable to 2-D extension.

Note that our extension requires additional MUXes only (see Fig. 3), but the local register file itself is a common

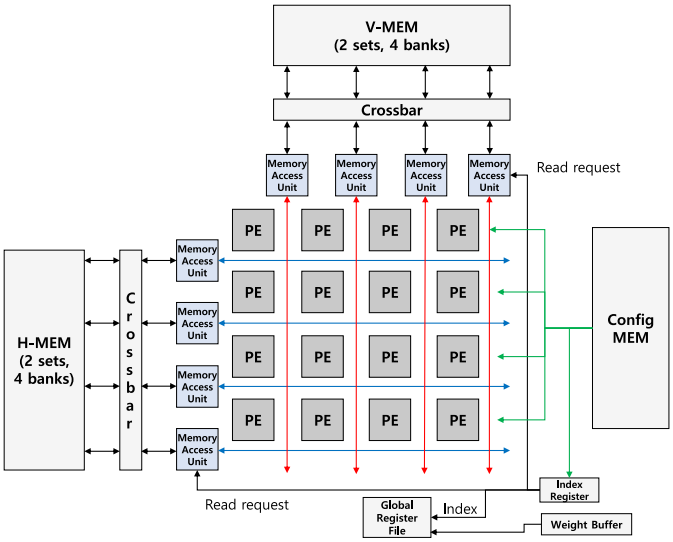


Fig. 4. Extended CGRA architecture.

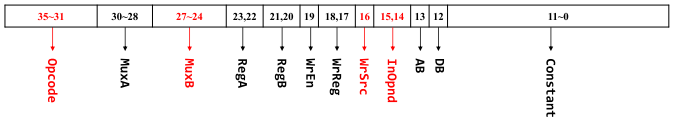


Fig. 5. Instruction definition.

feature, which we assume. Also our mapping schemes (see Section IV) use data for only one cycle. Thus, we do not use more than one entry in the local register file.

4) Other Changes: Fig. 4 shows our extended CGRA architecture. Our CGRA architecture has vertical memory (V-MEM), memory access units (MAUs), GRF, etc. Also, we change the PE structure for dual-mode MAC and operand reuse network. GRF [3], [21] is a simple register file with nine 16-bit registers, and has the following ports: one 4-bit index port (called index), one 16-bit read data port (GRF_rdata), and one 144-bit write data port (GRF_wdata). The GRF_wdata port is connected to the weight buffer and the GRF_rdata port is connected to all the PEs (one-to-all connection) through MUX B (see also Fig. 3).

The crossbar-style memory bus implies that the local memory should be divided into two, V-MEM connected to V-bus and H-MEM connected to H-bus. We set the combined size of V-MEM and H-MEM equal to that of the baseline CGRA's local memory. Also, AGUs are needed for streamed load store. In addition, for efficient mapping of DWC with stride of 1, our architecture includes a small single-port GRF, which is used to broadcast DWC weights to all PEs. The index for the GRF is given in the configuration. GRF can be filled either by DMA or through a dedicated buffer, called weight buffer, which can be very small as it is used for DWC only.

C. Instruction Format and Global Configuration

Fig. 5 illustrates our instruction (also called *context* in [2]) format to accommodate the architectural changes, where the updated fields are shown in red. We use the 32-bit instruction format from the CCF framework [29] as the baseline. The

TABLE II
PARAMETERS AND VARIABLES

Symbol	Meaning
N_r, N_c	Number of rows/columns of a CGRA's PE array
K, S	Kernel size and stride of convolution
N_i, N_o	The number of input/output channels
N_h, N_w	The height and width of OFM (Output Feature Map)
$B_r \times B_c$	Number of tiles in the current block (row×column)
AID_r, AID_c	Zero-based row (or column) number of an H(V)-AGU
N_a	Number of address bits of a bank (H-MEM or V-MEM)
tid_r, tid_c	Zero-based row/column coordinate of a tile within a block
t_{cycle}	Cycle count. A variable whose value is incremented every clock cycle and reset when a new tile starts
t_{wrap}	Wrap count. A variable whose value is incremented on every row index change and reset on every tile start
t_{wcycle}	Same as t_{cycle} but reset when t_{wrap} changes

```
for (h = 0; h < N_h; h++)
  for (w = 0; w < N_w; w++)
    for (c = 0; c < N_i; c++)
      for (i = 0; i < K; i++)
        for (j = 0; j < K; j++)
          Y[c,h,w] += W[c,i,j] × X[c,S h + i, S w + j];
```

Fig. 6. DWC kernel (a bias term is omitted).

RegA and RegB fields indicate the register indices for MUX A and MUX B, respectively (see Fig. 4). We assume four local registers per PE, following previous work [7], [20]. WrEn is write-enable for the private register file whereas WrReg is the register index for the register write operation. WrSrc selects the data to write between the MAC output and a neighbor PE's OutA (i.e., the control signal for the new 2-to-1 MUX), while InOpnd determines which neighbor's OutA passes the 4-to-1 MUX. address bus (AB) is the bit indicating to send a read request to on-chip memory (the output of the PE acts as the address, thus supporting addressed access), and data bus (DB) is the write request bit indicating that the PE output is the write data. In total, our instruction requires four more bits compared to that of the baseline PE, resulting in 36-bit instructions. In the configuration memory, four more bits are needed for the index of the GRF, plus 2 bits for H- and V-memory read request. Thus, the total number of bits of the configuration is $36 \cdot N_r \cdot N_c + 6$, where N_r and N_c are the number of rows and of columns of a CGRA PE array.

IV. APPLICATION MAPPING FOR NP-CGRA: DWC CASE

We now present our application mapping for DWC kernels (PWC mapping is already outlined in Section III-B1). Table II lists parameters and variables used. First we present a general method that works for any stride, then an optimized version for $S = 1$, which is most common. While in this article we mainly describe PE scheduling and data routing, which is crucial for maximizing PE utilization and minimizing memory access, our implementation and evaluation results include complete mapping including data layout and AGU algorithms (see Section V).

A. Depthwise Convolution

Fig. 6 illustrates a DWC kernel. Contrary to 3-D convolution, DWC requires the same number of channels (N_i, also known as depth) in all of IFM, output feature map (OFM), and weight tensors. Also, DWC computes each output channel (Y) by a 2-D convolution between one input channel (X) and one weight channel (W).

$X_{0,0}$	$X_{0,1}$	$X_{0,2}$	$X_{0,3}$	$X_{0,4}$	$X_{0,5}$	$X_{0,6}$	$X_{0,7}$	$X_{0,8}$
$X_{1,0}$	$X_{1,1}$	$X_{1,2}$	$X_{1,3}$	$X_{1,4}$	$X_{1,5}$	$X_{1,6}$	$X_{1,7}$	$X_{1,8}$
$X_{2,0}$	$X_{2,1}$	$X_{2,2}$	$X_{2,3}$	$X_{2,4}$	$X_{2,5}$	$X_{2,6}$	$X_{2,7}$	$X_{2,8}$
$X_{3,0}$	$X_{3,1}$	$X_{3,2}$	$X_{3,3}$	$X_{3,4}$	$X_{3,5}$	$X_{3,6}$	$X_{3,7}$	$X_{3,8}$
$X_{4,0}$	$X_{4,1}$	$X_{4,2}$	$X_{4,3}$	$X_{4,4}$	$X_{4,5}$	$X_{4,6}$	$X_{4,7}$	$X_{4,8}$
$X_{5,0}$	$X_{5,1}$	$X_{5,2}$	$X_{5,3}$	$X_{5,4}$	$X_{5,5}$	$X_{5,6}$	$X_{5,7}$	$X_{5,8}$
$X_{6,0}$	$X_{6,1}$	$X_{6,2}$	$X_{6,3}$	$X_{6,4}$	$X_{6,5}$	$X_{6,6}$	$X_{6,7}$	$X_{6,8}$
$X_{7,0}$	$X_{7,1}$	$X_{7,2}$	$X_{7,3}$	$X_{7,4}$	$X_{7,5}$	$X_{7,6}$	$X_{7,7}$	$X_{7,8}$
$X_{8,0}$	$X_{8,1}$	$X_{8,2}$	$X_{8,3}$	$X_{8,4}$	$X_{8,5}$	$X_{8,6}$	$X_{8,7}$	$X_{8,8}$

