EFFICIENT PIPELINED RERAM-BASED PROCESSING-IN-MEMORY ARCHITECTURE FOR CONVOLUTIONAL NEURAL NETWORK INFERENCE

A Dissertation
Presented to
The Academic Faculty

By

Sho Ko

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

May 2020

EFFICIENT PIPELINED RERAM-BASED PROCESSING-IN-MEMORY ARCHITECTURE FOR CONVOLUTIONAL NEURAL NETWORK INFERENCE

To my parents for their love and support

ACKNOWLEDGEMENTS

I would like to thank my advisor Prof. Shimeng Yu and my instructors Prof. Tushar Krishna and Prof. Arijit Raychowdhury for their guidance and patience throughout my MS research work at Georgia Tech. Most importantly, none of these would be possible without the love and support from my parents.

TABLE OF CONTENTS

LIST OF FIGURES

SUMMARY

This research work presents a design of an analog ReRAM-based PIM (processing-in-memory) architecture for fast and efficient CNN (convolutional neural network) inference. For the overall architecture, we use the basic hardware hierarchy such as node, tile, core, and subarray. On the top of that, we design intra-layer pipelining, inter-layer pipelining, and batch pipelining to further exploit parallelism in the architecture and increase overall throughput for the inference of an input image stream. Our simulator also optimizes the performance of the NoC (network-on-chip) using SMART (single-cycle multi-hop asynchronous repeated traversal) flow control. Finally, we experiment with weight replications for different CNN layers and report throughput, energy efficiency, and speedup of VGG (A-E) for large-scale data set ImageNet.

CHAPTER 1
BACKGROUND AND INTRODUCTION

This chapter gives the detailed background knowledge needed to understand the design in the thesis.

1.1 Convolutional Neural Network Algorithm

In this section, we present the exact convolutional neural network (CNN) algorithm implemented in our design. A CNN has two phases: training and inference. The weights in a CNN are initialized with random values. Then the training process will update and refine the weights to a specific data set. Finally, a well-trained a CNN can be used for inference of new images. Typically, the training process has much more power and time consumption than the inference process because training requires forward propagation, back propagation, and weight update while inference only requires forward propagation. However, a CNN needs to be trained only once, and then it can be used for inference for numerous times. For the training process of a CNN, please refer to [1] for the detailed mathematical deduction. Typically, a CNN can be successfully trained on a GPU within several days.

For the inference process of a CNN, it consists of multiple layers with three basic types: convolution layers, pooling layers, and fully-connected layers, as shown in Fig. 1.1. The following subsections focus on explaining the details of each layer in a CNN.

1.1.1 Convolution Layers

The convolution layer is the signature of a CNN. It catches the edges of an image in order to achieve high accuracy of image classification. However, it consumes the largest portion of time and power for the whole inference process.

Common 2D convolution algorithms include single kernel single channel (SKSC), single kernel multiple channel (SKMC), and multiple kernel multiple channel (MKMC). MKMC is widely used in most CNNs. In order to present how MKMC without batching works, we define input feature map (IFM) to be I, kernel to be K, and output feature map (OFM) to be O. I is a 3D matrix with dimensions $c(channel) \times h(height) \times w(width)$. K is a 4D matrix with dimensions $n(kernel) \times c(channel) \times l(length) \times l(length)$. O is a 3D matrix with dimensions $c(kernel) \times h(height) \times w(width)$. Note that we have to pad $l-1$ columns of zeros to the right of I and $l-1$ rows of zeros to the bottom of I in order to get the correct dimensions of O. SKSC is a simple dot product between one channel of the image and one channel of one kernel, which is defined as

$$SKSC(I_i, K_{j,i})[a,b] = conv(I_i, K_{j,i}) = \sum_{m=0}^{l-1} \sum_{n=0}^{l-1} I_{i,a+m,b+n} \times K_{j,i,m,n} \tag{1.1}$$

where $i \in [0,c)$, $j \in [0,n)$, $a \in [0,h)$, and $b \in [0,w)$. SKMC is calculated by summing the result of SKSC of every corresponding channel of the image and one kernel, which is defined as

$$SKMC(I, K_j) = \sum_{i=0}^{c-1} conv(I_i, K_{j,i}) \tag{1.2}$$

where $j \in [0,n)$. MKMC is computed by concatenating the result of SKMC of the image

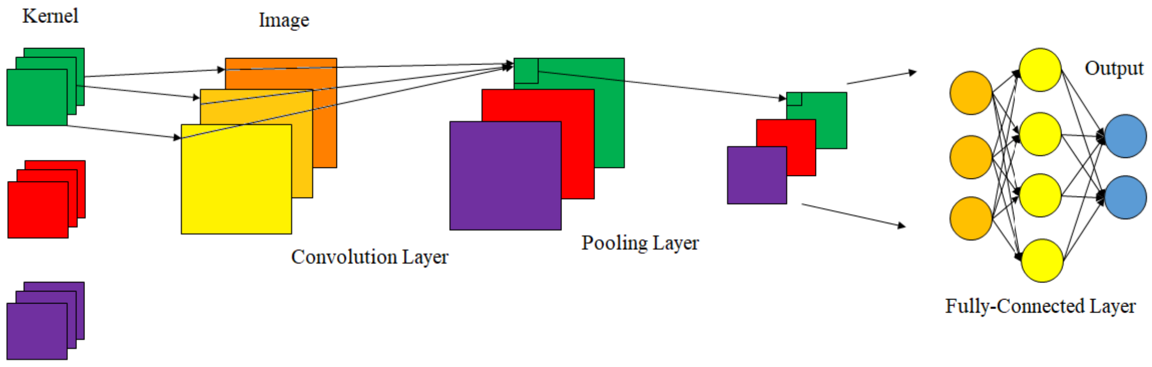


Figure 1.1: Convolutional Neural Network.

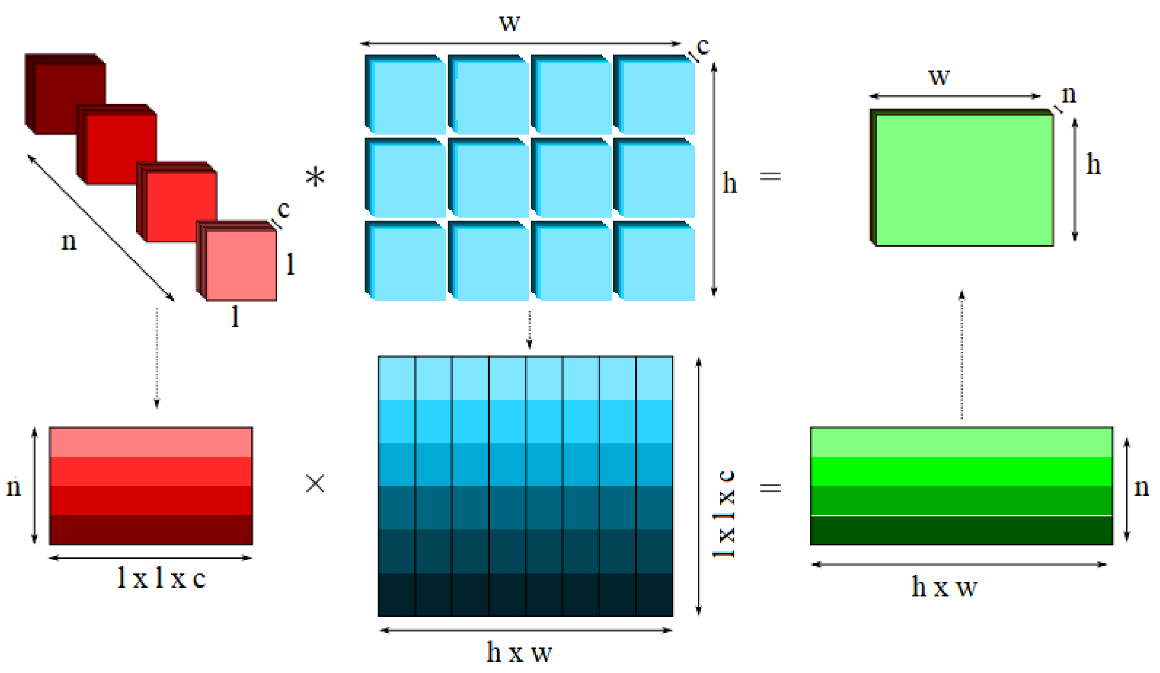


Figure 1.2: Convolution Layer Unroll to Matrix Multiplication [2].

and every kernel, which is defined as

$$O = MKMC(I, K) = SKMC(I, K_0)| \cdots |SKMC(I, K_{n-1}) \tag{1.3}$$

where | represents concatenation.

Typically, MKMC is calculated by unrolling each kernel into a row vector in the kernel matrix and corresponding image pixels to a column vector in the image matrix, as shown in Fig. 1.2, which is redacted from [2]. Then multiplying the two matrices gives the result. Note that it takes $h \times w$ logical cycles to pass in the entire image and get the full results.

1.1.2 Pooling Layer

The pooling layer is used to reduce the feature map by using one pixel to represent a relatively small grid (usually 2×2). In this way, it alleviates the amount of computations for the following layer. Max pooling (MP) is used most often with great performance.

Using the terms from 1.1.1, MP of size 2×2 is defined as

$$MP(I_i)[a,b] = max(I_{i,a,b}, I_{i,a+1,b}, I_{i,a,b+1}, I_{i,a+1,b+1}) \tag{1.4}$$

where $i \in [0,c)$, $a \in [0,h)$, and $b \in [0,w)$.

From a hardware perspective, pooling layers effectively reduces the amount of on-chip storage required for the inference of a CNN. However, pooling layers degrade the performance of inter-layer pipelining because the next layer has to wait for the pooled result which comes from different columns of the current feature map, which introduces extra pipeline bubbles, increases latency, and decreases throughput. More hardware implementation details will be described in Chapter 2.

1.1.3 Fully-Connected Layer

After all convolution layers and pooling layers are finished, the final OFM will reshaped into a vector and fed into a series of full-connected layers (FC) , also named as multi-layer perceptron (MLP). The series of FCs can reduce the large initial vector into the final small vector. After normalization, each number in the final vector represents the probability of an input image belonging to that particular class. For example, a CNN for handwritten digits classification task would have 10 classes, so the final vector is of size 10. In order to explain how FC works, we define input vector to be I, weight matrix to be W, and ouput vector to be O. I is a 1D matrix with dimensions n. W is a 2D matrix with dimensions $m \times n$. O is a 1D matrix with dimensions m. FC is defined as

$$O_a = FC(I)[a] = \sum_{i=0}^{n-1} I_i \times W_{a,i} \tag{1.5}$$

where $a \in [0,m)$.

