



저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

Master's Thesis

NP-CGRA: Extending CGRAs for Efficient
Processing of Light-weight Deep Neural Networks

Jungi Lee

Department of Electrical Engineering

Ulsan National Institute of Science and Technology

2021

NP-CGRA: Extending CGRAs for Efficient Processing of Light-weight Deep Neural Networks

Jungi Lee

Department of Electrical Engineering

Ulsan National Institute of Science and Technology

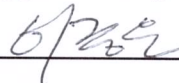
NP-CGRA: Extending CGRAs for Efficient Processing of Light-weight Deep Neural Networks

A thesis submitted to
Ulsan National Institute of Science and Technology
in partial fulfillment of the
requirements for the degree of
Master of Science

Jungi Lee

12/15/2020

Approved by



Advisor

Jongeun Lee

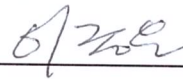
NP-CGRA: Extending CGRAs for Efficient Processing of Light-weight Deep Neural Networks

Jungi Lee

This certifies that the thesis of Jungi Lee is approved.

12/15/2020

Signature



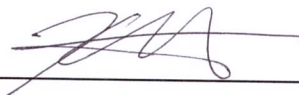
Advisor: Jongeun Lee

Signature



Woongki Baek

Signature



Kyuho Lee

Signature

Abstract

Coarse-grained reconfigurable architectures (CGRAs) can provide both high energy efficiency and flexibility, making them well-suited for machine learning applications. However previous work on CGRAs has a very limited support for deep neural networks (DNNs), especially for recent light-weight models such as depthwise separable convolution (DSC), which are an important workload for mobile environment. In this paper, we propose a set of architecture extensions and a mapping scheme to greatly enhance CGRA's performance for DSC kernels. Our experimental results using MobileNets demonstrate that our proposed CGRA enhancement can deliver 8~18 \times improvement in area-delay product 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 3D convolution with similar efficiency than previous work, demonstrating the effectiveness of our architectural features beyond DSC layers.

Contents

I	Introduction	1
II	Related Work	2
	2.1 Baseline CGRA	2
	2.2 CGRA Architecture Exploration	2
	2.3 DPU Optimization	2
III	NP-CGRA Architecture	3
	3.1 CGRA Performance Bottleneck Analysis	3
	3.2 Our Proposed Architecture Extension	3
	3.3 Instruction Format and Global Configuration	6
IV	Application Mapping for NP-CGRA: DWC Case	7
	4.1 Depthwise Convolution with Arbitrary Stride	7
	4.2 Depthwise Convolution with $S = 1$	8
V	Address Generation	11
	5.1 Pointwise Convolution	11
	5.2 Depthwise Convolution with Arbitrary Stride	14
	5.3 Depthwise Convolution with $S = 1$	15
	5.4 Performance Analysis	15

VI	Experiments	19
6.1	Experimental Setup	19
6.2	Depthwise Separable Convolution Results	19
6.3	Hardware Overhead Evaluation	20
6.4	Comparison with Previous Work Using MobileNet	22
6.5	AlexNet Convolution Layer Results	23
VII	Conclusion	24
	References	25
	Acknowledgements	27

List of Figures

1	Mapping PWC (or matrix mult.) to a 2×2 CGRA.	4
2	Proposed PE architecture (our extension shown in red).	5
3	Instruction definition.	6
4	Extended CGRA architecture.	7
5	Mapping DWC on a 2×2 CGRA ($K = 3, S = 2$).	9
6	Schedule and data movement for DWC ($S = 1$).	9
7	Data access patterns in DWC.	11
8	Mapping DWC with stride 1 on a 2×2 CGRA.	12
9	PWC IFM data in external memory and H-MEM.	13
10	DWC2 IFM data in external memory and H-MEM.	14
11	IFM data in external memory and in V-MEM for DWC (shown in red is a tile).	17
12	Area comparison.	22

List of Tables

1	Theoretical min latency (ms, sum of 7 DWC layers)	3
2	Parameters and variables	8
3	Performance analysis	15
4	NP-CGRA specifications	19
5	MobileNet DSC result	20
6	Comparison with previous CGRA and DPU implementations	21

I Introduction

Recently a number of DNN (Deep Neural Network) processing units, or DPUs, have been proposed, which can be classified as soft DPUs (implemented on an FPGA) and hard DPUs (fabricated into a chip) [1]. Hard DPUs such as TPU [2] can have higher performance and energy efficiency but lack flexibility, and may be difficult to support future application changes. Soft DPUs such as BrainWave [1] can be easily upgraded, but typically have much lower performance per cost and energy efficiency compared to hard DPUs. CGRAs can strike a balance between energy efficiency and flexibility, such as supporting new activation functions (e.g., leaky ReLU [3]) and skip connections. Also CGRAs can be utilized for other applications than DNNs.

Previous work on mapping DNNs to CGRAs includes new architectures [4,5] and a new compilation method [6], but they all target 3D convolution only (such as used in AlexNet [7]) [4–6]. However, for mobile applications, conventional 3D convolutions are superseded by light-weight models exploiting depthwise separable convolution (DSC) such as MobileNets [8,9] due to their significantly higher inference performance and greatly reduced model size and computation complexity. Depthwise separable convolution is realized as a combination of depthwise convolution (DWC) and pointwise convolution (PWC) layers. While PWC typically accounts for over 90% MAC operations, in terms of runtime DWC can account for up to 40% due to its low computation-to-data-transfer ratio and difficulty in mapping DWC. Hence it is important to provide optimized mapping for DWC as well as PWC.

In this paper we first present our analysis showing that CGRAs are not necessarily slower than hard DPUs when it comes to machine learning workload, if a right set of architectural features are provided. Based on the analysis, we present three generic architecture extensions for CGRAs—crossbar-style memory bus, dual-mode MAC (multiply-accumulate) unit, and operand reuse network—along with a mapping scheme that can greatly enhance CGRA’s performance for DSC kernels.

Our experimental results using MobileNet V1 and V2 [10,11] demonstrate that our proposed features can improve the efficiency of CGRA for DWC and PWC layers by 8 and 18 \times , respectively, in terms of area-delay product (ADP) over a compiler approach [12]. Moreover, though not explicitly optimized for, 3D convolution on our architecture is also quite efficient, generating competitive performance and ADP as a CGRA [5] explicitly optimized for machine learning algorithms including 3D convolution.