(a)

	$PE_{0,0}$		$PE_{0,1}$	
Cycle	opA	opB	opA	opB
1	$X_{0,0}$	$W_{0,0}$		
2	$X_{0,1}$	$W_{0,1}$		
3	$X_{0,2}$	$W_{0,2}$	$X_{0,2}$	$W_{0,0}$
4			$X_{0,3}$	$W_{0,1}$
5			$X_{0,4}$	$W_{0,2}$

	$PE_{1,0}$		$PE_{1,1}$	
Cycle	opA	opB	opA	opB
1	$X_{2,0}$	$W_{0,0}$		
2	$X_{2,1}$	$W_{0,1}$		
3	$X_{2,2}$	$W_{0,2}$	$X_{2,2}$	$W_{0,0}$
4			$X_{2,3}$	$W_{0,1}$
5			$X_{2,4}$	$W_{0,2}$

(b)

Fig. 7. Mapping DWC on a 2×2 CGRA ($K = 3$ and $S = 2$). (a) IFM data (shown in gray is an input tile). (b) Schedule.

B. Tiles and Blocks

A common technique to maximize data reuse in a multi-nested loop is to divide the work into *blocks* such that the entire input and output data for a block of computation can fit in the local memory, which is commonly known as *loop blocking* (or *tiling*) transformation. The same transformation can also be used to specify parallelism employed by array processors [40]. To distinguish the two different use cases, we use the term *block* to refer to the first usage, and *tile* for the second. In other words, a *block* is the amount of work that can be done using only the *local* data. A *tile* is the amount of work that is done *simultaneously* by a CGRA; therefore, tile size is determined by the CGRA size. Block size is determined by the local memory size, and is usually a multiple of the tile size.

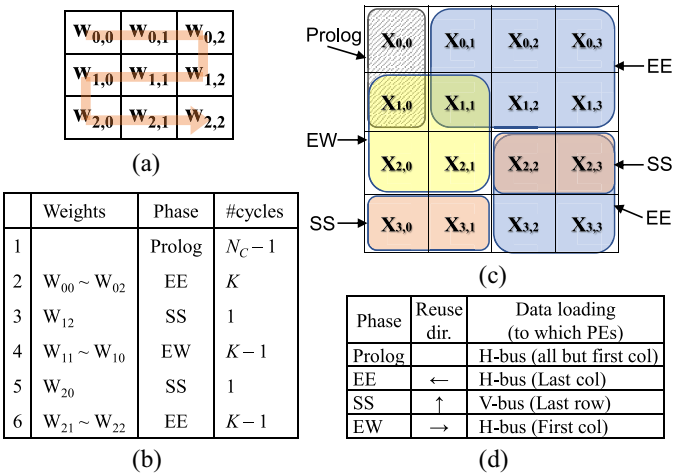
C. Depthwise Convolution With Arbitrary Stride

Consider mapping DWC with $K = 3$ to a 2×2 CGRA. Here, we parallelize the computation of one channel across the PE array. The following equations reveal the terms needed to compute the first 2×2 output, an *output tile*:

$$y_{0,0} = w_{0,0}x_{0,0} + w_{0,1}x_{0,1} + w_{0,2}x_{0,2} + w_{1,0}x_{1,0} + \cdots + w_{2,2}x_{2,2}$$
$$y_{0,1} = w_{0,0}x_{0,2} + w_{0,1}x_{0,3} + w_{0,2}x_{0,4} + w_{1,0}x_{1,2} + \cdots + w_{2,2}x_{2,4}$$
$$y_{1,0} = w_{0,0}x_{2,0} + w_{0,1}x_{2,1} + w_{0,2}x_{2,2} + w_{1,0}x_{3,0} + \cdots + w_{2,2}x_{4,2}$$
$$y_{1,1} = w_{0,0}x_{2,2} + w_{0,1}x_{2,3} + w_{0,2}x_{2,4} + w_{1,0}x_{3,2} + \cdots + w_{2,2}x_{4,4}.$$

The *input tile*, which is the set of IFM data needed to produce an output tile, is the gray-filled rectangle in Fig. 7(a).

Fig. 7(b) illustrates our proposed schedule. For instance, to compute the top row of the output tile, the top three rows of the input tile are needed, which are given sequentially through an H-bus. Each PE performs MAC operations when they see the corresponding input data on the H-bus, which simplifies schedule. One can see that our schedule achieves maximal data



(a)

	Weights	Phase	#cycles
1		Prolog	N_C-1
2	$W_{00} \sim W_{02}$	EE	K
3	W_{12}	SS	1
4	$W_{11} \sim W_{10}$	EW	$K-1$
5	W_{20}	SS	1
6	$W_{21} \sim W_{22}$	EE	$K-1$

(b)

(c)

Phase	Reuse dir.	Data loading (to which PEs)
Prolog		H-bus (all but first col)
EE	←	H-bus (Last col)
SS	↑	V-bus (Last row)
EW	→	H-bus (First col)

(d)

Fig. 8. Schedule and data movement for DWC ($S = 1$). (a) Weight processing order. (b) Phase definition. (c) How IFM data is accessed. (d) Data reuse and loading.

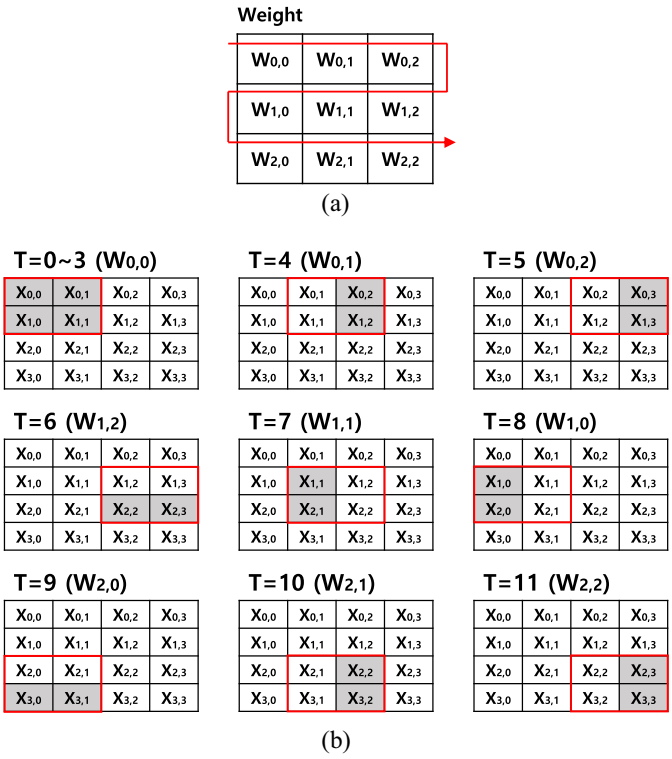
reuse within each row, since the data needed for each row is presented only once. Weight parameters can be provided through V-busses because each column of PEs use the same weight parameters every cycle.

D. Depthwise Convolution With $S = 1$

Consider an example where $K = 3$ and the CGRA size is 2×2. Again we handle one channel at a time. Similar to the general version, our scheme is output stationary such that after a certain number of cycles the 2×2 PE array will contain the data for the first 2×2 output. The key problem is how to feed all the PEs with necessary input/weight data every cycle without oversubscribing memory access resources.