1.1.4 Sigmoid Function

At the end of each convolution layer and fully connected layer, we need to pass the output through a sigmoid function (SIG), which is defined as

$$SIG(x) = \frac{1}{1 + e^{-x}} \tag{1.6}$$

SIG functions like a lubricant for a CNN because it maps the output to the range between 0 and 1. It is also differentiable so it's compatible with the training process.

1.2 Digital Accelerators

Since CNNs are generally computation intensive and power hungry, the current research community focuses on building efficient hardware platforms to accelerate deep neural networks. In this section, we discuss some digital accelerators in both cloud and edge for CNNs.

1.2.1 Digital Accelerators in Cloud

Typically, CNNs are trained in the cloud with high accuracy. Some companies use a cluster of CPUs and GPUs to do distributed training efficiently. For example, Google Cat uses 16000 CPU cores. Microsoft ResNet uses 8 GPU cores. In general, GPU can achieve around 0.1 TOPS/W power efficiency and dominates the market for AI hardware. More recently, Google invented TPU with 256×256 systolic array of MAC units, clocked at 700 MHz [3]. Its peak throughput can reach $700M \times 256 \times 256 \times 2 = 92$ TOPS.

1.2.2 Digital Accelerators in Edge

After CNNs are trained in the cloud, the weights are sent back to the edge devices for real-time inference. Although inference is less computation intensive and power hungry than training, it still requires tons of MAC computations. For a conventional digital inference engine, both weights and feature maps need to be stored in the global buffer SRAM or even off-chip DRAM. The processing engine (PE) needs to constantly access the SRAM or even DRAM to read and write the data. Therefore, the system-level performance is limited by the I/O bandwidth. More recently, some processing-in-memory (PIM) architectures are proposed. They fix the weights in each PE and only move feature maps around, unlike the conventional architectures which move both weights and feature maps around. Since moving data costs time and energy, these PIM architectures has higher throughput and power efficiency. The on-chip memory used to be SRAM. But now with the emerging memory technologies, resistance-based non-volatile memories can also be used to do on-chip analog PIM. More details will be discussed in Section 1.3 and 1.4.

1.3 ReRAM Device and Circuit

In this section, we present ReRAM from device and circuit perspectives.

1.3.1 ReRAM Device

From the device perspective, resistance-based emerging non-volatile memories (eNVM) have become more and more mature and manufacturable. Memories such as resistive random access memory (ReRAM) [4], phase change memory (PCM) [5], and spin-transfer torque magnetic random access memory (STT-MRAM) [6] start to gain more and more popularity. These eNVMs have much smaller cell size than SRAM. They can achieve multiple bits per cell (MLC). Therefore, they can map the entire weights on-chip at once and eliminate off-chip access. They are also non-volatile and CMOS-process compatible. Their access speed is within 10 ns, which is in the same magnitude as SRAM.

1.3.2 ReRAM Circuit

From the circuit perspective, 2D ReRAM is a grid structure consisting of ReRAM cells, as shown in Fig. 1.3. Such design can exploit the analog characteristics of ReRAM to perform fast and energy-efficient matrix multiplication and convolution. Vector-matrix multiplication can be easily calculated using ReRAM, because of two basic electrical theorems, Ohm's law and Kirchhoff's current law. Ohm's law states that the current through a resistor is equal to the voltage across the resistor divided by the resistance of the resistor ($I = V/R$), which is also equal to the voltage across the resistor multiplied by the conductance of the resistor ($I = VG$). This law makes performing analog floating-point multiplication possible. Kirchhoff's current law states that the total current output is equal to the sum of all input current for a node in the circuit. This law makes performing analog floating-point addition possible. Vector-matrix multiplication can be mapped to ReRAM in the following three steps, as shown in Fig. 1.3. First, the digital input is converted to analog signals by digital-to-analog converters (DACs) and then mapped to the voltage on horizontal bit lines (WLs); Second, the weight matrix is quantized and then mapped to the conductance

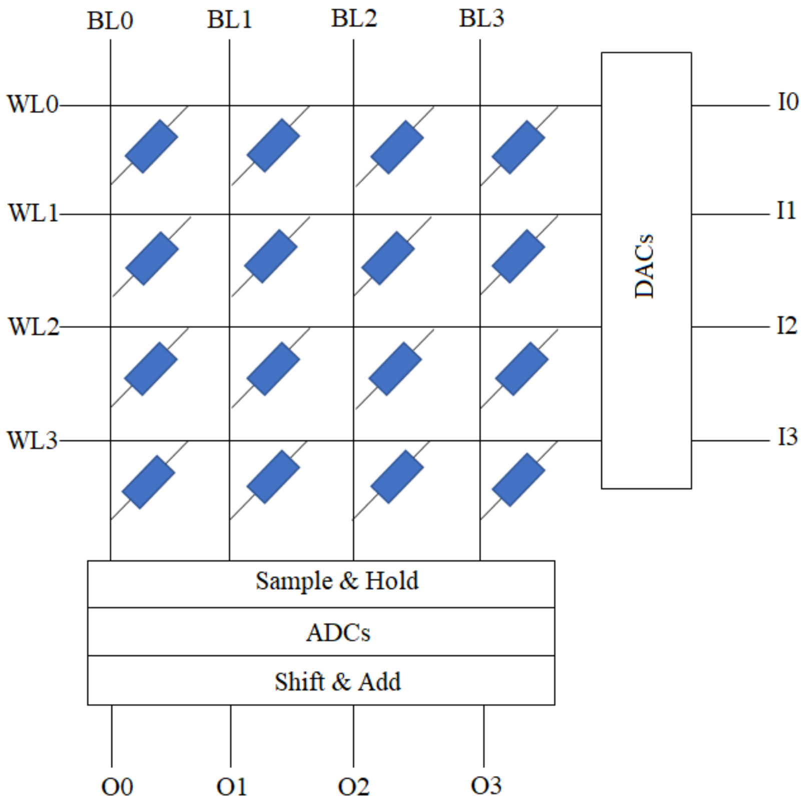


Figure 1.3: ReRAM Array with Peripheral Circuits.

of ReRAM cells; Third, the analog output signals are read from the current on the vertical bit lines (BLs), stored in sample hold units, converted to digital output by analog-to-digital converters (ADCs), and some columns are shifted and added together to produce the final results.

1.4 Analog Processing-In-Memory Accelerators

Recently, several ReRAM-based PIM architectures have been presented for CNN inference, such as PUMA [7], ISAAC [8], and PRIME [9].

PUMA creates its own compiler and domain-specific instruction set architecture (ISA) to make the architecture more general-purpose, programmable, and reconfigurable. It's a spatial architecture in which each tile is executing its own ISAs simultaneously with all other tiles. It uses a state machine to synchronize among different cores, it has a large synchronization overhead. In addition, the penalty of ISA, instruction decoder, and instruction memory is also large if the workloads are only CNNs.

Unlike PUMA, ISAAC and PRIME are ASICs specifically for CNN inference. PRIME is slightly different from ISAAC in the sense that PRIME stores positive and negative weights in separate subarrays while ISAAC stores them in the same subarray and uses a small trick to differentiate based on the MSB of a positive 2's complementary number is 0 while the MSB of a negative 2's complementary number is 1. Therefore, PRIME comes with more area and power penalty, which leads to less area and power efficiency.

In addition, there are also proposed architectures for CNN training such as PipeLayer [10] and TIME [11]. These architectures usually use an mixed-signal approach to combine ReRAM analog in-memory computing and conventional digital logic for the computation intensive backpropagation and weight update, which is sometimes inefficient for fast and efficient inference.

CHAPTER 2
OVERALL ARCHITECTURE, PIPELINING, AND NOC

This research work presents a design of an analog ReRAM-based PIM (processing-in-memory) architecture for fast and efficient CNN (convolutional neural network) inference. For the overall architecture, we use the basic hardware hierarchy such as node, tile, core, and subarray. On the top of that, we design intra-layer pipelining, inter-layer pipelining, and batch pipelining to further exploit parallelism in the architecture and increase overall throughput for the inference of an input image stream. Our simulator also optimizes the performance of the NoC (network-on-chip) using SMART (single-cycle multi-hop asynchronous repeated traversal) flow control. Finally, we experiment with weight replications for different CNN layers and report throughput, energy efficiency, and speedup of VGG (A-E) for large-scale data set ImageNet.

2.1 Overall Architecture

The overall chip, also called a node, as shown in Fig. 2.1. The node is composed of $16 \times 20 = 320$ tiles. Each tile has a outer associated with it. The routers form a mesh structure. Within each tile, there are 12 cores, a local memory of 64KB eDRAM, a shift and add unit, an output register of 2KB eDRAM, two sigmoid units, and a max pooling unit. Within each core, there are eight ReRAM subarrays of size 128×128, 128×8 1-bit DACs, 128×8 sample hold units, eight ADCs with 8-bit resolution, four shift add units, an input register of 2KB eDRAM, and an output register of 2KB eDRAM. There are buses within each tile and each core. The number of each component is designed so that there is no structural hazard during run time. For our design, the weights and feature maps are both fixed 16 bits. Lots of previous research has shown that 16 bits are accurate enough for CNN inference. We conservatively assume 2-bit MLC for each ReRAM cell. Therefore,