In this paper we make the following contributions. 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 we evaluate our proposed CGRA called *NP-CGRA* (*Neural Processing-CGRA*) for MobileNet models.

II Related Work

2.1 Baseline CGRA

While CGRA is a generic term encompassing many different architectures [4–6, 12–14], we consider ADRES-like CGRAs [14] as our baseline, which have been most extensively studied. The main datapath consists of a 2D array of PEs (Processing Elements) interconnected with a mesh-like network, plus local memory implemented as multi-banked SRAM blocks for high on-chip bandwidth. PEs can perform arithmetic/logic and memory operations though details vary. The PE operations and inter-PE connections are dynamically reconfigurable with no runtime overhead, thereby supporting pipelining of loops with II (Initiation Interval) greater than 1.

There are two kinds of memory operations on CGRAs in the literature: addressed vs streamed load-store. Addressed load-store [14] is more common among CGRA compilers as it supports random memory access, but requires explicit address computation (which uses PE cycles). Streamed load-store [13] requires dedicated AGUs (Address Generation Units), which support only a limited set of access patterns. In either case, it is possible for all connected PEs to simultaneously read a memory bus if needed.

2.2 CGRA Architecture Exploration

CGRA architecture exploration has been performed in [15], 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 cycle time impact of operation chaining. An extreme version of operation chaining has been proposed [16] 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 the literature [17, 18] generally refer to output-to-input networks.

2.3 DPU Optimization

DSC computation has been targeted by both hard DPUs [11] and soft DPUs, but not by CGRAs. We do not consider pruning [19] directly in this work, since DSC is already a form of sparsity at coarse granularity [20] while being much more amenable to hardware parallelization than fine-grained sparsity. We also do not consider aggressive quantization, but the width of datapath is trivially configurable at design time.

Table 1: Theoretical min latency (ms, sum of 7 DWC layers)

Architecture	Compute time	L1 transfer	Layer latency
CGRA baseline (4x4)	1.68	0.75~4.10	1.68~4.10
CGRA enhanced (8x8)	0.21	0.19	0.21
Eyeriss (168 PEs)	0.20	0.23	0.23

III NP-CGRA Architecture

3.1 CGRA Performance Bottleneck Analysis

Table 1 compares a baseline CGRA [6] with Eyeriss [10], a reference hard DPU, in terms of minimum theoretical latency, using 7 DWC layers from MobileNet V2, one from each bottleneck (we see similar results with other layers as well). The baseline CGRA has 4x4 PEs and runs at 500 MHz with 4-byte word size, and Eyeriss has 168 PEs and runs at 200 MHz with 2-byte word size.

We calculate minimum theoretical latency simply by the max of compute time (assuming 100% PE utilization), L1 transfer time (i.e., on-chip memory access latency), and external memory DMA (Direct Memory Access) 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 4 load-store units (one per row) are 100% utilized, and consider two scenarios: the least and most data reuse of IFM (Input Feature Map). For Eyeriss we assume 32 load-store units, and most data reuse.

Our result suggests that there is $\sim 8\times$ compute time difference between the baseline CGRA and Eyeriss DPU even if we assume 100% PE utilization, which may be harder for CGRA. The difference grows if CGRA fails to reuse IFM data optimally.

To fill the gap, we consider *CGRA enhanced*, which is the same CGRA but with 8x8 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, *one per row or column*, and the most data reuse scenario.

To summarize, our analysis suggests that CGRA is capable of delivering hard DPU-level performance, but needs a few major changes: single-cycle MAC, larger array size, at least $2\times$ on-chip memory bandwidth, and extremely high PE utilization.

3.2 Our Proposed Architecture Extension

Our driving application is pointwise convolution (PWC), which is also known as 1x1 convolution and algorithmically equivalent to matrix multiplication. While one can use a CGRA compiler

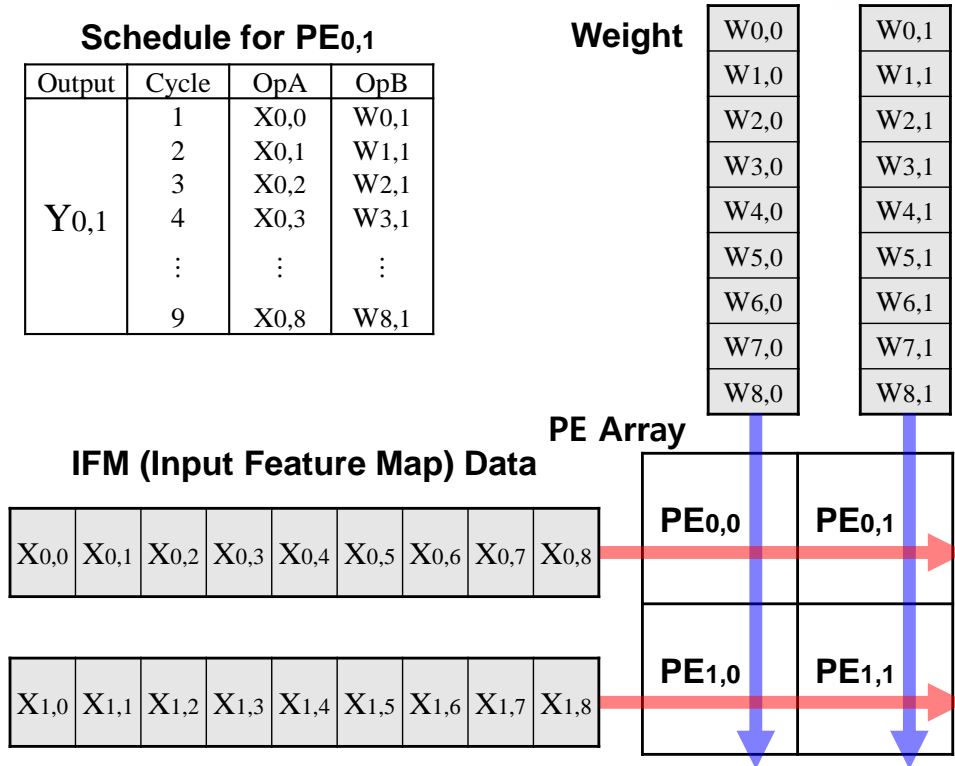


Figure 1: Mapping PWC (or matrix mult.) to a 2×2 CGRA.

(e.g., [21, 22]) to compile matrix multiplication for a CGRA, it would yield a vastly suboptimal schedule. In 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.