Fig. 8 illustrates our solution. During the initial N_c-1 cycles (called *prologue*), IFM data [the top-left $N_r \times (N_c - 1)$ sub-matrix] is loaded through H-busses into all PEs except the first column. For the next K cycles, the PE array processes the first row of the weight matrix using IFM data partially reused from the previous cycle (from the east-side PEs) and partially loaded from local memory (for the easternmost column), which is called expand east (EE) phase. In the next cycle, the PE array processes $W_{1,2}$, which requires reusing IFM data from the south-side PEs and the southernmost PEs to load new IFM data, called the shift south (SS) phase. In the next $K - 1$ cycles, the PE array processes the remaining elements of the second row of weight, which is similar to the EE phase except that we expand west (EW), thus called EW. This pattern of EE-SS-EW-SS is repeated until we finish processing all weight. In this schedule all PEs use the same weight element, which is provided by GRF, indexed by the CGRA controller.

This schedule takes $N_c - 1 + K^2$ cycles, including prologue, except for initial memory streaming delay and final cycles for writing output data back to local memory (see Fig. 10 for the complete schedule). The data layout and AGU logic to support the above access pattern are a little complicated due to the SS phase. An alternative would be to load data for the southernmost PEs through H-bus over N_c cycles, which increases latency significantly. We place the full IFM data in H-MEM and the part needed for the SS phases in V-MEM. Loading data to both H-MEM and V-MEM is done by DMA.



(a)

(b)

Fig. 9. Data access patterns in DWC. (a) Weight data access pattern. (b) IFM data access pattern.

Fig. 9 illustrates how data reuse can help achieve high performance in DWC. In this example, the DWC weight matrix is 3× 3 matrix (for one channel), stride is one, and the CGRA size is 2 × 2. Only one channel is considered in this mapping, which is repeated for all channels to complete DWC.

To achieve 100% PE utilization, we must generate 2×2 output in 9 cycles ($= K^2$ for our example), assuming each PE can do one MAC operation per cycle. Fig. 9(b) shows how to achieve that, with details such as which elements of the IFM (indicated by red boxes) and which weight element are used by the CGRA in each cycle. Moreover, only the gray elements are loaded from the memory and the white IFM elements in red boxes are received from neighbor PEs and thus reused, which is crucial to achieving optimal mapping with limited memory bandwidth. Most of the memory accesses can be fulfilled by H-busses, with a few exceptions; $T = 6$ (or $T = 9$) can be done in a single cycle by utilizing V-busses, and step 1 takes two cycles but can be done as part of initialization and potentially merged with other operations.

Fig. 10 is a cycle-by-cycle diagram showing how data flows among PEs and on-chip memories for the same example. Cycles 0–1 are due to the initial data streaming delay while cycle 1 is prologue. One can note that the memory access pattern is the same for the cycles belonging to the same phase (e.g., cycles 3–5, 10 and 11). The diagram confirms that this

TABLE III
PERFORMANCE ANALYSIS

	Tile latency ($= T$)	Block latency	Layer latency
PWC	$N_i + \lambda$	$B_r B_c T$	$B_r B_c T \cdot \lceil \frac{N_w}{B_r N_r} \rceil \cdot \lceil \frac{N_o}{B_c N_c} \rceil \cdot N_h$
DWC General	$K((N_c-1)S+K)+\lambda$		$B_r B_c T \cdot \lceil \frac{N_h}{B_r N_r} \rceil \cdot \lceil \frac{N_w}{B_c N_c} \rceil \cdot N_i$
DWC Optimized	$K^2 + N_c - 1 + \lambda$		

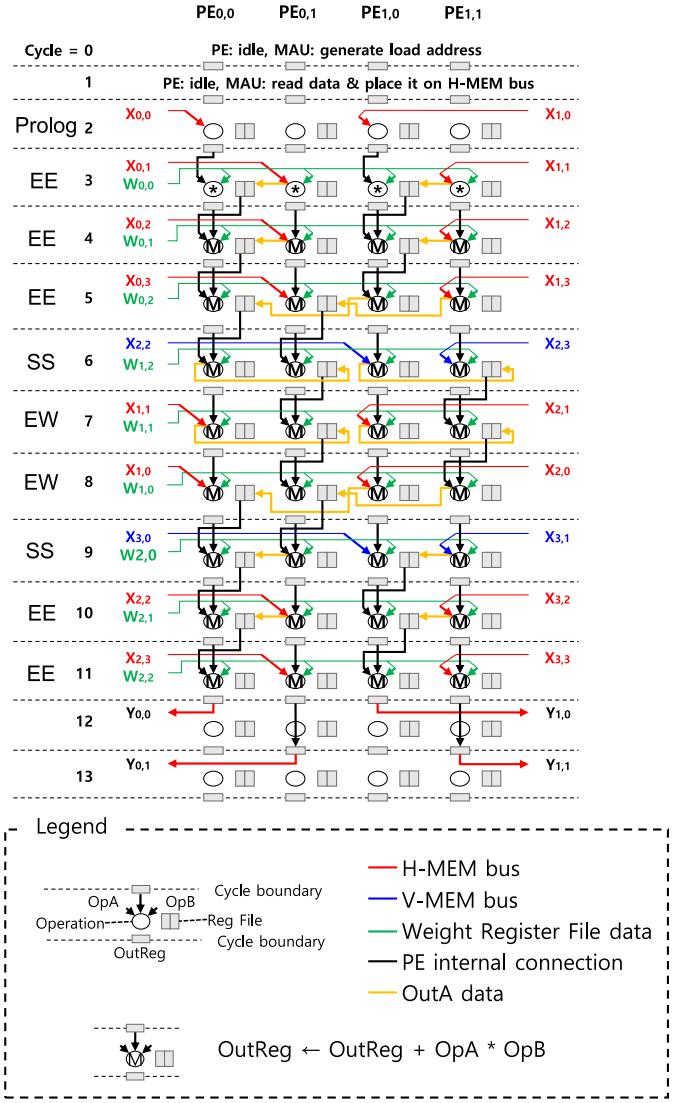


Fig. 10. Detailed dataflow of DWC ($S = 1$) mapped to 2×2 CGRA.

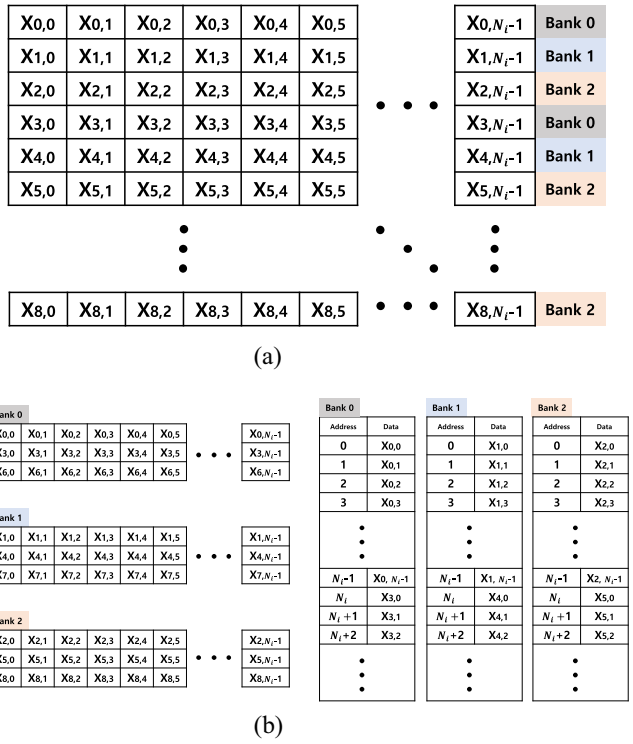


Fig. 11. PWC IFM data in external memory and H-MEM. (a) Logical view of IFM data, and H-MEM bank assignment. (b) Partitioning of IFM data into banks, and IFM data layout.

schedule takes ten cycles excluding the initial and final delays due to memory access.