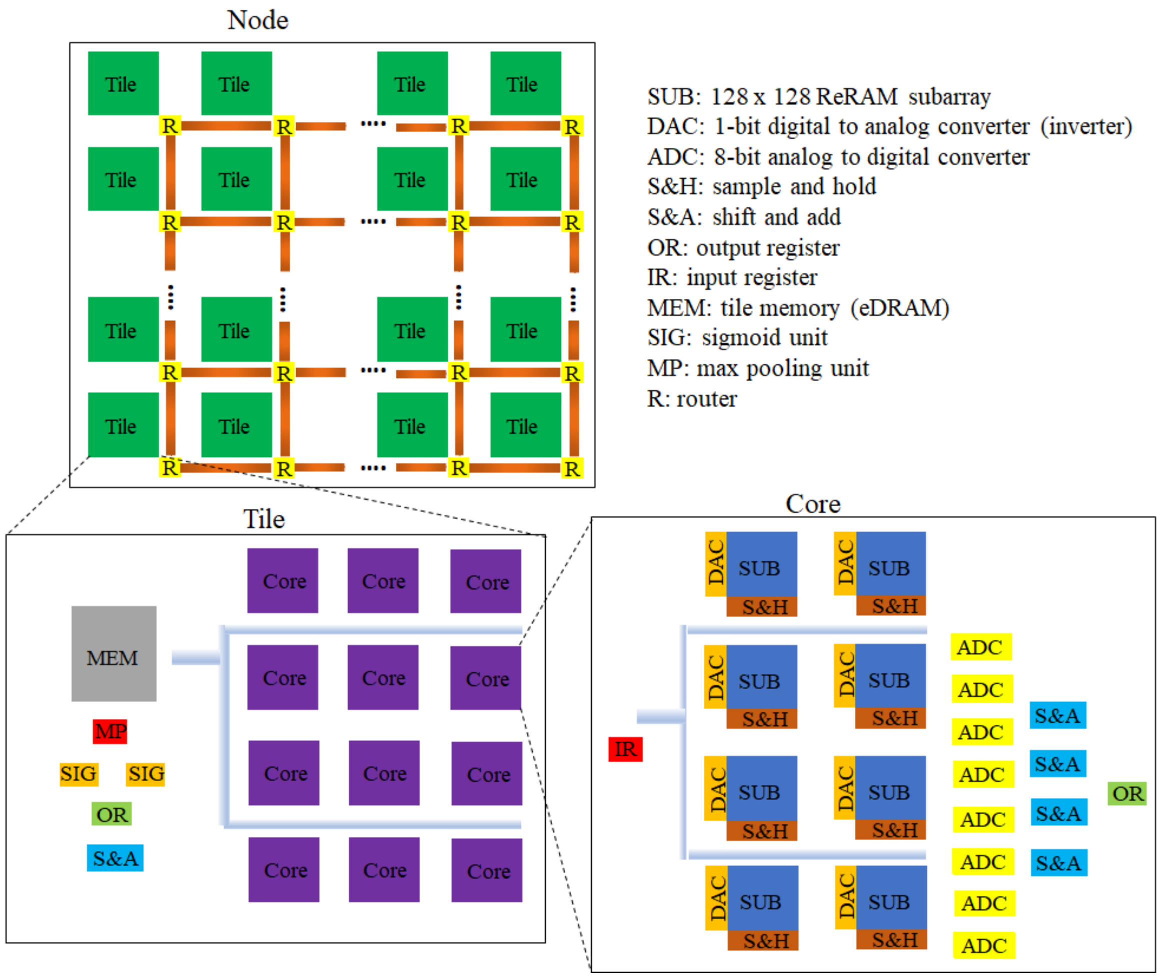
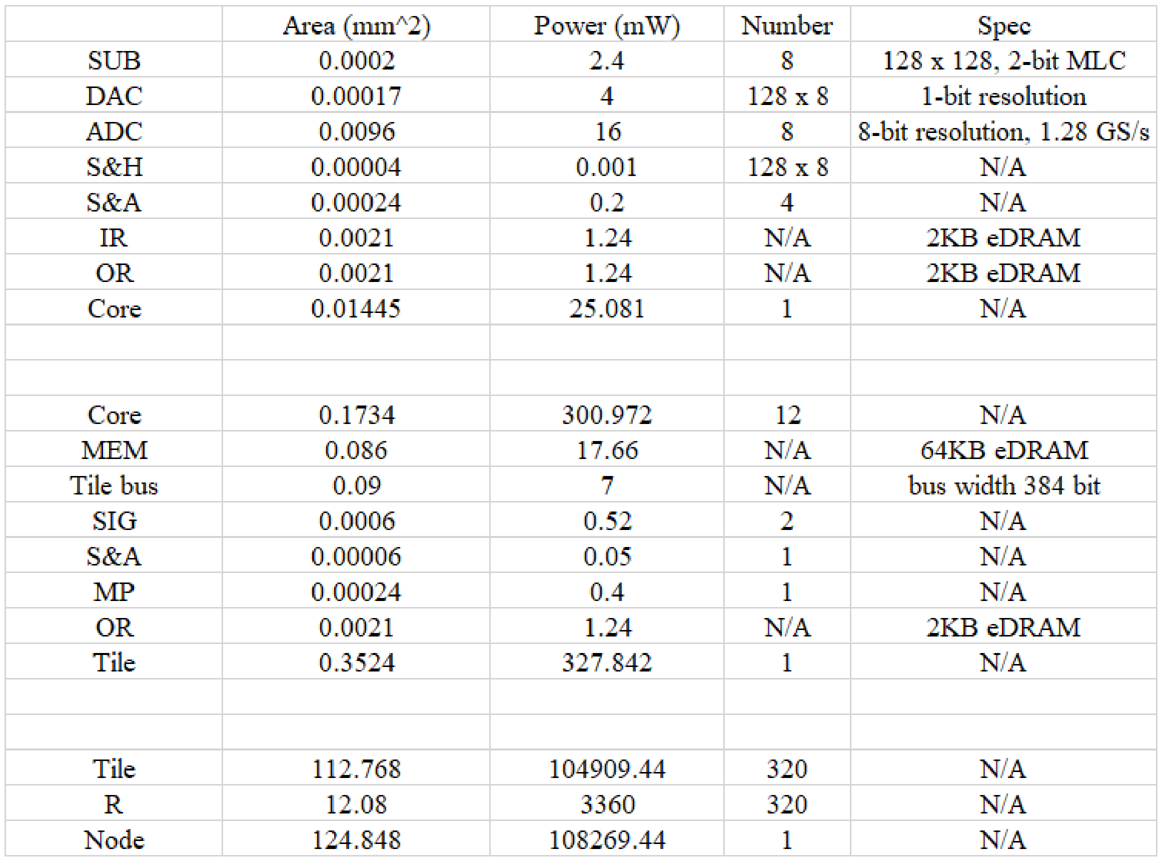


Figure 2.1: Overall Architecture of a Node.

we need eight cells across eight different columns to encode all of them. In addition, our DAC is of 1-bit resolution, which is trivial. Since 16-bit DAC has too much noise and takes too much area and power, we choose to pass in the 16-bit IFM bit by bit sequentially in 16 cycles. Therefore, we only need 1-bit DACs. Note that since we partition the weight spatially across different columns and we also partition the input temporally within the same column, the shift and add unit after the ADC will be necessary to produce the correct final results.

Fig. 2.2 shows the power and area of each individual component, we gather the data from PUMA [7] and ISAAC [8], both of which are in 32 nm CMOS technology node. Note that this table shows the power consumption when the component is functioning. The node



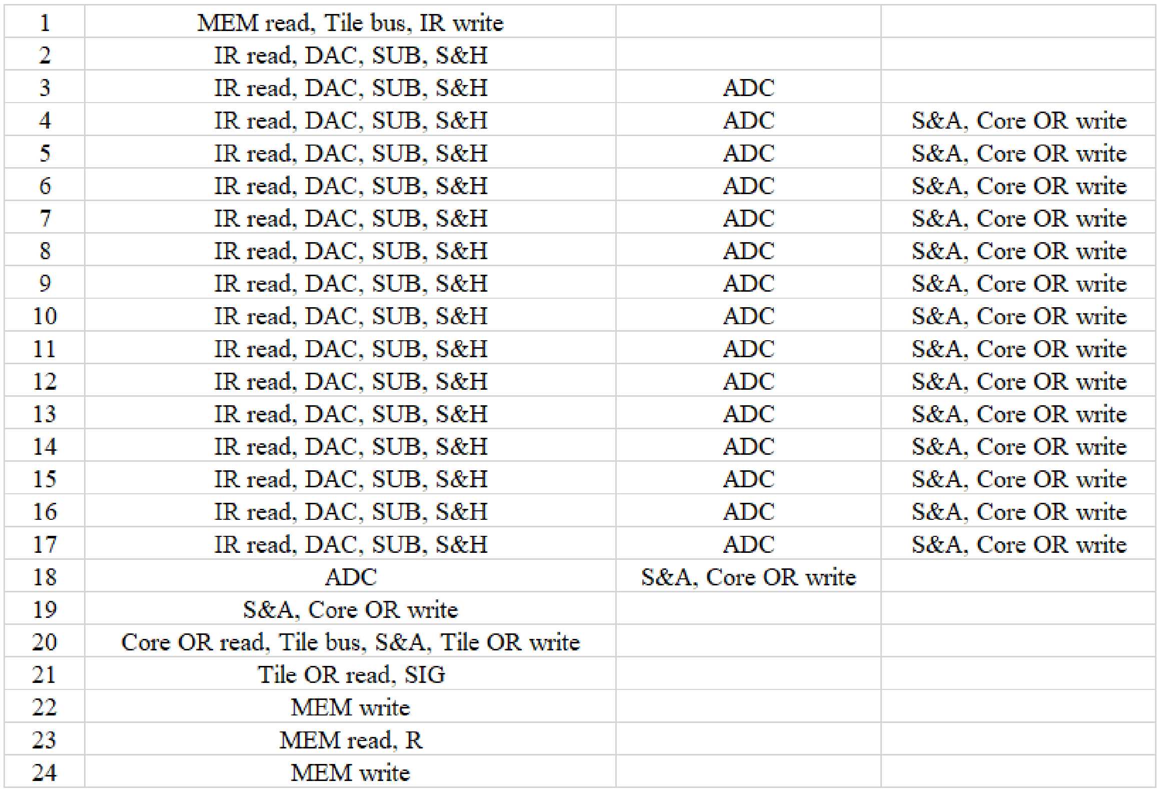
	Area (mm^2)	Power (mW)	Number	Spec
SUB	0.0002	2.4	8	128 x 128, 2-bit MLC
DAC	0.00017	4	128 x 8	1-bit resolution
ADC	0.0096	16	8	8-bit resolution, 1.28 GS/s
S&H	0.00004	0.001	128 x 8	N/A
S&A	0.00024	0.2	4	N/A
IR	0.0021	1.24	N/A	2KB eDRAM
OR	0.0021	1.24	N/A	2KB eDRAM
Core	0.01445	25.081	1	N/A
Core	0.1734	300.972	12	N/A
MEM	0.086	17.66	N/A	64KB eDRAM
Tile bus	0.09	7	N/A	bus width 384 bit
SIG	0.0006	0.52	2	N/A
S&A	0.00006	0.05	1	N/A
MP	0.00024	0.4	1	N/A
OR	0.0021	1.24	N/A	2KB eDRAM
Tile	0.3524	327.842	1	N/A
Tile	112.768	104909.44	320	N/A
R	12.08	3360	320	N/A
Node	124.848	108269.44	1	N/A

Figure 2.2: Power and Area of Each Component.

has a total area of 124.848 mm^2. The total power is 108.26944 W, which is the peak power consumption assuming every component on the chip is functioning in every cycle. We'll have a more detailed analysis of power consumption during run time, which depends on the workloads.