Crossbar-style Memory Bus

Fig. 1 illustrates our proposed mapping for a 2×2 CGRA, in which we use the first 2 rows of one source matrix (X) and the first 2 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 ($0 \leq i \leq 8$) through a horizontal bus (called H-bus), and similarly, PE_{0,0} and PE_{1,0} share $W(i, 0)$ through a vertical bus (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.

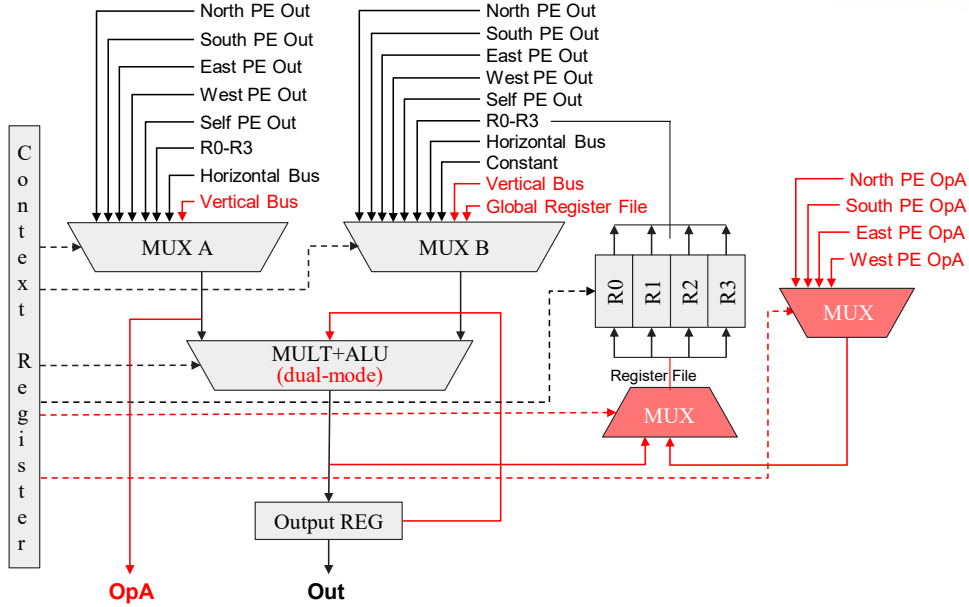


Figure 2: Proposed PE architecture (our extension shown in red).

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*. A detailed diagram of dual-mode MAC is omitted due to page limit, but it is straightforward to design one.

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 FIR filter example: $y_i \leftarrow w_0x_i + w_1x_{i+1} + w_2x_{i+2}$, 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 operand reuse network allows one of the source operands of a PE (i.e., the output of an input MUX) to be passed to neighbor PEs without affecting other computation that PEs may be doing, as illustrated in Fig. 2.

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

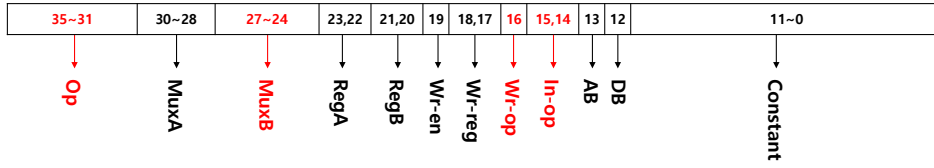


Figure 3: Instruction definition.

3.3 Instruction Format and Global Configuration

As the CGRA PE structure was changed, the bit width of the instruction increased. We use R type 32bit instruction from CCF framework instruction format [21]. If we don't consider R, P type of instruction, we can use this instruction by 31 bit. Fig. 3 shows our instruction format. Reg a, b indicates index of register file which applies to muxA and muxB. Wr-en means write on register file enable bit. Wr-reg means the index of register file when write on register file. Wr-op determines which data will be written on the register where from output register of itself or the neighbor output of PE muxA. In-op is the bit that determines which muxA of the neighbor PE is to be written on register file. AB is the bit for sending read requests to memory using output as address, and DB is the write request bit for storing output in memory. Our instruction format needs 36 bits. Op, muxB, wr-op requires an additional 1 bit each, and in-op requires an additional 2 bits. In the configuration memory, 4 bits need for index of global register file and 2 bits for H and V memory read requests. So bit of the configuration memory is calculated as $36 \times \text{Num of CGRA PEs} + 6$ format.

Other Changes

Fig. 4 shows our extended CGRA architecture. Our CGRA architecture has vertical memory(V-MEM), memory access module(MAM), global register file, etc. Also we change PE structure for dual mode mac and operand reuse network.

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 MAM which consists of 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 global register file (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.

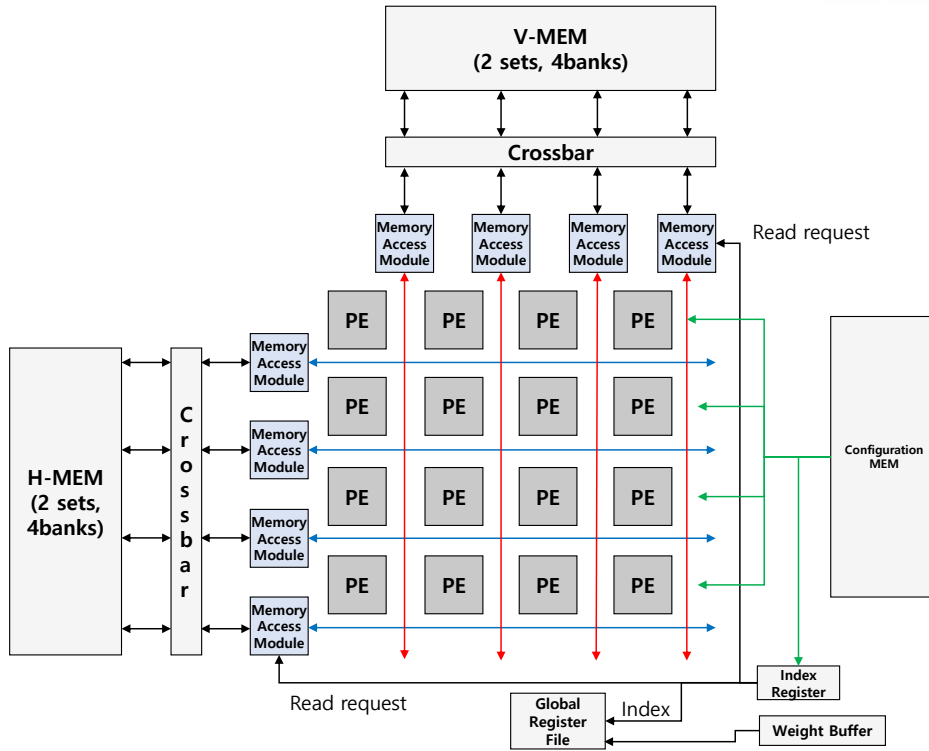


Figure 4: Extended CGRA architecture.