E. Performance Analysis

Table III summarizes latency of each mapping, where λ is used to capture constant delay due to initial/final delay in pipelining. Note that the analytical performance models are provided only to characterize our mapping scheme. Our performance evaluation is based on cycle-accurate simulation (see Section VI-A).

PWC: PWC mapping multiplies $N_w \times N_i$ IFM matrix with $N_i \times N_o$ weight matrix N_h times. In order to multiply IFM and weight matrix, these matrix should be divided into $B_r N_r \times N_i$ and $N_i \times B_c N_c$ blocks. The number of blocks is $\lceil N_w/(B_r N_r) \rceil \times \lceil N_o/(B_c N_c) \rceil$, each of which takes $B_r B_c T$ cycles, producing the layer latency.

DWC: DWC General (i.e., arbitrary stride) mapping divides IFM data by block size per channel. To compute one channel, $\lceil N_h/(B_r N_r) \rceil \times \lceil N_w/(B_c N_c) \rceil$ blocks are generated. When processing one tile, $((N_c - 1) \cdot S + K)$ IFM data is used, along with $1 \times K$ weight data, which is repeated K times. Thus, the tile latency is $K \cdot ((N_c - 1) \cdot S + K) + \lambda$. DWC General shares the layer latency formula with DWC Optimized (i.e., $S = 1$), though tile latency is different.

V. Data Layout and Address Generation

We now present our mapping methods for PWC and DWC kernels in more detail, focusing on data movement within the PE array and between PEs and memories.

A. Pointwise Convolution

Fig. 11(a) illustrates how IFM data can be stored in the external memory. To move this block of data to the local

memory (H-MEM), we first assign consecutive rows of the IFM data into different banks as illustrated in Fig. 11(b). Then the rows assigned to the same bank are combined together in a sequential manner and stored into the local memory banks as illustrated in the IFM data layout. This ensures that all the IFM data can be fed to correct PEs without ceasing. Weight data are stored in the V-MEM local memory in a similar fashion, with one difference that weight data need to be partitioned along the *column* direction, requiring matrix transpose or reshaping. However, since weight data are constant during inference, any preprocessing, if needed, can be done in advance before runtime.

To utilize all PEs for MAC operations, we delegate address generation to AGUs in MAUs. For PWC, generating addresses to access V-MEM and H-MEM is straightforward. The V-MEM address is given as follows:

$$\text{addr} = (\text{AID}_c \ll N_a) \mid \left(\text{tid}_c \cdot N_i + t_{\text{cycle}}\right) \tag{1}$$

where $\ll$ and $\mid$ represent the bitwise left shift and bitwise OR operations (same as in C). This address value, addr, can be easily computed by an AGU from t_{cycle} variable, and is shared among all AGUs. The H-MEM address should distinguish between load and store, since H-MEM is used for both OFM and IFM. The detailed algorithms to generate memory addresses for our PWC and DWC mappings are given in the Appendix.

B. Depthwise Convolution With Arbitrary Stride

The data layout to support the proposed mapping is illustrated in Fig. 12. First the rows of the IFM data (which can be regarded as 2-D since we consider only one channel at a time) are mapped to banks as Fig. 12(a), where the idea is to map each set of continuous S rows starting from the top to the next bank. Second, all the rows mapped to a bank are combined and placed into the bank in a sequential manner [see Fig. 12(b)].

Note that contrary to PWC, the data layout for DWC does not place all the IFM data needed for one row of CGRA PEs into one bank. But this does not cause a problem, since 1) there is a crossbar switch between the set of H-AGUs and the set of memory banks and 2) there is no bank conflict (i.e., all H-AGUs access different memory banks all the time). To show the absence of bank conflict, it suffices to see that the second H-AGU always accesses an input row that is S-rows below what the first H-AGU accesses, and so on.

The weight parameters needed by PEs are uniform vertically but not uniform horizontally, which suggests that using V-busses is beneficial. Thus, we store weight parameters in V-MEM (duplicated in all banks) and use V-busses to provide weight parameters for PEs as in PWC mapping.

The address generated by V-AGUs is: $\text{addr} = (\text{AID}_c \ll N_a)|(t_{\text{wcycle}} - \text{AID}_c \cdot S + t_{\text{wrap}} \cdot K)$. Here, t_{wrap} tracks which IFM row the CGRA is currently processing, or the *row* number of weight parameters accessed by PEs.

C. Depthwise Convolution With $S = 1$

DWC with $S = 1$ uses the same method for storing data in H-MEM. But this method cannot be used for V-MEM

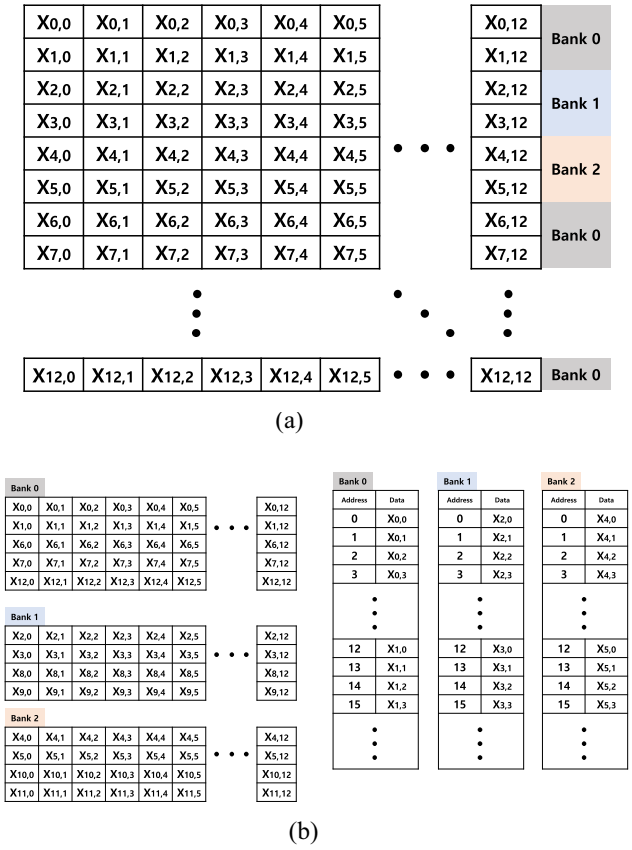


Fig. 12. DWC with arbitrary stride IFM data in external memory and H-MEM. (a) Logical view of IFM data, and H-MEM bank assignment. (b) Partitioning of IFM data into banks, and IFM data layout.

because V-MEM stores only the data required for the SS phase. The data layout to support the proposed mapping is illustrated in Fig. 13(a). Since they are separated by the interval of the CGRA column size, they store in V-MEM based on this interval. For example, $X_{3,2}$, $X_{3,5}$, and $X_{3,8}$ are stored in Bank 0. Fig. 13 shows the data layout in the external memory and partial IFM data partitioned into banks and laid out on V-MEM. Weight data for DWC with $S = 1$ is stored in GRF.