2.2 Intra-layer Pipelining

The kernel of a specific CNN layer may be mapped to one or more tiles depending the size of the kernel. If the kernel is mapped to more than one tile, it need some extra cycles in its intra-layer pipeline to synchronize the OFM. Note that no matter how large the kernel is and how many tiles the kernel needs to be mapped to, we only need one tile to synchronize the OFM for all mapped tiles. The tile is called a collector. This is because all tiles are designed large enough to contain at least one row of the kernel. Fig. 2.3 to Fig. 2.6

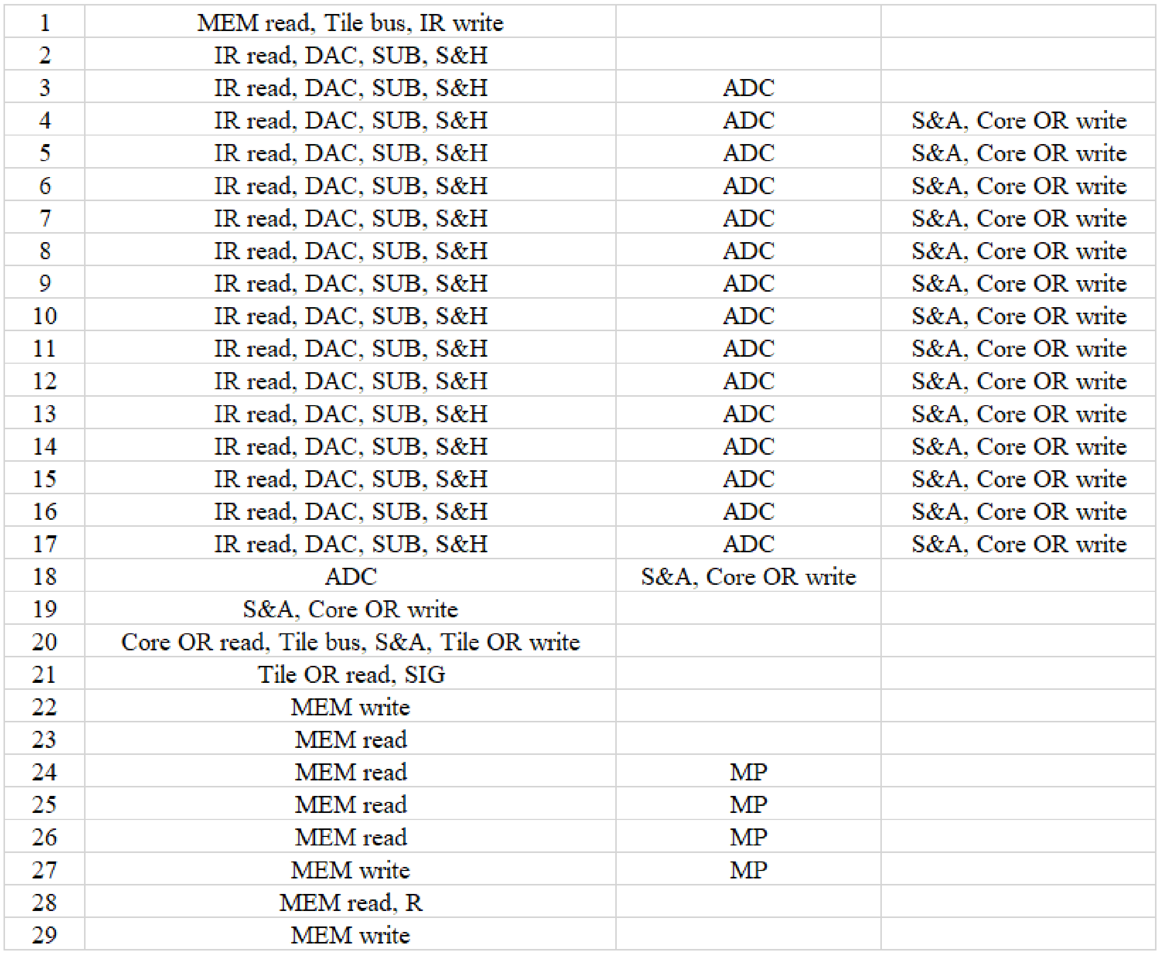


1	MEM read, Tile bus, IR write		
2	IR read, DAC, SUB, S&H		
3	IR read, DAC, SUB, S&H	ADC	
4	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
5	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
6	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
7	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
8	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
9	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
10	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
11	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
12	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
13	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
14	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
15	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
16	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
17	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
18	ADC	S&A, Core OR write	
19	S&A, Core OR write		
20	Core OR read, Tile bus, S&A, Tile OR write		
21	Tile OR read, SIG		
22	MEM write		
23	MEM read, R		
24	MEM write		

Figure 2.3: Intra-layer Pipelining of Single-Mapped Tile with no Pooling.

show the intra-layer pipeline for four different scenarios depending on whether the layer is mapped to a single tile or multiple tiles and whether the layer has pooling operations at the end. Specifically, Fig. 2.3 shows the intra-layer pipeline for single-mapped tile with no pooling. Fig. 2.4 shows the intra-layer pipeline for single-mapped tile with pooling. Fig. 2.5 shows the intra-layer pipeline for multi-mapped tiles and no pooling. Fig. 2.6 shows the intra-layer pipeline for multi-mapped tiles with pooling. All pipelines process one set of inputs (one pixel from all channels). Assume the IFM is a 3D matrix with dimensions $c(channel) \times h(height) \times w(width)$. It takes $h \times w$ logical cycles to pass the entire IFM into this pipeline.

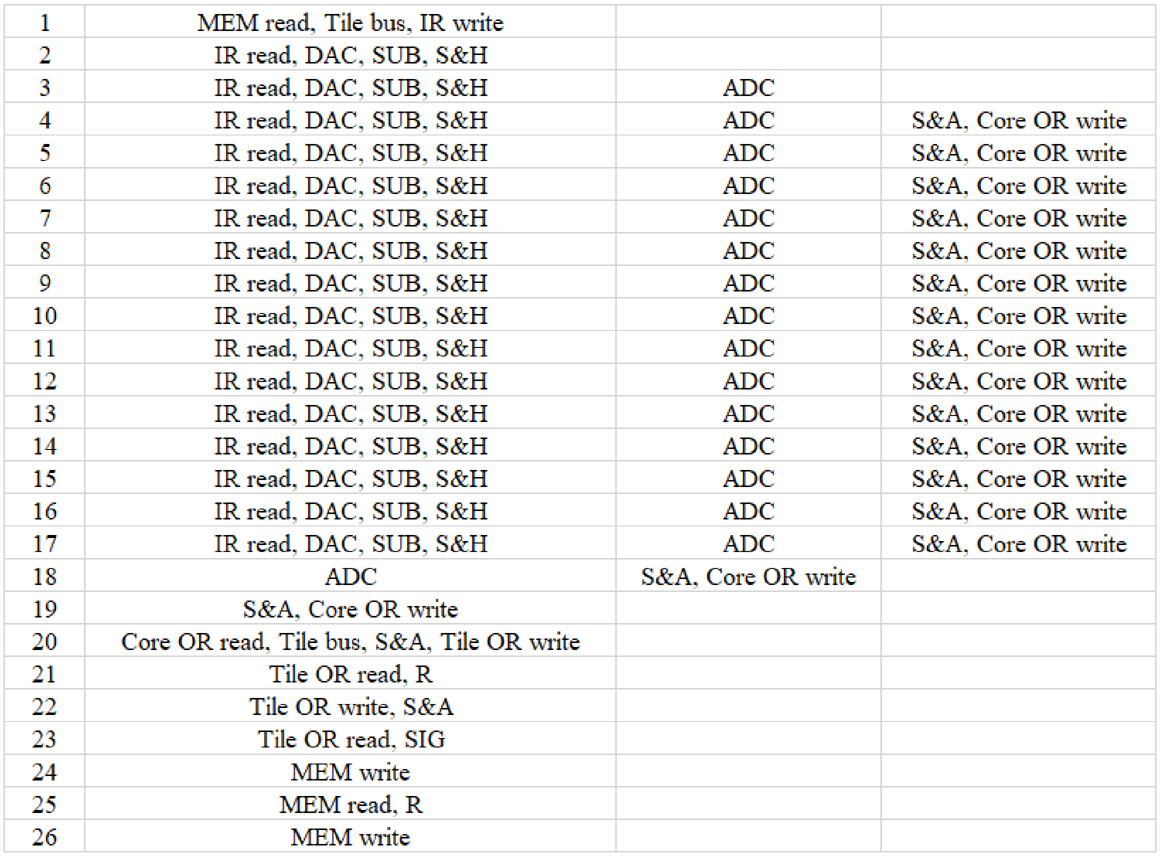
For Fig. 2.3, in the first cycle, the IFMs are read from tile memory, passed through the bus from the tile to each core, and written to the input register in each core. For each cycle in the next 16 cycles, 1 bit from the 16-bit input will be read from the input register, passed through the DAC into the subarray, starting from least significant bit first (LSBF). Then



1	MEM read, Tile bus, IR write		
2	IR read, DAC, SUB, S&H		
3	IR read, DAC, SUB, S&H	ADC	
4	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
5	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
6	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
7	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
8	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
9	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
10	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
11	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
12	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
13	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
14	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
15	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
16	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
17	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
18	ADC	S&A, Core OR write	
19	S&A, Core OR write		
20	Core OR read, Tile bus, S&A, Tile OR write		
21	Tile OR read, SIG		
22	MEM write		
23	MEM read		
24	MEM read	MP	
25	MEM read	MP	
26	MEM read	MP	
27	MEM write	MP	
28	MEM read, R		
29	MEM write		

Figure 2.4: Intra-layer Pipelining of Single-Mapped Tile with Pooling.

the output value will be stored in the sample and hold unit in the same cycle. In cycle 18, the ADC will convert the output to digital value. In cycle 19, digital values from different columns will be shifted and added, and then written to the output register in the core. In cycle 20, the results will be read from the core output register, passed through the bus from each core to the tile, shifted and added with results from other cores in the tile, and finally written to the tile output register. In cycle 21, the results will be read from the tile output register and passed to the sigmoid unit, which is required at the end of every convolution and full-connected layer. In cycle 22, the results will be written to the tile memory. In cycle 23, the results are read from the tile memory and put on the NoC by the router. In cycle 24, after the results arrive at the tiles for the next layer, they are written to their tile