IV Application Mapping for NP-CGRA: DWC Case

We now present our application mapping for DWC kernels (PWC mapping is already outlined in section 3.2). Table 2 lists parameters and variables used in this section and section V. First we present a general method that works for any stride, then an optimized version for $S = 1$, which is most common. In the following, a *tile* refers to the amount of work (or corresponding data) that is done *simultaneously* by a CGRA, with its size determined by the CGRA size. *Block* is the amount of work that can be done using *local* data only. Block size is a multiple of tile size.

While in this paper 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.

4.1 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*.

Table 2: 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 \times 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

$$y_{0,0} = w_{0,0}x_{0,0} + w_{0,1}x_{0,1} + w_{0,2}x_{0,2} + w_{1,0}x_{1,0} + \dots + w_{2,2}x_{2,2}$$

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

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

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

The *input tile*, which is the set of IFM data needed to produce an output tile, is the gray-filled rectangle in Fig. 5a.

Fig. 5b 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 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.

4.2 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. 6 illustrates our solution. During the initial $N_c - 1$ cycles (called *prologue*), IFM data (the top-left $N_r \times (N_c - 1)$ submatrix) is loaded through H-busses into all PEs except the first

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}

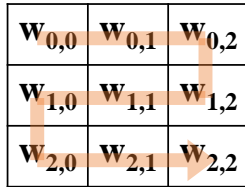
(a) IFM data (shown in gray is an input tile)

	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) Schedule

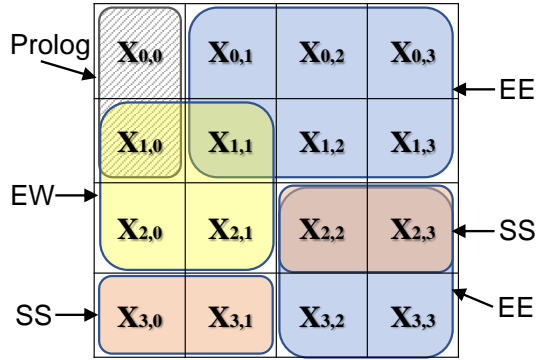
Figure 5: Mapping DWC on a 2×2 CGRA ($K = 3, S = 2$).



(a) Weight processing order

	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) Phase definition



(c) How IFM data is accessed

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) Data reuse and loading

Figure 6: Schedule and data movement for DWC ($S = 1$).

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 EE (Expand East) 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 SS (Shift South) phase. In the next $K - 1$ cycles, the PE array processes the remaining elements of the 2nd row of weight, which is similar to the EE phase except that we expand west, thus called EW (Expand West). 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. 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.

Fig. 7b 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. 7b 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. 8 illustrates data path between PE and memory and among PEs. The figure shows the detail data path of the previous example. Cycle 0 2 are prolog phase. Cycle 3, 4, 5, 10, and 11 are EE phase. Cycle 6, 9 are SS phase. Cycle 7 and 8 are EW phase. It shows that this schedule task 10 cycle except initial 2 and final 2 cycles.

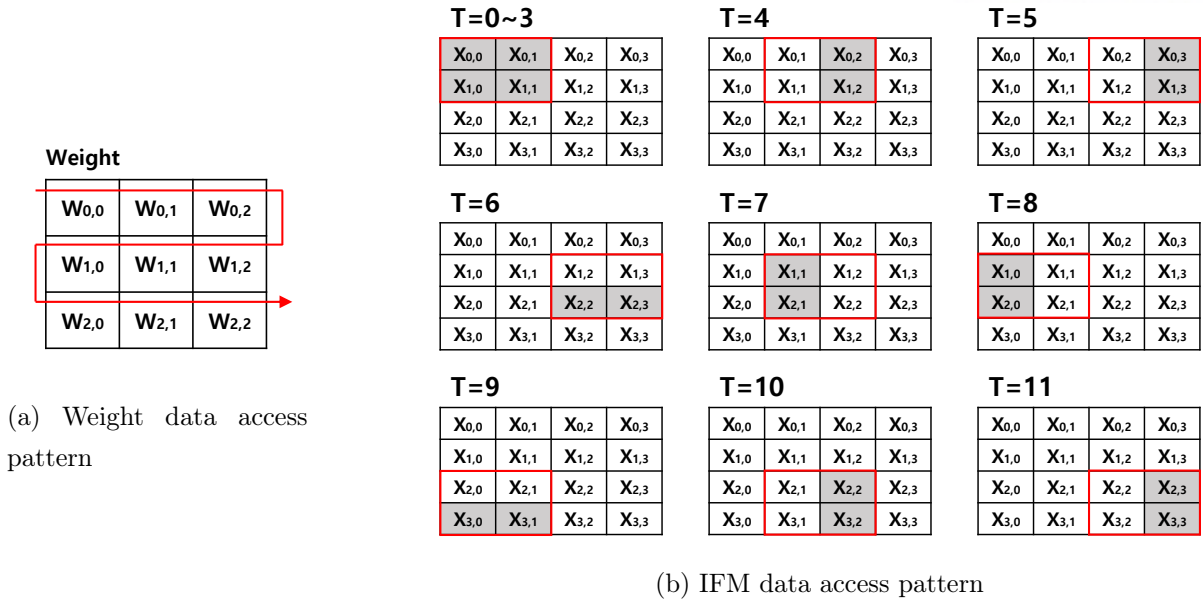


Figure 7: Data access patterns in DWC.

V Address Generation

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

5.1 Pointwise Convolution

Fig. 9a 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 the Fig. 9b. 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. The weight data are stored in the V-MEM local memory in a similar fashion. One important difference is that weight data needs to be partitioned along the column direction, which may require matrix transpose or reshaping. But since weight data is 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 (address generation units) in memory access units. For PWC, generating addresses to access V-MEM and H-MEM is straightforward. The V-MEM address is given as: $addr = (AID_c \ll N_a) | (tid_c \cdot N_i + t_{cycle})$, where \ll and $|$ have the same meaning as in C. This address can be easily computed by an AGU using t_{cycle} that 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, and is given as in Algorithm 1.