D. Cross-Channel Optimization for DWC

DWC kernels have relatively high data-transfer-to-computation ratio, which can cause performance degradation due to some portion of L2 data transfer latency not hidden behind the CGRA's computation time. Often this problem is caused not by the lack of bandwidth with the external memory, but rather access latency, which is exacerbated when the unit of access is small. This is what happens in our DWC mapping. Our DWC mapping algorithm discussed so far processes one channel at a time; i.e., our CGRA does the computation for one channel and the loading/storing of necessary data before moving on to the next channel. In such a mapping we find that it takes more time to transfer data between the external memory, which is often dynamic random access memory (DRAM), and local memories

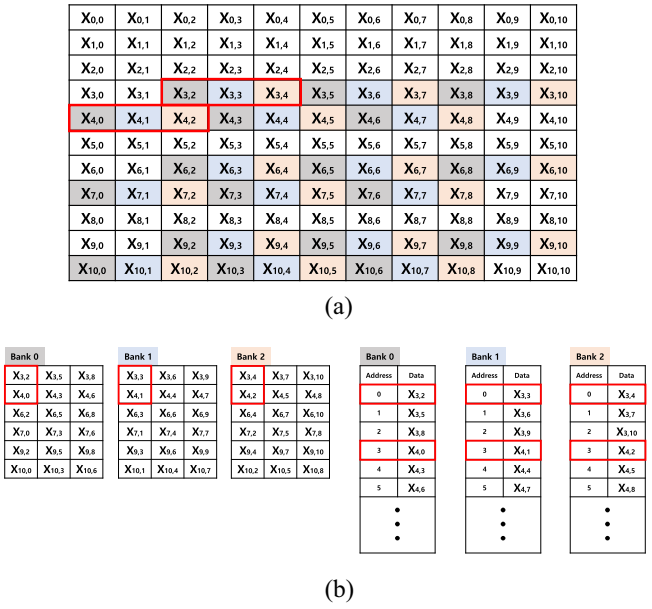


Fig. 13. IFM data in external memory and in V-MEM for DWC with stride 1 (shown in red is a tile). (a) Logical view of IFM data and V-MEM bank assignment. (b) Partial IFM data partitioned into banks, and laid out on V-MEM.

(H- and V-memories) compared with computation time, if the height and width of IFM are small.

To solve this problem we change the mapping such that multiple channels are processed continuously. It is very similar to prefetching data for the next channel, which can ameliorate the memory bottleneck problem for DWC layers at the expense of higher on-chip memory usage. The memory requirement for cross-channel optimization is given as follows:

$$2 \cdot \left(N'_h \cdot N'_w + N_h \cdot N_w\right) \leq \text{H-MEM size} \tag{2}$$

where N'_h and N'_w are the height and width of IFM. The requirement states that the (on-chip) H-memory size should have enough to contain two channels' worth of data for DWC mapping. Our cross-channel optimization enables the prefetching of the next channel's data while the PE array is doing computation for the current channel.

VI. Experiments

A. Experimental Setup

To evaluate the effectiveness of our proposed architecture, we use MobileNets and compare against previous CGRA approaches as well as other DPUs. However, since MobileNet results are not reported by previous CGRA architectures, we also map AlexNet convolution layers to NP-CGRA and compare our result with those of previous CGRAs and DPUs as reported in the literature.

Our main comparison metric is inference throughput (frames/s) and cost efficiency (in ADP). We have developed a cycle-accurate simulator and also created a register transfer level (RTL) design for the baseline CGRA and our NP-CGRA, including PE array, AGUs, GRF, and the CGRA controller, which we have validated in terms of functionality and cycle-level behavior. For area and power estimation we have synthesized RTL designs using Synopsys Design Compiler with Samsung 65 nm standard-cell library. The area and power of on-chip memories are estimated using Cacti 7.0 [41].

TABLE IV
NP-CGRA Specifications

Number of PEs	64 (8×8)
Word size	16-bit
Clock frequency	500 MHz
Off-chip memory bandwidth	12.5 GB/s
DMA latency	200 cycles
H-MEM size (= V-MEM size)	39 KB (×2 sets)
Configuration memory size	9248 bytes (2312×32 bits)
Weight buffer size	1152 bytes (144×64 bits)

Table IV summarizes the specification of NP-CGRA. The off-chip memory bandwidth is set to 12.5 GB/s as in SDT-CGRA [5]. H-MEM and V-MEM have the same size, which is set to $N_i K^2 \times N_r$ words, to make mapping AlexNet easier, although smaller memory sizes can also be accommodated by our mapping strategy. The number of configuration bits per cycle is $2312 = 36 \times 64 + 8$; each PE needs four more bits than the baseline PE due to increased input MUX sizes (1 bit) and the operand reuse network's MUXes (3 bits), and eight more bits globally for GRF index and to control streamed load store. The number of contexts supported in our implementation is 32, which gives the total configuration memory size as listed in Table IV. weight buffer, which is optional, is set to hold 64 copies of GRF contents.

B. Depthwise Separable Convolution Results

We use the first three layers right after the first standard convolution (i.e., 3-D convolution) layer in MobileNet V1 [12] (width multiplier 1, resolution 224). We compare three cases:

1) *Baseline+CCF:* Baseline CGRA with CCF compiler [29].
2) *Matmul DWC:* NP-CGRA + Matrix multiplication-based DWC.
3) *Our Mapping:* NP-CGRA + Our mapping scheme for PWC/DWC.

For this experiment only, the CGRA size is set to 4×4 due to CCF compilation flow (for all three cases). The clock speed is 500 MHz for both the baseline and NP-CGRA.

The first case represents the state-of-the-art CGRA solution. For CCF, we apply loop pipelining to the loop level with the largest trip count, which is image height (N_h). The second case uses our mapping scheme for PWC only. DWC is converted into matrix multiplication by im2col, essentially using only one column of a CGRA, to which the K^2 dimension is mapped. The im2col time is not taken into the account in this part.

Table V summarizes the result. The architectural factor is about 2×, since our NP-CGRA has 2× faster arithmetic and memory operation rate than the baseline CGRA. So the large performance difference is attributed to mapping. A close look at the generated code has revealed that CCF generates extra one MUL and three ADD ops for every MAC operation (one MUL and one ADD) in the program, which is due to address

TABLE V
MOBILENET V1 DSC RESULT (MD: MATMUL DWC)

Layer	Latency (ms) CCF	MD	Ours	PE utilization (%) CCF	MD	Ours	ADP (mm^2·ms) CCF	MD	Ours
PWC	78.91	3.72	3.72	8.14	86.42	86.42	122.48	6.85	6.85
DWC S=1	11.10	2.82	0.92	8.14	16.04	49.00	17.22	5.19	1.70
DWC S=2	7.74	1.41	0.81	5.83	16.01	28.00	12.02	2.60	1.49

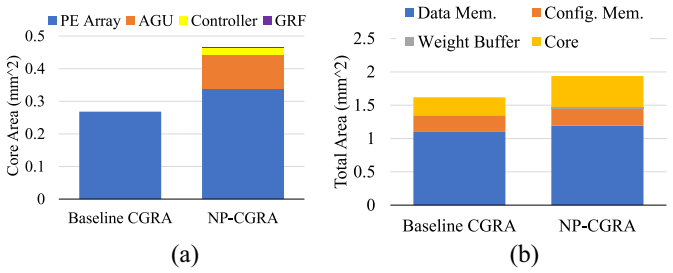


Fig. 14. Area comparison. (a) Core area comparison. (b) Total area comparison.

generation as it uses addressed load store. Also the scheduled code has some empty slots, which further lowers the PE utilization. Overall, the mapping efficiency difference is about 10× in the case of PWC for the relatively small CGRA size. We expect the difference to increase for larger CGRA sizes. All in all, our NP-CGRA generates over 20× speed up and close to 18× ADP reduction for PWC over the baseline (our architecture has 20% larger total area including SRAM memory; synthesis result is discussed in Section VI-C).