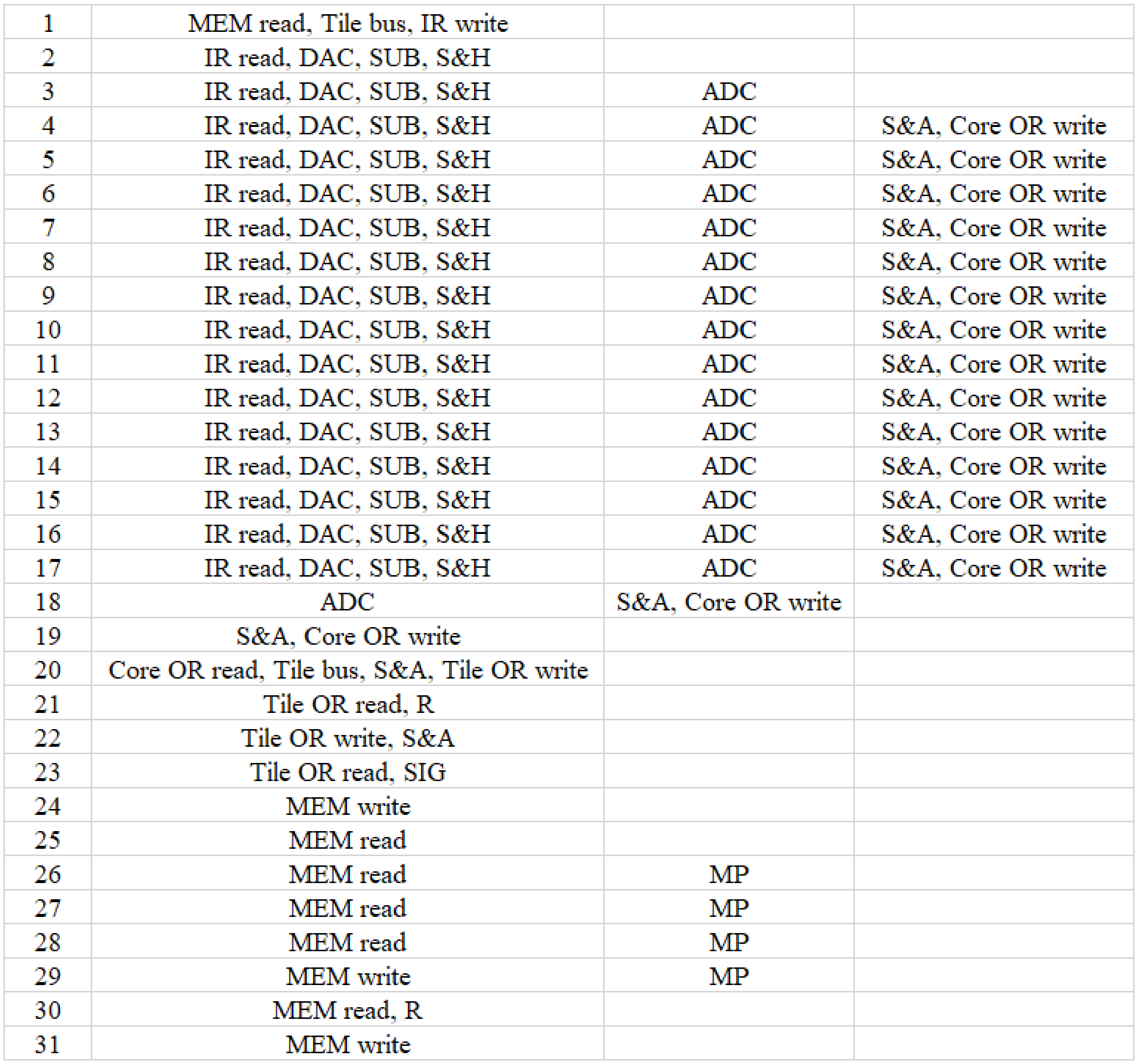


1	MEM read, Tile bus, IR write		
2	IR read, DAC, SUB, S&H		
3	IR read, DAC, SUB, S&H	ADC	
4	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
5	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
6	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
7	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
8	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
9	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
10	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
11	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
12	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
13	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
14	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
15	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
16	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
17	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
18	ADC	S&A, Core OR write	
19	S&A, Core OR write		
20	Core OR read, Tile bus, S&A, Tile OR write		
21	Tile OR read, R		
22	Tile OR write, S&A		
23	Tile OR read, SIG		
24	MEM write		
25	MEM read, R		
26	MEM write		

Figure 2.5: Intra-layer Pipelining of Multi-Mapped Tiles with no Pooling.

memory. Note that in cycles from 3 to 17, we can efficiently pipeline **ADC** stage and **SA, Core OR write** stage in the same cycle as **IR read, DAC, SUB, SH** stage. In this way, we can maximize parallelism and throughput of the intra-layer pipeline. However, the setup time for **ADC** stage and **SA, Core OR write** is 1 cycle, so we need cycle 18 and 19 to do them separately in order to finish processing the 16-bit input before the start of cycle 19.

Sometimes after the convolution layer, there is a pooling layer. We focus on 2×2 pooling in this thesis. Therefore, as shown in Fig. 2.4, we need 4 cycles to read the IFMs from tile memory to the max pooling unit before putting the results on the router and sending them to other tiles. In cycle 27, the results will be written back to the tile memory. Note that we can also pipeline the **MP** stage in the same cycle as the **MEM read** stage to hide some latency. Therefore, it takes 24 cycles to process a set of inputs without pooling and 29 cycles with pooling for single-mapped tile.

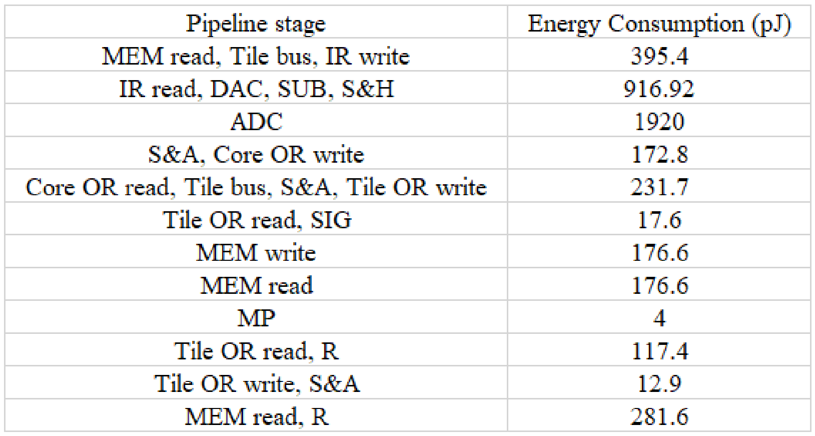


1	MEM read, Tile bus, IR write		
2	IR read, DAC, SUB, S&H		
3	IR read, DAC, SUB, S&H	ADC	
4	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
5	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
6	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
7	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
8	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
9	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
10	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
11	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
12	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
13	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
14	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
15	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
16	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
17	IR read, DAC, SUB, S&H	ADC	S&A, Core OR write
18	ADC	S&A, Core OR write	
19	S&A, Core OR write		
20	Core OR read, Tile bus, S&A, Tile OR write		
21	Tile OR read, R		
22	Tile OR write, S&A		
23	Tile OR read, SIG		
24	MEM write		
25	MEM read		
26	MEM read	MP	
27	MEM read	MP	
28	MEM read	MP	
29	MEM write	MP	
30	MEM read, R		
31	MEM write		

Figure 2.6: Intra-layer Pipelining of Multi-Mapped Tiles with Pooling.

In addition, when a layer is mapped to multiply tiles, we need two extra cycles to send the partial results to the collector tile and sum them to produce the correct final results. In cycles 21 and 22 in Fig. 2.5 and Fig. 2.6, partial results in the tile output register are read and put on the NoC by the router. After the results arrive at the collector tile, they will be shifted and added together and written to the tile output register. Therefore, it takes 26 cycles to process a set of inputs without pooling and 31 cycles with pooling for multi-mapped tiles.

Fig. 2.7 shows the energy consumption of each pipeline stage. Based on the compo-



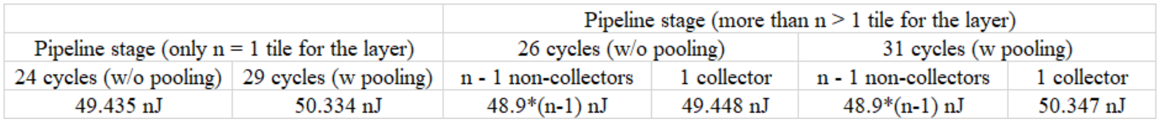
Pipeline stage	Energy Consumption (pJ)
MEM read, Tile bus, IR write	395.4
IR read, DAC, SUB, S&H	916.92
ADC	1920
S&A, Core OR write	172.8
Core OR read, Tile bus, S&A, Tile OR write	231.7
Tile OR read, SIG	17.6
MEM write	176.6
MEM read	176.6
MP	4
Tile OR read, R	117.4
Tile OR write, S&A	12.9
MEM read, R	281.6

Figure 2.7: Energy Consumption of Each Stage.

nents functioning in each cycle, we estimate the total energy consumption in that cycle. It's easy to notice that the stages that involve ADC and DAC consume the most energy. Fig. 2.8 shows the total energy consumption of the 24-stage, 29-stage, 26-stage, and 31-stage pipeline. We observe that energy consumption doesn't change much but latency varies much due to the pooling layers and synchronization. In addition, pooling layers can further degrade the performance of inter-layer pipelining because the next layer has to wait for the pooled result which comes from different columns of the current feature map, which introduces extra pipeline bubbles, increases latency, and decreases throughput. We will discuss more about inter-layer pipelining in Section 2.3.

2.3 Inter-layer Pipelining

After figuring out the intra-layer pipeline, we focus on designing an efficient pipeline that maximizes intra-layer parallelism for CNN inference. We observe that we don't need to wait for the previous layer to produce the entire OFM in order to start the current layer. We



		Pipeline stage (more than n > 1 tile for the layer)			
Pipeline stage (only n = 1 tile for the layer)		26 cycles (w/o pooling)		31 cycles (w pooling)	
24 cycles (w/o pooling)	29 cycles (w pooling)	n - 1 non-collectors	1 collector	n - 1 non-collectors	1 collector
49.435 nJ	50.334 nJ	48.9*(n-1) nJ	49.448 nJ	48.9*(n-1) nJ	50.347 nJ

Figure 2.8: Energy Consumption of Each Pipeline.

only need to wait for enough information from the previous layer that is able to start the first convolution of the current layer. To better illustrate this concept, we define the IFM of the previous layer to be I, the kernel of the previous layer to be K, and the OFM of the previous layer (also IFM of the current layer) to be O. I is a 3D matrix with dimensions $c(channel) \times h(height) \times w(width)$. K is a 4D matrix with dimensions $n(kernel) \times c(channel) \times l(length) \times l(length)$. O is a 3D matrix with dimensions $c(kernel) \times h(height) \times w(width)$. The number of values in O that the current layer needs to wait is shown in Eq. 2.1. In addition, the number of cycles the current layer needs to wait is shown in Eq 2.2. Note that we assume the kernel strides in the row-majored fashion.

$$valuesWait = (w \times (l-1) + l) \times n \tag{2.1}$$

$$cyclesWait = w \times (l-1) + l \tag{2.2}$$

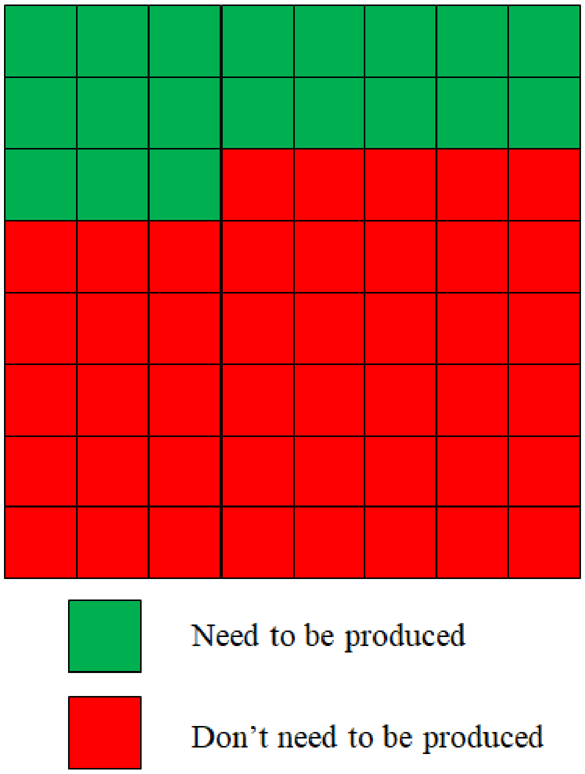


Figure 2.9: # of Values Needed to Start the Current Layer.