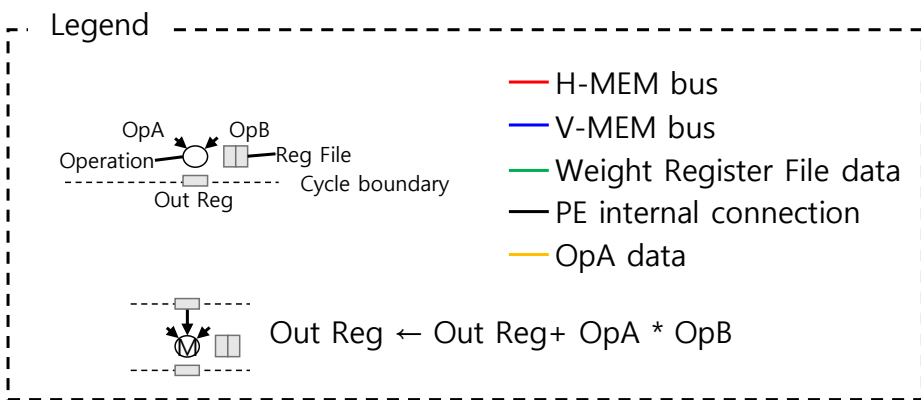
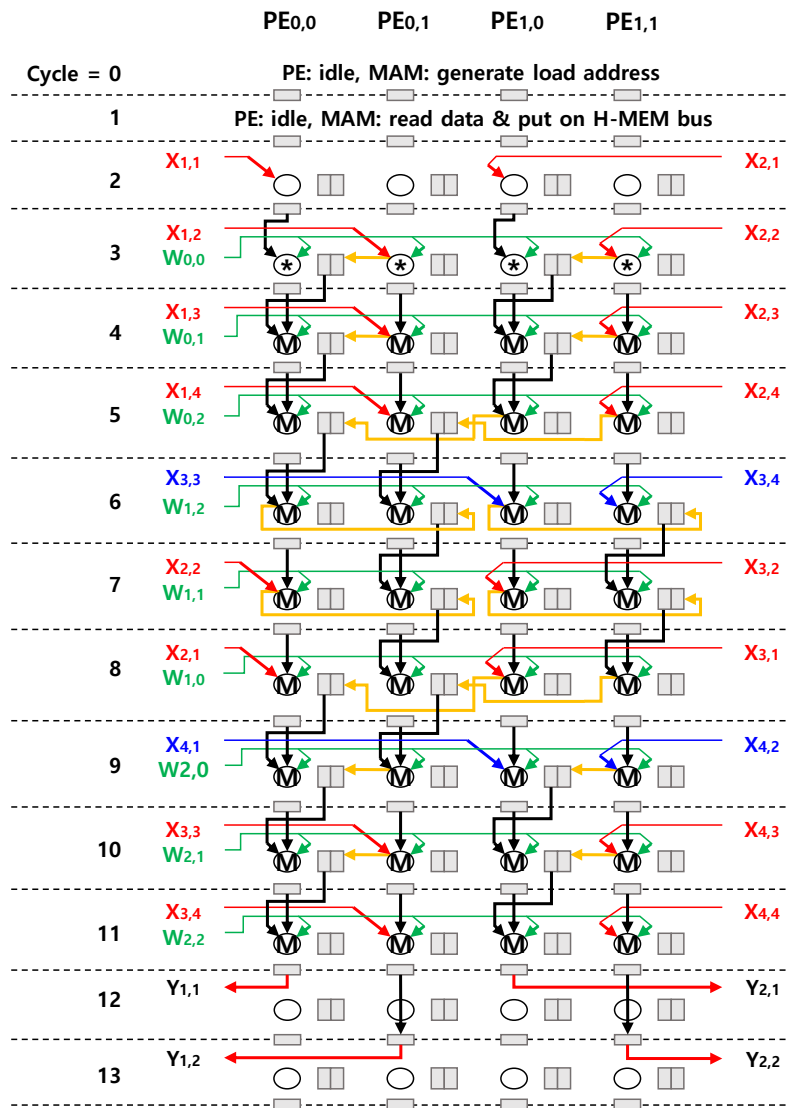
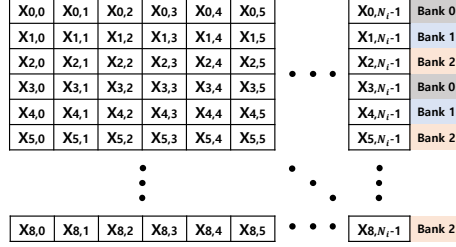
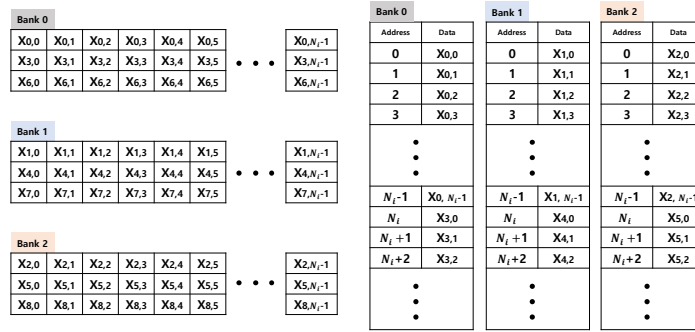


Figure 8: Mapping DWC with stride 1 on a 2×2 CGRA.



(a) Logical view of IFM data, and H-MEM bank assignment

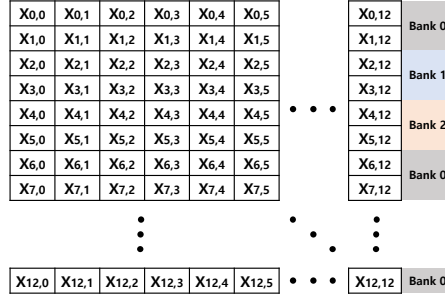


(b) Partitioning of IFM data into banks, and IFM data layout

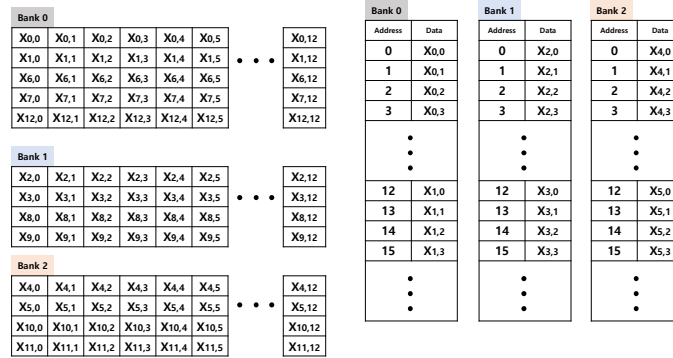
Figure 9: PWC IFM data in external memory and H-MEM.