For DWC our NP-CGRA continues to deliver better performance and ADP than the baseline. While the utilization of the Matmul DWC case is around 16% (and cannot exceed 25% using only one CGRA column), our DWC mapping generates about 1.75∼3× higher performance and efficiency than the matmul-based mapping. Note that DWC ($S = 2$) layers are the rarest in MobileNets while PWC accounts for the majority of MAC operations, which may justify relatively low effort made in optimizing for the former case.

C. Hardware Overhead Evaluation

Fig. 14 compares the synthesized areas of two 8×8 CGRAs at the target frequency of 500 MHz (timing met in both). The largest area increase in the core area comes from AGUs, which may be justified given the so many freed PEs by AGUs. The common logic and variables used by AGUs such as iterators are implemented in the controller, shown in the graph. The increase in the PE array is modest (the baseline architecture has a homogeneous operation set, meaning all PEs support MUL and ADD operations). On the other hand, the total area is dominated by SRAM memories, putting the overall area overhead of NP-CGRA at 20%.

While we use the same clock frequency for both CGRAs in our ADP evaluation, our dual-mode MAC does increase the critical path delay. When driven for maximum speed, the critical path delay is increased from 1.23 (baseline) to 1.65 ns (NP-CGRA), which is due to the difference between MAC delay (1.08 ns) and MUL delay (0.68 ns). Considering the potential 2× increase in computation throughput, the 34% increase in cycle time seems justifiable. On the other hand, MAC operations are not utilized by current CGRA compilers (e.g., CCF), which can limit applicability.

D. Comparison With Previous Work Using MobileNet

No previous CGRA reports MobileNet or DSC performance. A few MobileNet accelerators for FPGAs exist but the lack of reported standard-cell result makes it difficult to compare with them directly. Eyeriss v2 [17] targets MobileNet V1 with width multiplier 0.5 and resolution 128, which we compare in Table VI. Eyeriss v2 has much more capable PEs than NP-CGRA, performing two MAC ops per cycle, which partially explains higher absolute performance compared with NP-CGRA. On the other hand, NP-CGRA is much smaller. Also Eyeriss v2 uses 8-bit data width, so we convert the gate count to 16-bit equivalent by multiplying it by 2, which we believe is conservative. Overall, the NP-CGRA turns out to have much higher cost efficiency compared with Eyeriss v2. Even if we assume the same clock speed for both architectures, NP-CGRA can deliver over 2× higher performance per logic area.

We have also compared energy efficiency. The energy numbers for the previous work were obtained from their respective papers. Our energy estimation includes both dynamic and leakage power of the CGRA core as well as all the on-chip memories. Similar to the area breakdown of Fig. 14, the configuration memory accounts for about 31% of the total energy consumption regardless of the layer type, while the CGRA core accounts for about 25% (in the case of PWC and DWC with $S = 2$) ∼ 45% (DWC with $S = 1$), with the rest being mostly due to data memories. We find that for MobileNet V1, NP-CGRA consumes about 27% higher energy than Eyeriss v2. One reason for this is the higher clock frequency of NP-CGRA, but the use of configuration memory also makes NP-CGRA not as energy-efficient as DPU hardware. Also note that in practice CGRAs are used with a main processor, which can further add to its energy overhead.

E. AlexNet Convolution Layer Results

While our architecture is not explicitly optimized for 3-D convolution, we map AlexNet convolution layers to NP-CGRA, for quantitative comparisons with previous CGRA results as well as to see broader applications of our extensions outside DSC layers (see Table VI). For NP-CGRA, we convert convolution into matrix multiplication using im2col and use PWC mapping. The im2col part is assumed to be done on the ARMv8 processor on Xilinx Ultra96-V2 board, which we have used to measure the runtime of im2col functions. The auto-tuning approach [6] applies various combinations of loop transformations (e.g., interchange and unrolling) to find the best loop nest for CGRA mapping, which is done by an in-house CGRA compiler. SDT-CGRA [5] is a novel architecture optimized for machine learning algorithms including CNNs. Eyeriss [16] and Eyeriss v2 [17] are hard DPUs optimized for CNNs.

As expected, the auto-tuning approach has the lowest performance and efficiency, attributed to poor scheduling.

TABLE VI
COMPARISON WITH PREVIOUS CGRA AND DPU IMPLEMENTATIONS

	Eyeriss [16]	Eyeriss-v2 [17]	Auto-tuning [6]	SDT-CGRA [5]	NP-CGRA (Ours)
Technology	ASIC (65 nm)	ASIC (65 nm)	CGRA (32 nm)	CGRA (55 nm)	CGRA (65 nm)
Clock frequency (MHz)	200	200	500	450	500
#PEs (#Ops/cycle)	168 (336)	192 (768)	16 (16)	25 (205)	64 (128)
Data width (bits)	16	8	32	16	16
On-chip data memory (kB)	108	192	320	54.6	156
NAND-2 gate count (logic only) or area	1176k	2695k	47k*	5.19 mm^2†	323k
Gate count or area, converted for 65 nm, 16-bit	1176k	5390k	47k*	7.25 mm^2†	323k
MobileNet V1 runtime (DSC only, ms)	-	0.78	-	-	2.22
MobileNet V2 runtime (DSC only, ms)	-	-	-	-	13.29
AlexNet (Conv. runtime, ms)	28.82	9.79	990	23.24	40.07
AlexNet energy (mJ)	8	-	149	35.6	-
MobileNet V1 energy (mJ)	5.72	0.51	-	-	0.65

*Not reported in the paper, and assumed to be the area of the 4×4 baseline CGRA.
†The area number of SDT-CGRA is for a full chip including SRAM whereas gate count numbers of the others are for logic only.

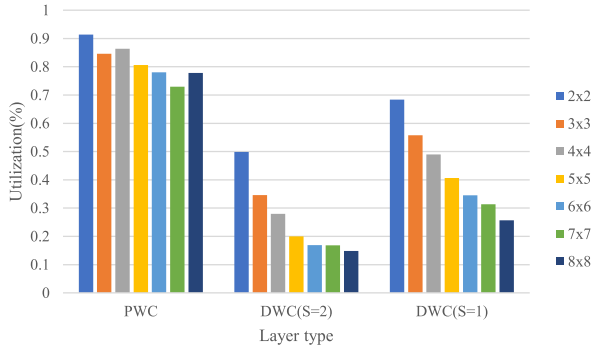


Fig. 15. PE utilization for different array sizes.

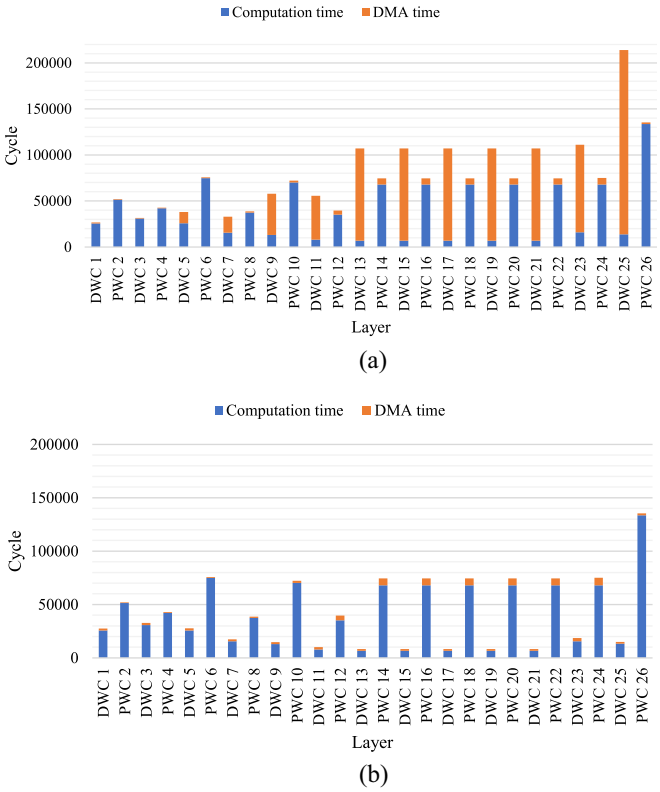


Fig. 16. Effect of cross-channel optimization (MobileNet V1). (a) Runtime before cross-channel optimization. (b) Runtime after cross-channel optimization

Eyeriss and Eyeriss v2 are among the fastest. Our NP-CGRA result does not include the area of the ARM processor, but it is quite competitive with other CGRA or DPU architectures in terms of both speed and cost efficiency, demonstrating the efficacy of our extensions beyond DSC layers.