Fig. 2.7 shows an example of a IFM for the current layer with $h = 8$, $w = 8$, and $n = 1$. The kernel is 3×3. The number of values we need to wait is $8 \times (3 - 1) + 3 = 19$. It's easy to see that as long as the top 3×3 grid is ready, the current layer can start its convolution operation. In the same cycle the convolution operation is performed, the previous layer will generate another value in the OFM, which is another value for in the IFM for the current layer. Therefore, in the next cycle, the current layer can perform another convolution operation. Then we have a good pipeline which overlaps the operations of the current layer and previous layer in parallel. However, there are still pipeline bubbles for the current layer if there is a pooling layer between the previous layer and the current layer. In general, a pooling size of 2×2 will slow down the current layer pipeline by a factor of 4, because in average the previous layer has to produce four output values in order to generate only one more value in the IFM for the current layer to move on. A good hardware solution for this problem is to replicate weights for the previous layer to speed up the processing speed of the previous layer pipeline and reduce the pipeline bubbles for the current layer pipeline. We will discuss more about weight replication in Chapter 3.

2.4 Batch Pipelining

After designing the intra-layer pipeline and inter-layer pipeline, we consider the input to the chip as a stream of images. In order to achieve real-time inference, we need to reach a certain frame rate (FPS). Therefore, it's important to figure out another pipeline for a batch of input images. We assume that a batch (B) of input images come per second. In order to increase B, we design a pipeline that overlaps the latency among input images that come consecutively to each other.

We follow two design principles to design the batch pipeline. First, there should be no structural hazard, which means in the same cycle, the pipeline cannot process a specific layer (say layer 1) from two or more different images. In other words, a specific layer in a specific cycle can only process one single image. Second, dependencies between

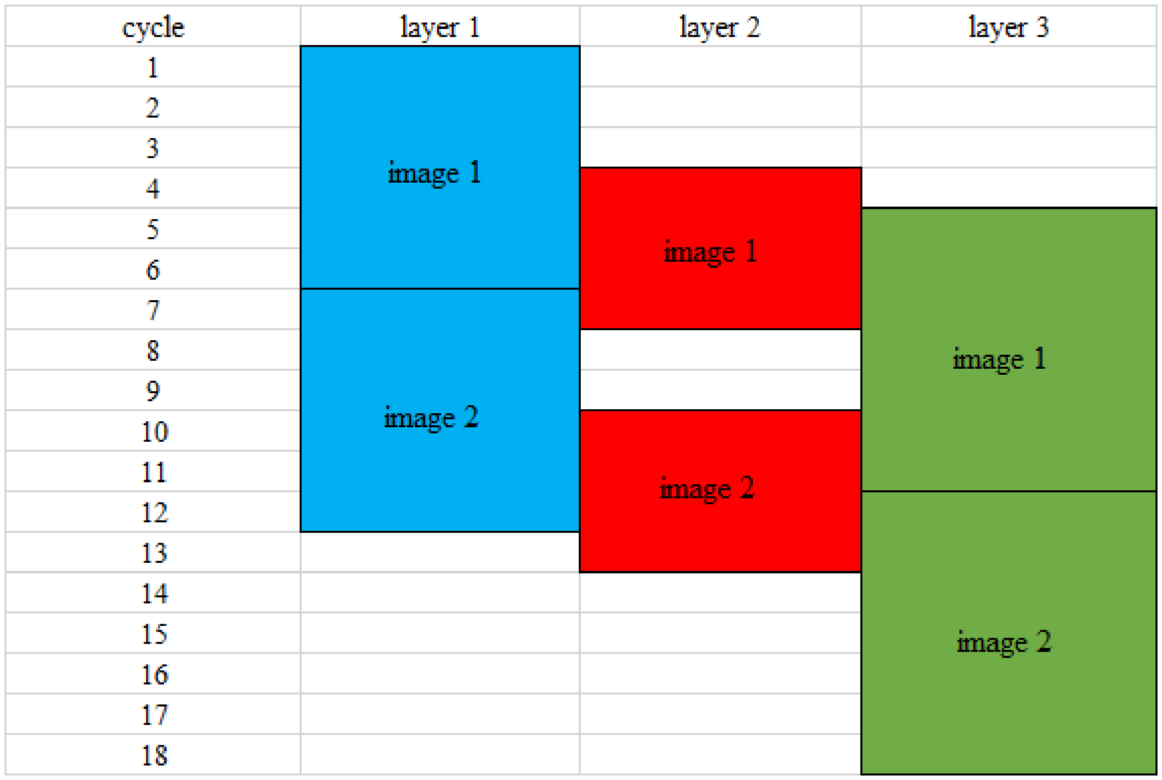


Figure 2.10: Batch Pipeline.

consecutive layer (say layer 1 and layer 2) should be strictly followed for all images. In other ways, if layer 2 has to wait for 2 cycles after layer 1 starts, all layer 2 from all images have to wait for 2 cycles after the corresponding layer 1 starts.

Fig. 2.8 shows an example of batch pipeline with two input images and each image has three layers. The setup is the following: layer 1 has six cycles, layer 2 has four cycles, layer 3 has seven cycles. In addition, layer 2 has to wait for 3 cycles after layer 1 starts and layer 3 has to wait for 1 cycle after layer 2 starts.

For image 1, layer 1 starts from cycle 1 and ends at cycle 6, layer 2 starts cycle 4 and ends at cycle 7, and layer 3 starts at cycle 5 and ends at cycle 11, which satisfy principle 1 and principle 2. For image 2, since layer 1 is always independent of each other, it can directly starts from cycle 7 and ends at cycle 12. For layer 2, it cannot simply start at cycle 8 since layer 2 needs to wait for 3 cycles after layer 1 starts to satisfy principle 1. Therefore, layer 2 of image 2 has to start at cycle 10. For layer 3, it cannot directly starts at cycle 11, since layer 3 of image 1 is still in the pipeline. To satisfy principle 1, layer 3 of image 2 has to start at cycle 12. As a result, Fig. 2.8 shows the pipeline with the lowest latency to process the two image inputs with three layers for each image. It takes a total of 18 cycles. If we don't do batch pipeline and only starts processing the next image when he first image completely finishes, then the total cycle count is $2 \times 11 = 22$ cycles. In our simulator, we take advantage of the batch pipeline to maximize the performance of the overall architecture. More details about the performance-related results will be given in Chapter 3.

2.5 NoC Model

The topology of a NoC describes the connection between routers via links/channels. NoC topologies include bus, ring, mesh, torus, flattened butterfly, fully connected and so on. The most common topology is a 2D mesh because it can be laid out easily. In our design, the NoC uses a 16×20 2D mesh topology.

The routing of a NoC describes the links that a flit takes from the source router to the destination router. Dimension-ordered routing (DOR) such as XY and YX routing is typically used. XY routing means when choosing the routing path from the source to the destination, the flit always goes horizontal (X direction) and then vertical direction (Y direction). In addition, a turn model such as north-last model or east-first model can be used. It disallows some turns to get rid of deadlocks in the NoC. In our design, we use XY/YX routing to provide the maximum path diversity to the NoC for better throughput. Path diversity describes the number of paths available from the source to the destination. In addition, we set the link width to be 64 bits, which is the flit size. A packet has 8 flits, which is $64 \times 8 = 512$ bits.

The flow control of a NoC describes when a flit can traverse to the downstream router or it has to stay at the upstream router if there is traffic in the NoC. The most common algorithm is the wormhole flow control. In wormhole, The link is allocated at the packet

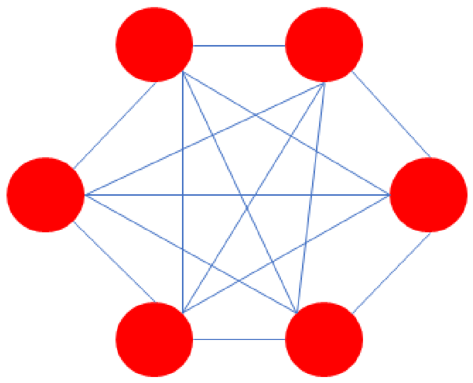


Figure 2.11: Fully Connected Topology.

level and the buffer is allocated at the flit level. It significantly improves the performance of virtual cut-through flow control because its buffer can have flits from different packets. However, wormhole still suffers from poor link utilization and results in head-of-line blocking (HoL). HoL blocking means if the first flit in the buffer cannot move, all of the rest flits in the buffer cannot move either. In our design, we use wormhole as one baseline.

In order to present the total latency in cycles to send a packet from the source to the destination, we define the wire delay for one link to be t_w, the hop count to be H, the contention delay to be t_c, and the serialization delay to be t_s. A typical formula for latency in cycles is defined as

$$T = t_w \times H + t_c + t_s \tag{2.3}$$

where the bottleneck is the term H. The ideal solution to reduce H down to 1 is to use the fully connected topology, as shown in Fig. 2.11. However, since it's nearly impossible to lay out a topology like this, we will never achieve this ideal performance. Instead, SMART flow control algorithm can make the NoC behave closely to an ideal fully connected topology.