Algorithm 1 Generate H-MEM addresses for PWC

- 1: **if** $t_{cycle} < N_c$ **then**
 - 2: *// generate load address*
 - 3: $addr \leftarrow tid_r \cdot N_i + t_{cycle} + addr_{IFM}$
 - 4: **else**
 - 5: *// generate store address*
 - 6: $addr \leftarrow tid_c \cdot N_c + tid_r \cdot N_c \cdot B_c + t_{cycle} - N_i + addr_{OFM}$
 - 7: **end if**
 - 8: *// prepend bank index*
 - 9: **return** $(AID_r \lll N_a) \mid addr$
-



(a) Logical view of IFM data, and H-MEM bank assignment



(b) Partitioning of IFM data into banks, and IFM data layout

Figure 10: DWC2 IFM data in external memory and H-MEM.

5.2 Depthwise Convolution with Arbitrary Stride

The data layout to support the proposed mapping is illustrated in Fig. 10. First the rows of the IFM data (which can be regarded as 2D since we consider only one channel at a time) are mapped to banks as Fig. 10a, 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. 10b).

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, which however does not cause a problem, since (i) there is a crossbar switch between the set of H-AGUs and the set of memory banks and (ii) 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 2^{nd} H-AGU always accesses an input row that is S -rows below what the 1^{st} H-AGU accesses, and so on.

The weight parameters needed by PEs are uniform vertically but not uniform horizontally, which suggests the use of V-busses. 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: $addr = (AID_c \ll N_a) \mid (t_{wcycle} - AID_c \cdot S + t_{wrap} \cdot K)$. Here t_{wrap} tracks which IFM row the CGRA is currently processing, or the *row* number of weight

Table 3: Performance analysis

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

parameters accessed by PEs. The H-MEM addresses are given in Algorithm 2.

Algorithm 2 Generate addresses for DWC2 H-MEM access

```

1:  $block_w \leftarrow S \cdot (B_c \cdot N_c - 1) + K$ 
2: if  $t_{wrap} < K$  then
3:   // generate load bank index and address
4:    $over\_bank \leftarrow ((t_{wrap}/S) + AID_r)/N_r$ 
5:   //generate load bank index
6:    $bank\_number \leftarrow ((t_{wrap}/S) + AID_r) \% N_r$ 
7:    $addr \leftarrow tid_r \cdot block_w \cdot S + tid_c \cdot S \cdot N_c + over\_bank \cdot block_w \cdot S + t_{wcycle} + (t_{wrap} \% S) \cdot block_w$ 
8: else
9:   // generate store bank index and address
10:   $bank\_number \leftarrow AID_r$ 
11:   $addr \leftarrow AID_c \cdot N_c + tid_r \cdot N_c \cdot B_c + t_{wcycle} - 1 + addr_{OFM}$ 
12: end if
13: return  $(bank\_number \ll N_a) | addr$ 

```

5.3 Depthwise Convolution with $S = 1$

DWC with $s = 1$ use same method for storing data in H-MEM. But this method can't be used for V-MEM because V-MEM stores only the data required for SS phases. The data layout to support the proposed mapping is illustrated in Fig. 11a. 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 Bank0. Fig. 11 shows data layout in external memory and partial IFM data partitioned into banks and layed out on V-MEM. Weight data for DWC with $s = 1$ is stored in GRF.

5.4 Performance Analysis

Table 3 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 6.1).

Algorithm 3 Generate addresses for DWC1 H-MEM access

```

1:  $block_w \leftarrow 2 + B_c \cdot N_c$ 
2:  $tile\_latency \leftarrow 1 + 2 \cdot N_c + K^2$ 
3: // generate bank index
4: if  $t_{wrap} \geq K$  then
5:    $bank_{number} \leftarrow AID_r$ 
6: else
7:    $over\_bank \leftarrow (t_{wrap} + AID_r) / N_r$ 
8:   //generate load bank index
9:    $bank_{number} \leftarrow (t_{wrap} + AID_r) \% N_r$ 
10: end if
11: if  $t_{wrap} \geq K$  then
12:   // generate store address
13:    $addr \leftarrow tid_c \cdot N_c + tid_r \cdot N_c \cdot B_c + N_c + cycle - tile\_latency + 1 + addr_{OFM}$ 
14: else
15:    $std\_addr \leftarrow tid_c \cdot N_c + tid_r \cdot block_w$ 
16:   // generate load address
17:   if  $t_{wrap} = 0$  then
18:     // Kernel 0 row
19:      $addr \leftarrow std\_addr + t_{wcycle} + over\_bank * block_w$ 
20:   else
21:     if  $t_{wrap} \% 2 = 1$  then
22:       // Kernel odd row
23:        $addr \leftarrow std\_addr + K - 1 - t_{wcycle} + over\_bank \cdot block_w$ 
24:     else
25:       // Kernel even row
26:        $addr \leftarrow std\_addr + N_c - 1 + t_{wcycle} + over\_bank \cdot block_w$ 
27:     end if
28:   end if
29: end if
30: return  $(bank_{number} \ll N_a) | addr$ 

```

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 _{0,9}	X _{0,10}
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 _{1,9}	X _{1,10}
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 _{2,9}	X _{2,10}
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 _{3,9}	X _{3,10}
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 _{4,9}	X _{4,10}
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 _{5,9}	X _{5,10}
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 _{6,9}	X _{6,10}
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 _{7,9}	X _{7,10}
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}	X _{8,9}	X _{8,10}
X _{9,0}	X _{9,1}	X _{9,2}	X _{9,3}	X _{9,4}	X _{9,5}	X _{9,6}	X _{9,7}	X _{9,8}	X _{9,9}	X _{9,10}
X _{10,0}	X _{10,1}	X _{10,2}	X _{10,3}	X _{10,4}	X _{10,5}	X _{10,6}	X _{10,7}	X _{10,8}	X _{10,9}	X _{10,10}

(a) Logical view of IFM data and V-MEM bank assignment

Bank 0	Bank 1	Bank 2	Bank 0	Bank 1	Bank 2	
X _{3,2}	X _{3,5}	X _{3,8}	Address	Data	Address	Data
X _{4,0}	X _{4,3}	X _{4,6}	0	X _{3,2}	0	X _{3,5}
X _{6,2}	X _{6,5}	X _{6,8}	1	X _{3,5}	1	X _{3,8}
X _{7,0}	X _{7,3}	X _{7,6}	2	X _{3,8}	2	X _{3,10}
X _{9,2}	X _{9,5}	X _{9,8}	3	X _{4,0}	3	X _{4,1}
X _{10,0}	X _{10,3}	X _{10,6}	4	X _{4,3}	4	X _{4,5}
			5	X _{4,6}	5	X _{4,8}
			•			
			•			
			•			

(b) Partial IFM data partitioned into banks, and layed out on V-MEM

Figure 11: IFM data in external memory and in V-MEM for DWC (shown in red is a tile).