F. Effect of CGRA Size

Fig. 15 shows PE utilization of each mapping for different CGRA sizes. For this study we use the first three layers after the first 3-D convolution layer in MobileNet V1, which include one PWC and two DWC layers. We observe that PWC mapping generates the highest PE utilization ranging 73%–91%, which is good because PWC accounts for the most MAC operations in light-weight CNNs. The main reason why PWC mapping shows less than 100% utilization is the output data writeback phase, which takes N_r or N_c cycles after each PWC kernel execution, as well as fragmentation, which occurs due to boundary tiles being less than the CGRA size. The general trend of decreasing PE utilization as the PE array size increases is due to the fragmentation, which also explains why 4×4 and 8×8 arrays have higher utilization than smaller arrays in PWC.

The next common layer type is DWC with stride 1, which can be mapped with 25%–68% PE utilization, substantially higher than that of DWC mapping with arbitrary stride. One factor that contributes to the lower PE utilization in DWC mappings is the initial and final cycles due to memory access and the prologue phase. These overheads increase in proportion to N_r or N_c, which explains the rather steep reduction in utilization for larger arrays. Finally, DWC mappings with arbitrary stride yield 15%–50% PE utilization when stride is 2. In this case, data is reused within a row only, which is the main reason for lower utilization.

G. Effect of Cross-Channel Optimization

To evaluate the effect of cross-channel optimization we use MobileNet V1. Fig. 16(a) shows the cycle counts for each layer without cross-channel optimization. The cycle count is broken down into two parts. Computation time is the number of cycles that it takes for a layer if DMA is ignored

Algorithm 1 Generate H-MEM Addresses for PWC

if $t_{cycle} < N_c$ **then**
  // generate load address
  $addr \leftarrow tid_r \cdot N_i + t_{cycle} + addr_{IFM}$
else
  // generate store address
  $addr \leftarrow tid_c \cdot N_c + tid_r \cdot N_c \cdot B_c + t_{cycle} - N_i + addr_{OFM}$
end if
// prepend bank index
return $(AID_r \ll N_a) \mid addr$

(i.e., input data is already in the on-chip memory, and output data need not be stored in the off-chip memory). DMA time is the difference between the actual layer latency (which is the total number of cycles for a layer) and computation time. We can see that DMA time is much higher in DWC layers, and particularly high in later DWC layers. This is because the height and width of IFM tensors decrease while the number of channels increases in later layers. In other words, DMAs get smaller in size but much more frequent as we go toward later layers, which can exacerbate the memory bottleneck problem. Fig. 16(b) shows the result after our cross-channel optimization. The computation time is not affected, nor is the PWC result, but only the DMA time in DWC layers is improved dramatically. As a result of cross-channel optimization, the latency of MobileNet V1 is reduced by 44.7% or its speed improved by 81%, demonstrating the effectiveness of our optimization.

VII. Conclusion

The fast evolution of DNNs poses both opportunities and challenges for hardware acceleration. Too specific to an application, and it can quickly become obsolete; too generic, and it may not be competitive enough. To solve this dilemma we proposed in this article to use CGRA with a small set of generic architecture extensions, which we have shown can greatly improve performance and efficiency for emerging lightweight DNN models. We also demonstrated that our proposed features are useful beyond DSC. As future work we plan to apply our architecture, NP-CGRA, to accelerating other machine learning algorithms and digital filters, many of which are based on matrix multiplication and convolution. Automatic generation of efficient code that exploits the new architectural features is also future work.

Appendix

A. AGU Algorithms

1) PWC: Algorithm 1 shows the algorithm to generate H-MEM addresses for PWC mapping. See Table II for the definition of symbols. The operators $\ll$ and $\mid$ denote bit-shift left and bit concatenation, respectively, as in the C programming language.

The algorithm is self-explanatory. Whether we generate load versus store address can be determined by comparing t_{cycle} with N_c. The constants addr_{IFM} and addr_{OFM} represent the start address of IFM and of OFM data, respectively.

2) DWC With Arbitrary Stride: Algorithm 2 gives the algorithm to generate the H-MEM addresses for our DWC mapping with arbitrary stride as explained in Section V-B. Note that the division operator is integer division (i.e., the result is truncated), and % denotes the modulo operator.

Algorithm 2 Generate H-MEM Addresses for DWC With Arbitrary Stride

$block_w \leftarrow S \cdot (B_c \cdot N_c - 1) + K$
if $t_{\text{wrap}} < K$ **then**
  // generate load bank index and address
  $over\_bank \leftarrow ((t_{wrap}/S) + AID_r)/N_r$
  //generate load bank index
  $bank_{number} \leftarrow ((t_{wrap}/S) + AID_r)\%N_r$
  $addr \leftarrow tid_r \cdot block_w \cdot S + tid_c \cdot S \cdot N_c + over\_bank \cdot block_w \cdot S + t_{wcycle} + (t_{wrap}\%S) \cdot block_w$
else
  // generate store bank index and address
  $bank_{number} \leftarrow AID_r$
  $addr \leftarrow AID_c \cdot N_c + tid_r \cdot N_c \cdot B_c + t_{wcycle} - 1 + addr_{OFM}$
end if
return $(bank_{number} \ll N_a) \mid addr$

Algorithm 3 Generate H-MEM Addresses for DWC With $S = 1$

$block_w \leftarrow 2 + B_c \cdot N_c$
$tile\_latency \leftarrow 1 + 2 \cdot N_c + K^2$
// generate bank index
if $t_{wrap} \geq K$ **then**
  $bank_{number} \leftarrow AID_r$
else
  $over\_bank \leftarrow (t_{wrap} + AID_r)/N_r$
  //generate load bank index
  $bank_{number} \leftarrow (t_{wrap} + AID_r)\%N_r$
end if
if $t_{wrap} \geq K$ **then**
  // generate store address
  $addr \leftarrow tid_c \cdot N_c + tid_r \cdot N_c \cdot B_c + N_c + cycle - tile\_latency + 1 + addr_{OFM}$
else
  $std\_addr \leftarrow tid_c \cdot N_c + tid_r \cdot block_w$
  // generate load address
  if $t_{wrap} = 0$ **then**
    // Kernel 0 row
    $addr \leftarrow std\_addr + t_{wcycle} + over\_bank * block_w$
  else
    if $t_{wrap}\%2 = 1$ **then**
      // Kernel odd row
      $addr \leftarrow std\_addr + K - 1 - t_{wcycle} + over\_bank \cdot block_w$
    else
      // Kernel even row
      $addr \leftarrow std\_addr + N_c - 1 + t_{wcycle} + over\_bank \cdot block_w$
    end if
  end if
end if
return $(bank_{number} \ll N_a) \mid addr$

3) DWC With Stride of One: Algorithm 3 gives the algorithm to generate the H-MEM addresses for DWC mapping with $S = 1$ as explained in Section V-C.

Acknowledgment

The EDA tool was supported by the IC Design Education Center (IDEC), Korea.