According to Prof. Tushar Krishna's PhD thesis at MIT [12], and two other papers about SMART (single-cycle multi-hop asynchronous repeated traversal) flow control [13, 14], our NoC model uses a similar approach to reduce NoC latency and increase the overall throughput. According to [13], place-and-route repeated wires can go up to 16 mm in 1 ns

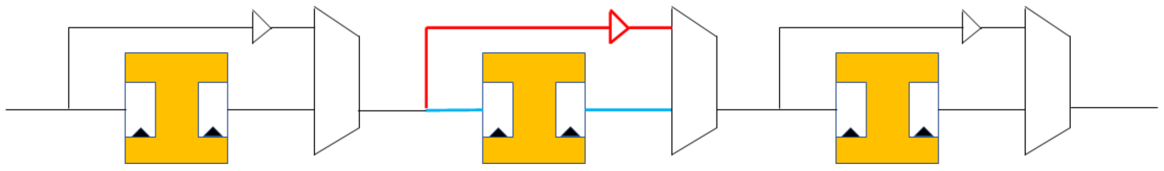


Figure 2.12: Router Bypass in SMART.

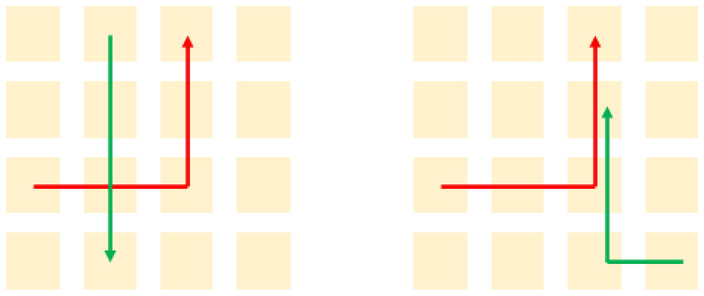


Figure 2.13: Two Cases in SMART.

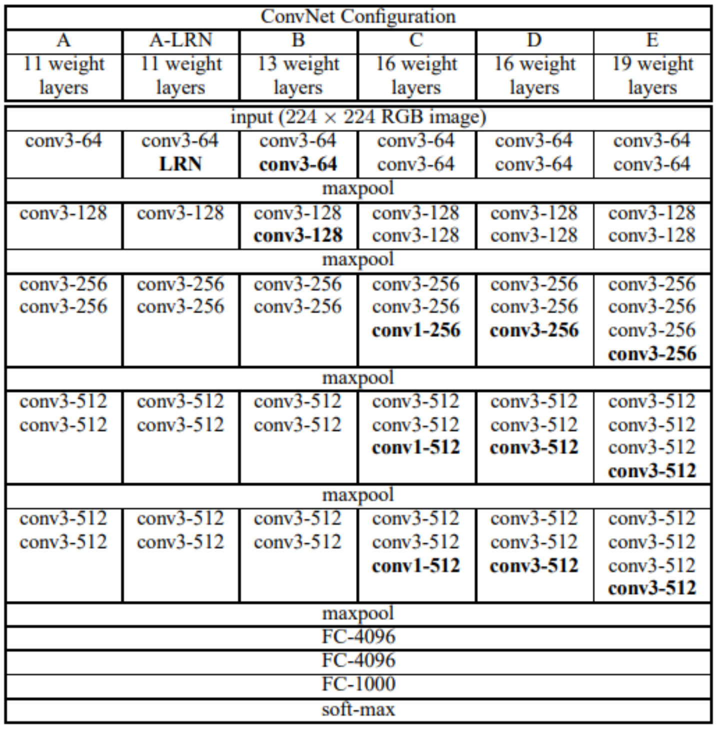
in 45 nm technology node. It can go further in projected 32 nm technology node because wire delay remains constant or decreases slightly due to technology scaling [13]. Therefore, on-chip wires can go fast enough to transfer across the chip within 1 clock cycle. The high level idea to achieve SMART flow control is to use multiplexers to bypass the routers on the path from the source to the destination. Fig. 2.12 shows that in order to bypass the middle router, we use a mux to choose the input directly (red path) instead of pass it to the router (blue path). Fig. 2.13 shows two possible cases for SMART flow control. The left case takes only 1 cycle because the two flits don't have a shared link in their individual route. However, the right case takes 2 cycles because the two flits have a overlap in their individual routes. Therefore, we need to enforce a certain order for the two paths. It turns out the right case is the bottleneck for the SMART flow control. More details about the detailed control signal setup in SMART flow control can be found in [13]. We simulate the performance of wormhole and SMART flow control in garnet2.0 using trace-based input. More details about the experiment and results will be discussed in Chapter 3.

CHAPTER 3
SIMULATION AND EXPERIMENT RESULTS

This chapter first presents the workloads used in our simulation and the weight mapping for each individual workload. Then we explore the effect of weight replication of different CNN layers on the performance. Our simulation is cycle-accurate and catches the exact cycle count of each layer in each workload. Finally, we report the throughput and energy efficiency for each workload.

3.1 Workloads

We use VGG (A-E) [15] for the large-scale data set ImageNet [16] as our workloads. VGG makes a thorough evaluation of networks of increasing depth using an architecture with



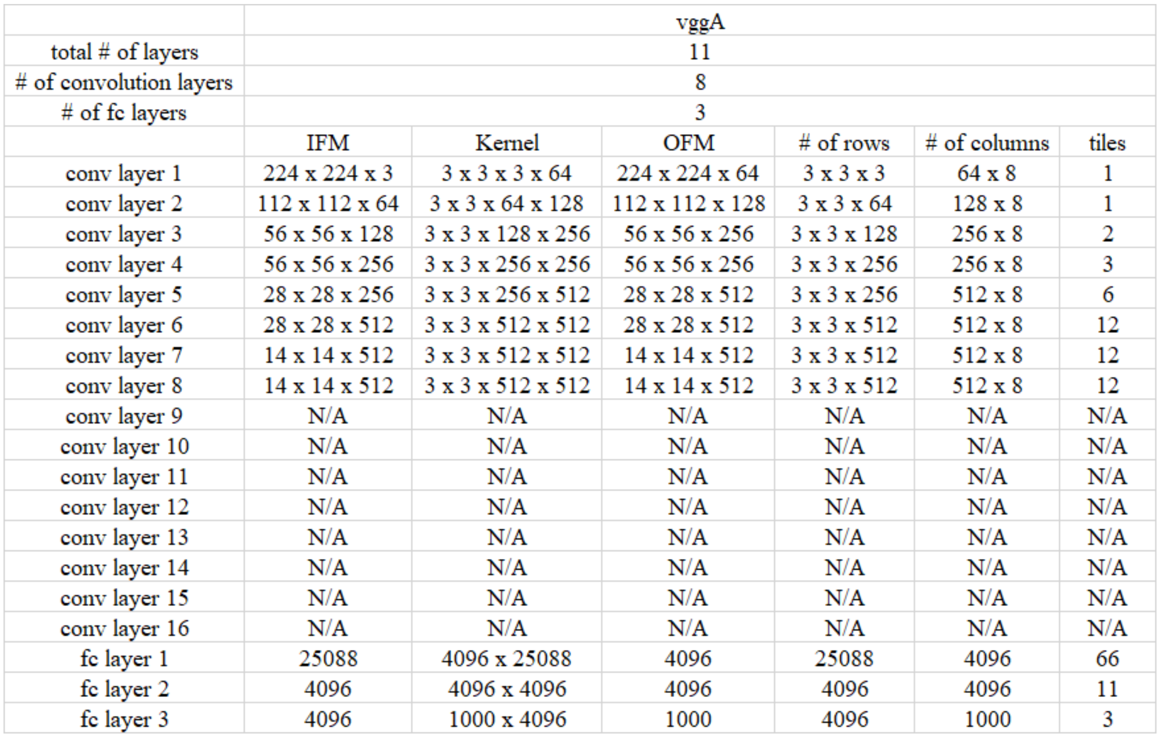
ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 3.1: VGG Configuration [12].

very small (3×3) convolution filters [15]. Compared to previous work like AlexNet [17], VGG improves the accuracy of computer vision and pattern recognition tasks by a wide margin, which is achieved by pushing the CNN depth from a few layers to tens of layers [15]. Fig. 3.1 shows the configuration of each layer for each VGG network, which is redacted from [15]. The configuration shows the kernel size of each layer, and then it's easy to figure out the size of IFM and OFM for each layer.

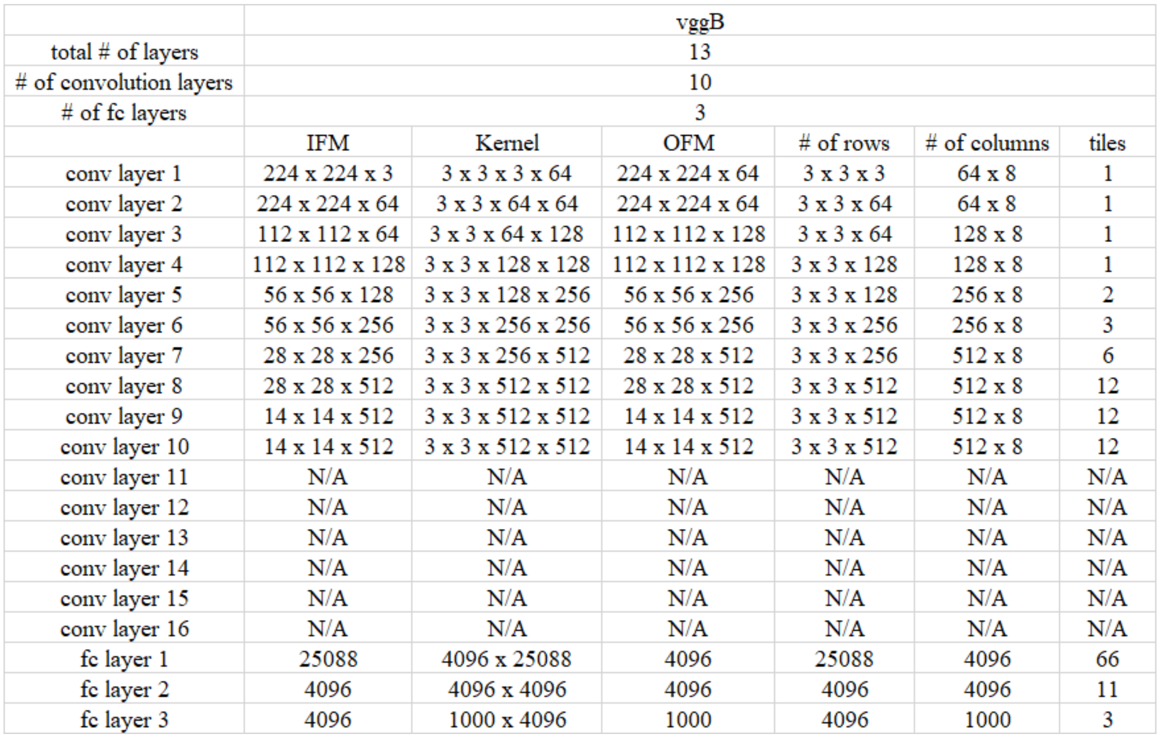
3.2 Weight Mapping

Fig. 3.2 to Fig. 3.6 shows the exact size of IFM, kernel, and OFM for each CNN layer in each VGG. After unrolling the kernel into a large 2D matrix of ReRAM cells, we can calculate the total number of rows and columns needed in the hardware. Suppose the kernel is a 4D tensor with dimensions $l(length) \times l(length) \times c(channel) \times n(kernel)$, then the number of rows is $l \times l \times c$ and the number of columns is $n \times 8$. Since each cell is 2-bit,