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.

Further optimization

We use PWC mapping schedule after im2col to execute the standard convolution. Even if pwc mapping scheduling is handled quickly, the im2col overhead may slow processing speed. We can reduce the im2col overhead if we first process the data in the direction of the channel rather than running the im2col to process the data within the channel first.

In addition, our dwc algorithm has the problem that when processing data for multiple channels. It repeats processing 1 channel and loading the data rather than processes multiple

channel and loading the data. It takes more communication time than computation time when the height and width of IFM are small. So we will add a method of continuous processing of channel data when processing data for multiple channels with input of small height width. In the future, we will solve these two problems to increase efficiency by reducing network inference time despite the AGU overhead.

Table 4: 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)

VI Experiments

6.1 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 designed RTL 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 estimation we have synthesized RTL designs with Synopsys Design Compiler using Samsung 65 nm standard-cell library. The on-chip memory area is estimated using Cacti 7.0 [23].

Table 4 lists specifications 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 4 more bits than a baseline PE due to increased input MUX sizes (1 bit) and the operand reuse network’s MUXes (3 bits), and 8 more bits globally for GRF index and to control streamed load-store. Weight Buffer, which is optional, is set to hold 64 copies of GRF contents.

6.2 Depthwise Separable Convolution Results

We use the first three layers right after the first standard convolution (i.e., 3D convolution) layer in MobileNet V1 [8] (width multiplier 1, resolution 224). We compare three cases:

- **Baseline + CCF**: Baseline CGRA with CCF compiler [21]
- **Matmul DWC**: NP-CGRA + Matrix multiplication-based DWC

Table 5: MobileNet DSC result

Metric	Layer	CCF	Matmul DWC	Our mapping
Latency	PWC	78.91 (8.14)	3.72 (86.42)	3.72 (86.42)
(util)	DWC (S=1)	11.10 (8.14)	2.82 (16.04)	0.92 (49.00)
(ms,%)	DWC (S=2)	7.74 (5.83)	1.41 (16.01)	0.81 (28.00)
ADP	PWC	122.48	6.83	6.83
(mm ² ·ms)	DWC (S=1)	17.22	5.17	1.69
	DWC (S=2)	12.02	2.59	1.48

- **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 isn't taken into the account in this part.

Table 5 summarizes the result. The architectural factor is about $2 \times$, since our NP-CGRA has $2 \times$ 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 1 MUL and 3 ADD ops for every MAC operation (1 MUL, 1 ADD) in the program, which is due to address 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 $10 \times$ 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 $20 \times$ speed up and close to $18 \times$ ADP reduction for PWC over the baseline (our architecture has 18% larger total area including SRAM memory; synthesis result is discussed in section 6.3).

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 $1.75 \sim 3 \times$ 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 optimizing for the former case.

6.3 Hardware Overhead Evaluation

Fig. 12 compares the synthesis area of two 8×8 CGRAs at the target frequency of 500 MHz (timing met in both). The largest core increase comes from AGUs, which may be justified given

Table 6: Comparison with previous CGRA and DPU implementations

	Eyeriss [10]	Eyeriss-v2 [11]	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
Reported area (mm ²)	12.25	≥ 12.25	1.55 [†]	5.19	2.14
Converted area (65 nm, 16-bit) (mm ²)	12.25	≥ 24.50	1.55 [†]	7.25	2.14
MobileNet V1 (DSC runtime, ms)	-	0.78	-	-	4.01
MobileNet V2 (DSC runtime, ms)	-	-	-	-	18.06
MobileNet V1 ADP (DSC only, mm ² ·ms)	-	19.11	-	-	8.60
AlexNet (Conv. runtime, ms)	28.82	9.79	990	23.24	40.07 [‡]
AlexNet ADP (Conv. only, mm ² ·ms)	353.03	239.96	1536.68	168.59	87.28 [‡]

[†]Not reported in the paper, and assumed to be the area of the 4×4 baseline CGRA.

[‡]The ARM processor’s runtime is included in latency but its area is *not* included in ADP.

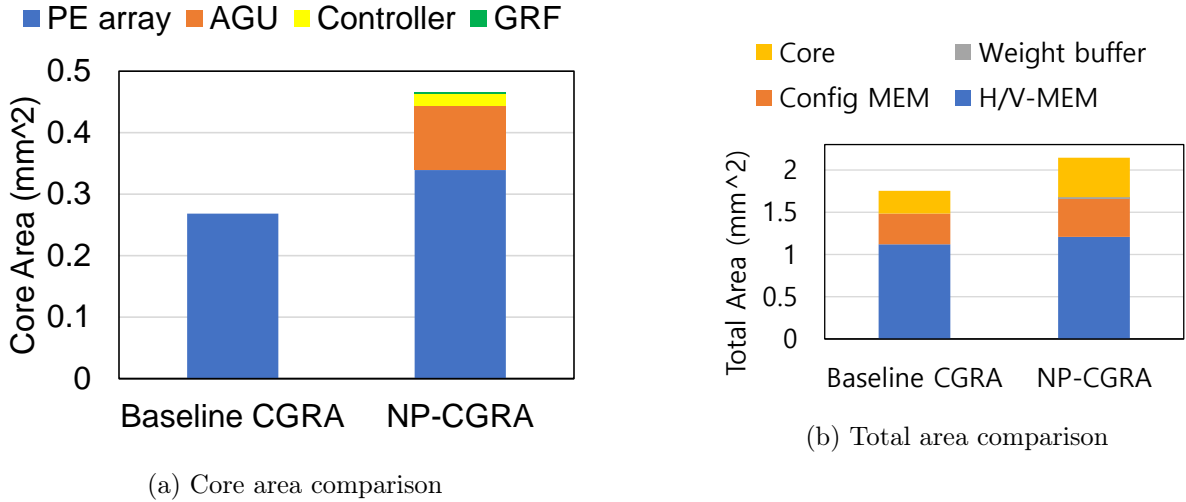


Figure 12: Area comparison.