References

[1] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, "A survey of accelerator architectures for deep neural networks," *Engineering*, vol. 6, no. 3, pp. 264–274, 2020.
[2] H. Singh *et al.*, "Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 465–481, May 2000.

[3] B. Mei, M. Berekovic, and J.-Y. Mignolet, *ADRES & DRESC: Architecture and Compiler for Coarse-Grain Reconfigurable Processors*. Dordrecht, The Netherlands: Springer, 2007, pp. 255–297.
[4] M. Tanomoto, S. Takamaeda-Yamazaki, J. Yao, and Y. Nakashima, "A CGRA-based approach for accelerating convolutional neural networks," in *Proc. IEEE 9th MCSoc*, 2015, pp. 73–80.
[5] X. Fan, D. Wu, W. Cao, W. Luk, and L. Wang, "Stream processing dual-track CGRA for object inference," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 6, pp. 1098–1111, Jun. 2018.
[6] I. Bae, B. Harris, H. Min, and B. Egger, "Auto-tuning CNNs for coarse-grained reconfigurable array-based accelerators," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2301–2310, Nov. 2018.
[7] S. Dave, M. Balasubramanian, and A. Shrivastava, "RAMP: Resource-aware mapping for CGRAs," in *Proc. IEEE DAC*, 2018, pp. 1–6.
[8] Y. Kim *et al.*, "High throughput data mapping for coarse-grained reconfigurable architectures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 11, pp. 1599–1609, Nov. 2011.
[9] Y. Kim, J. Lee, A. Shrivastava, and Y. Paek, "Memory access optimization in compilation for coarse-grained reconfigurable architectures," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, no. 4, pp. 1–27, Oct. 2011.
[10] H. Lee, D. Nguyen, and J. Lee, "Optimizing stream program performance on cgra-based systems," in *Proc 52nd Annu. Design Autom. Conf. (DAC)*, 2015, pp. 1–110.
[11] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. ICML workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.
[12] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*.
[13] M. Sandler *et al.*, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proc. CVPR*, 2018, pp. 4510–4520.
[14] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recogni.*, 2018, pp. 6848–6856.
[15] M. Tan and Q. V. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," 2019, *arXiv:1905.11946*.
[16] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.
[17] Y.-H. Chen, T. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE J. Emerg. Sel. Topic Circuits Syst.*, vol. 9, no. 2, pp. 292–308, Jun. 2019.
[18] J. Lee and J. Lee, "NP-CGRA: Extending CGRAs for efficient processing of light-weight deep neural networks," in *Proc. Design Autom. Test Europe (DATE)*, Feb. 2021, pp. 1408–1413.
[19] S. Dave, M. Balasubramanian, and A. Shrivastava, "Ureca: Unified register file for CGRAs," in *Proc. Design Autom Test Eur. Conf. Exhibit. (DATE)*, 2018, pp. 1081–1086.
[20] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Regimap: Register-aware application mapping on coarse-grained reconfigurable architectures (CGRAS)," in *Proc. 50th Annu. Design Autom. Conf.*, 2013, pp. 1–10.
[21] L. Chen and T. Mitra, "Graph minor approach for application mapping on CGRAs," *ACM Trans. Reconfig. Technol. Syst.*, vol. 7, no. 3, Sep. 2014. [Online]. Available: https://doi.org/10.1145/2655242
[22] B. Mei *et al.*, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," *IEE Proc. Comput. Digit. Techn.*, vol. 150, no. 5, pp. 255–261, 2003.
[23] H. Park *et al.*, "Edge-centric modulo scheduling for coarse-grained reconfigurable architectures," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 166–176.
[24] S. A. Chin and J. H. Anderson, "An architecture-agnostic integer linear programming approach to cgra mapping," in *Proc. 55th Annu. Design Autom. Conf.*, 2018, pp. 1–6.
[25] Y. Kim, J. Lee, T. X. Mai, and Y. Paek, "Improving performance of nested loops on reconfigurable array processors," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 1–23, Jan. 2012.
[26] J. Lee, S. Seo, H. Lee, and H. U. Sim, "Flattening-based mapping of imperfect loop nests for CGRAs," in *Proc. Int. Conf. Hardw. Softw. Codesign Syst. Synth. (CODES+ISSS)*, Oct. 2014, pp. 1–10.
[27] K. Han, K. Choi, and J. Lee, "Compiling control-intensive loops for CGRAs with state-based full predication," in *Proc. Design Autom Test Eur. (DATE)*, Mar. 2013, pp. 1579–1582.
[28] Y. Jeong, S. Seo, and J. Lee, "Evaluator-executor transformation for efficient pipelining of loops with conditionals," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 1–23, Dec. 2013.
[29] S. Dave and A. Shrivastava. *CCF: A CGRA Compilation Framework*. [Online]. Available: https://github.com/MPSLab-ASU/ccf (Accessed: Nov. 1, 2021).
[30] T. Zhao, Y. Zhang, and K. Olukotun, "Serving recurrent neural networks efficiently with a spatial accelerator," in *Proc. Mach Learn. Syst.*, vol. 1, 2019, pp. 166–177.
[31] D. Suh *et al.*, "Design space exploration and implementation of a high performance and low area coarse grained reconfigurable processor," in *Proc. IEEE Int. Conf. FPT*, 2012, pp. 67–70.
[32] S. A. Chin *et al.*, "CGRA-ME: A unified framework for CGRA modelling and exploration," in *Proc. IEEE 28th Int. Conf. Appl. Spec. Syst. Archit. Process. (ASAP)*, 2017, pp. 184–189.
[33] Y. Park, H. Park, and S. Mahlke, "CGRA express: Accelerating execution using dynamic operation fusion," in *Proc. CASES*, 2009, pp. 271–280.
[34] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal, "Scalar operand networks: On-chip interconnect for ILP in partitioned architectures," in *Proc. IEEE HPCA*, 2003, pp. 341–353.
[35] J. Balfour, R. Harting, and W. Dally, "Operand registers and explicit operand forwarding," *IEEE Comput. Archit. Lett.*, vol. 8, no. 2, pp. 60–63, Feb. 2009.
[36] Y. Xiong *et al.*, "Accelerating deep neural network computation on a low power reconfigurable architecture," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, 2020, pp. 1–5.
[37] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Adv. NIPS*, 2015, pp. 1135–1143.
[38] R. Zhao *et al.*, "Building efficient deep neural networks with unitary group convolutions," in *Proc. CVPR*, 2019, pp. 11303–11312.
[39] S. A. Chin *et al.*, "CGRA-ME: A unified framework for CGRA modelling and exploration," in *Proc. ASAP*, 2017, pp. 184–189.
[40] A. Rahman, S. Oh, J. Lee, and K. Choi, "Design space exploration of FPGA accelerators for convolutional neural networks," in *Proc. DATE*, Mar. 2017, pp. 1147–1152.
[41] R. Balasubramonian *et al.*, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, pp. 1–25, 2017.

Jungi Lee received the bachelor's degree in mechatronics from Chungnam National University, Daejeon, South Korea, in 2019, and the master's degree in electrical engineering from the Intelligent Computing and Codesign Laboratory, Ulsan National Institute of Science and Technology, Ulsan, South Korea, in 2021.

His current research interests include deep neural network processing and reinforcement learning.

Jongeun Lee (Member, IEEE) received the B.S. and M.S. degrees in electrical engineering and the Ph.D. degree in electrical engineering and computer science from Seoul National University, Seoul, South Korea, in 1997, 1999, and 2004, respectively.

Since 2009, he has been a Faculty with the Ulsan National Institute of Science and Technology, Ulsan, South Korea, where he is a Professor of Electrical Engineering. His research interests include neural network processors, reconfigurable architectures, and compilers.