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 homogeneous operation set, meaning all PEs support MUL and ADD operations). Not surprisingly, the total area is dominated by SRAM memory, which puts the overall area overhead of NP-CGRA at 22.2%.

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 ns (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 $2\times$ increase in computation throughput, the 34% increase in cycle time seems justifiable. On the other hand, MAC operations are not utilized by current CGRA compiler (e.g., CCF), which can limit applicability.

6.4 Comparison with Previous Work Using MobileNet

No previous CGRA reports MobileNet or DSC performance. A few MobileNet accelerators for FPGAs exist but no reported ASIC area makes direct comparison difficult. Eyeriss v2 [11] targets MobileNet V1 with width multiplier 0.5 and resolution 128, which we compare in Table 6. Eyeriss v2 has much more capable PEs than NP-CGRA, performing 2 MAC ops per cycle, which partially explains higher absolute performance compared with NP-CGRA. On the other hand, NP-CGRA is much smaller. While Eyeriss v2 reports gate count only, it appears larger than Eyeriss, so we assume Eyeriss v2 has the same area as Eyeriss. Also Eyeriss v2 uses 8-bit data width, we convert the area number to 16-bit equivalent by multiplying 2, which we believe is conservative. Overall, the NP-CGRA turns out to have lower ADP ($2.22\times$) though it is due to its faster clock speed.

6.5 AlexNet Convolution Layer Results

While 3D convolution is not explicitly optimized for by our architecture, 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 6). 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, 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 convolutional neural networks (CNNs). Eyeriss [10] and Eyeriss v2 [11] are hard DPUs optimized for CNNs.

To allow comparisons among different technologies and data widths, we convert the reported areas into 65 nm, 16 bit-equivalents, which are then multiplied with runtime to calculate ADP. As expected, the auto-tuning approach has the lowest performance and efficiency, attributed to poor scheduling. Eyeriss and Eyeriss v2 are among the fastest while SDT-CGRA is the most efficient in terms of ADP, which is again due to its faster clock speed. 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 ADP, demonstrating the efficacy of our extensions beyond DSC layers.

VII Conclusion

We presented a set of generic architecture extensions for CGRAs that can greatly improve performance and efficiency for light-weight DNN models. We have also demonstrated that our proposed features are useful beyond DSC layers, such as for 3D convolution. We plan to apply our 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 our proposed architectural features is left for future work.

References

- [1] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, “A configurable cloud-scale dnn processor for real-time ai,” in *2018 ACM/IEEE 45th ISCA*. IEEE, 2018, pp. 1–14.
- [2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *Proceedings of the 44th ISCA*, 2017, pp. 1–12.
- [3] A. L. Maas, A. Y. Hannun, and A. Y. Ng, “Rectifier nonlinearities improve neural network acoustic models,” in *ICML*, 2013.
- [4] M. Tanomoto, S. Takamaeda-Yamazaki, J. Yao, and Y. Nakashima, “A CGRA-based approach for accelerating convolutional neural networks,” in *2015 IEEE 9th MCSoc*, 2015.
- [5] X. Fan, D. Wu, W. Cao, W. Luk, and L. Wang, “Stream processing dual-track CGRA for object inference,” *IEEE Trans. VLSI*, vol. 26, no. 6, pp. 1098–1111, 2018.
- [6] I. Bae, B. Harris, H. Min, and B. Egger, “Auto-tuning CNNs for coarse-grained reconfigurable array-based accelerators,” *IEEE TCAD*, vol. 37, no. 11, 2018.
- [7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in NIPS*, 2012, pp. 1097–1105.
- [8] 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.
- [9] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.
- [10] Y.-H. 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, 2016.

- [11] Y. Chen, T. Yang, J. Emer, and V. Sze, “Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices,” *IEEE Journal on Emerging and Selected Topics in CAS*, vol. 9, no. 2, pp. 292–308, 2019.
- [12] S. Dave, M. Balasubramanian, and A. Shrivastava, “Ramp: resource-aware mapping for cgras,” in *2018 55th ACM/ESDA/IEEE DAC*. IEEE, 2018, pp. 1–6.
- [13] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, “Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications,” *IEEE transactions on computers*, vol. 49, no. 5, pp. 465–481, 2000.
- [14] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, “Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix,” in *International Conference on FPL*. Springer, 2003, pp. 61–70.
- [15] D. Suh, K. Kwon, S. Kim, S. Ryu, and J. Kim, “Design space exploration and implementation of a high performance and low area coarse grained reconfigurable processor,” in *2012 International Conference on FPT*. IEEE, 2012, pp. 67–70.
- [16] Y. Park, H. Park, and S. Mahlke, “Cgra express: accelerating execution using dynamic operation fusion,” in *CASES*, 2009, pp. 271–280.
- [17] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal, “Scalar operand networks: On-chip interconnect for ilp in partitioned architectures,” in *HPCA*. IEEE, 2003.
- [18] J. Balfour, R. Harting, and W. Dally, “Operand registers and explicit operand forwarding,” *IEEE Computer Architecture Letters*, vol. 8, no. 2, pp. 60–63, 2009.
- [19] S. Han, J. Pool, J. Tran, and W. Dally, “Learning both weights and connections for efficient neural network,” in *Advances in NIPS*, 2015, pp. 1135–1143.
- [20] R. Zhao, Y. Hu, J. Dotzel, C. D. Sa, , and Z. Zhang, “Building efficient deep neural networks with unitary group convolutions,” in *CVPR*, 2019.
- [21] S. Dave and A. Shrivastava, “Ccf: A cgra compilation framework.”
- [22] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, “Cgra-me: A unified framework for cgra modelling and exploration,” in *ASAP*, 2017, pp. 184–189.
- [23] R. Balasubramanian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, “Cacti 7: New tools for interconnect exploration in innovative off-chip memories,” *ACM TCAO*, vol. 14, no. 2, pp. 1–25, 2017.

Acknowledgements

Thank you to the advisor professor for helping me to write this thesis, and to all of you who came to me when I was in trouble.