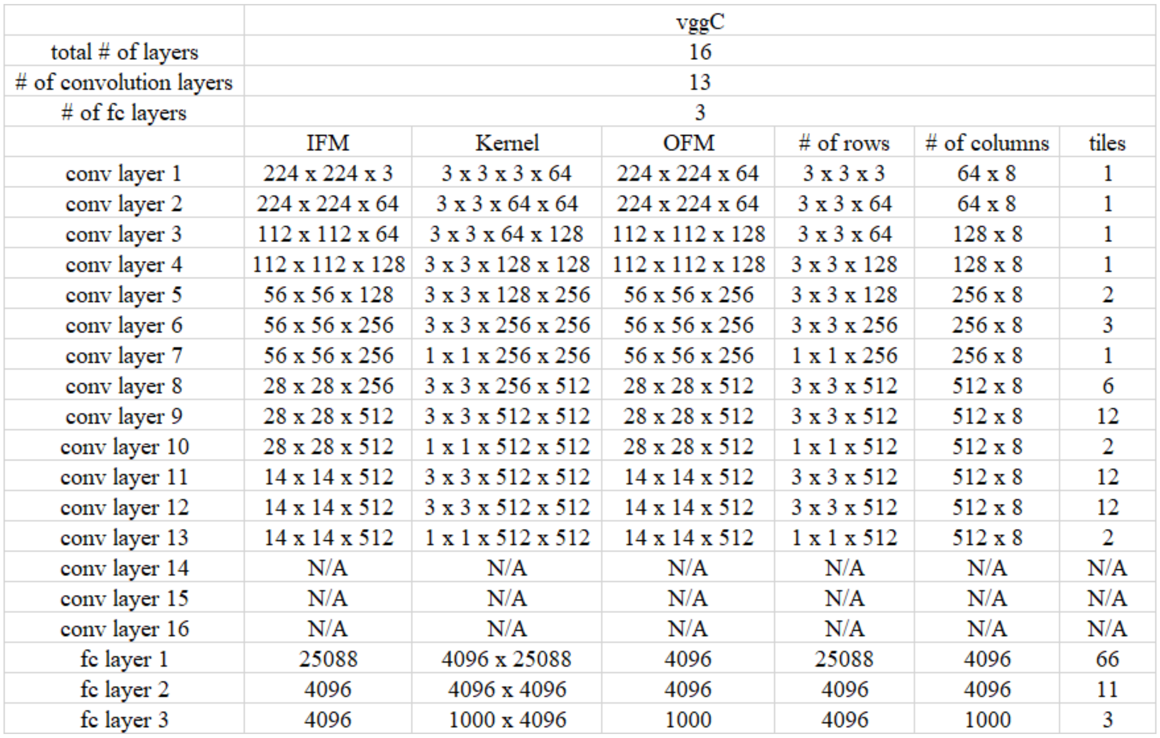
	vggA					
total # of layers	11					
# of convolution layers	8					
# of fc layers	3					
	IFM	Kernel	OFM	# of rows	# of columns	tiles
conv layer 1	224 x 224 x 3	3 x 3 x 3 x 64	224 x 224 x 64	3 x 3 x 3	64 x 8	1
conv layer 2	112 x 112 x 64	3 x 3 x 64 x 128	112 x 112 x 128	3 x 3 x 64	128 x 8	1
conv layer 3	56 x 56 x 128	3 x 3 x 128 x 256	56 x 56 x 256	3 x 3 x 128	256 x 8	2
conv layer 4	56 x 56 x 256	3 x 3 x 256 x 256	56 x 56 x 256	3 x 3 x 256	256 x 8	3
conv layer 5	28 x 28 x 256	3 x 3 x 256 x 512	28 x 28 x 512	3 x 3 x 256	512 x 8	6
conv layer 6	28 x 28 x 512	3 x 3 x 512 x 512	28 x 28 x 512	3 x 3 x 512	512 x 8	12
conv layer 7	14 x 14 x 512	3 x 3 x 512 x 512	14 x 14 x 512	3 x 3 x 512	512 x 8	12
conv layer 8	14 x 14 x 512	3 x 3 x 512 x 512	14 x 14 x 512	3 x 3 x 512	512 x 8	12
conv layer 9	N/A	N/A	N/A	N/A	N/A	N/A
conv layer 10	N/A	N/A	N/A	N/A	N/A	N/A
conv layer 11	N/A	N/A	N/A	N/A	N/A	N/A
conv layer 12	N/A	N/A	N/A	N/A	N/A	N/A
conv layer 13	N/A	N/A	N/A	N/A	N/A	N/A
conv layer 14	N/A	N/A	N/A	N/A	N/A	N/A
conv layer 15	N/A	N/A	N/A	N/A	N/A	N/A
conv layer 16	N/A	N/A	N/A	N/A	N/A	N/A
fc layer 1	25088	4096 x 25088	4096	25088	4096	66
fc layer 2	4096	4096 x 4096	4096	4096	4096	11
fc layer 3	4096	1000 x 4096	1000	4096	1000	3

Figure 3.2: vggA.



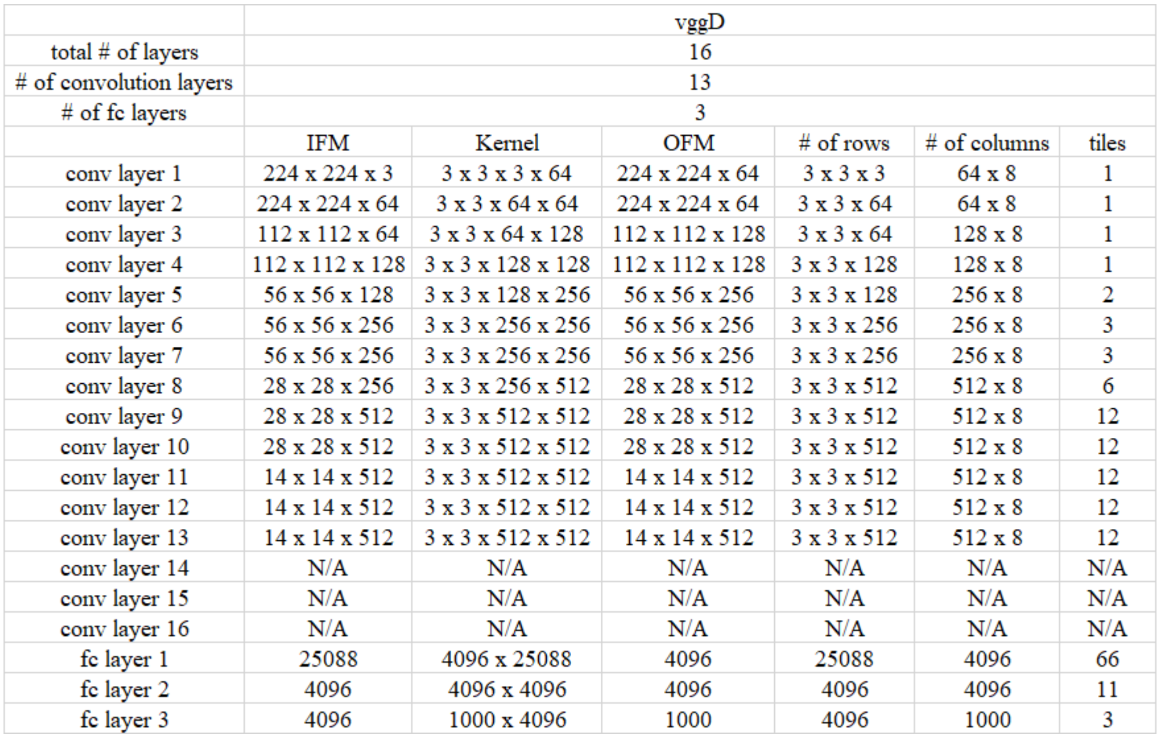
	vggB					
total # of layers	13					
# of convolution layers	10					
# of fc layers	3					
	IFM	Kernel	OFM	# of rows	# of columns	tiles
conv layer 1	224 x 224 x 3	3 x 3 x 3 x 64	224 x 224 x 64	3 x 3 x 3	64 x 8	1
conv layer 2	224 x 224 x 64	3 x 3 x 64 x 64	224 x 224 x 64	3 x 3 x 64	64 x 8	1
conv layer 3	112 x 112 x 64	3 x 3 x 64 x 128	112 x 112 x 128	3 x 3 x 64	128 x 8	1
conv layer 4	112 x 112 x 128	3 x 3 x 128 x 128	112 x 112 x 128	3 x 3 x 128	128 x 8	1
conv layer 5	56 x 56 x 128	3 x 3 x 128 x 256	56 x 56 x 256	3 x 3 x 128	256 x 8	2
conv layer 6	56 x 56 x 256	3 x 3 x 256 x 256	56 x 56 x 256	3 x 3 x 256	256 x 8	3
conv layer 7	28 x 28 x 256	3 x 3 x 256 x 512	28 x 28 x 512	3 x 3 x 256	512 x 8	6
conv layer 8	28 x 28 x 512	3 x 3 x 512 x 512	28 x 28 x 512	3 x 3 x 512	512 x 8	12
conv layer 9	14 x 14 x 512	3 x 3 x 512 x 512	14 x 14 x 512	3 x 3 x 512	512 x 8	12
conv layer 10	14 x 14 x 512	3 x 3 x 512 x 512	14 x 14 x 512	3 x 3 x 512	512 x 8	12
conv layer 11	N/A	N/A	N/A	N/A	N/A	N/A
conv layer 12	N/A	N/A	N/A	N/A	N/A	N/A
conv layer 13	N/A	N/A	N/A	N/A	N/A	N/A
conv layer 14	N/A	N/A	N/A	N/A	N/A	N/A
conv layer 15	N/A	N/A	N/A	N/A	N/A	N/A
conv layer 16	N/A	N/A	N/A	N/A	N/A	N/A
fc layer 1	25088	4096 x 25088	4096	25088	4096	66
fc layer 2	4096	4096 x 4096	4096	4096	4096	11
fc layer 3	4096	1000 x 4096	1000	4096	1000	3

Figure 3.3: vggB.



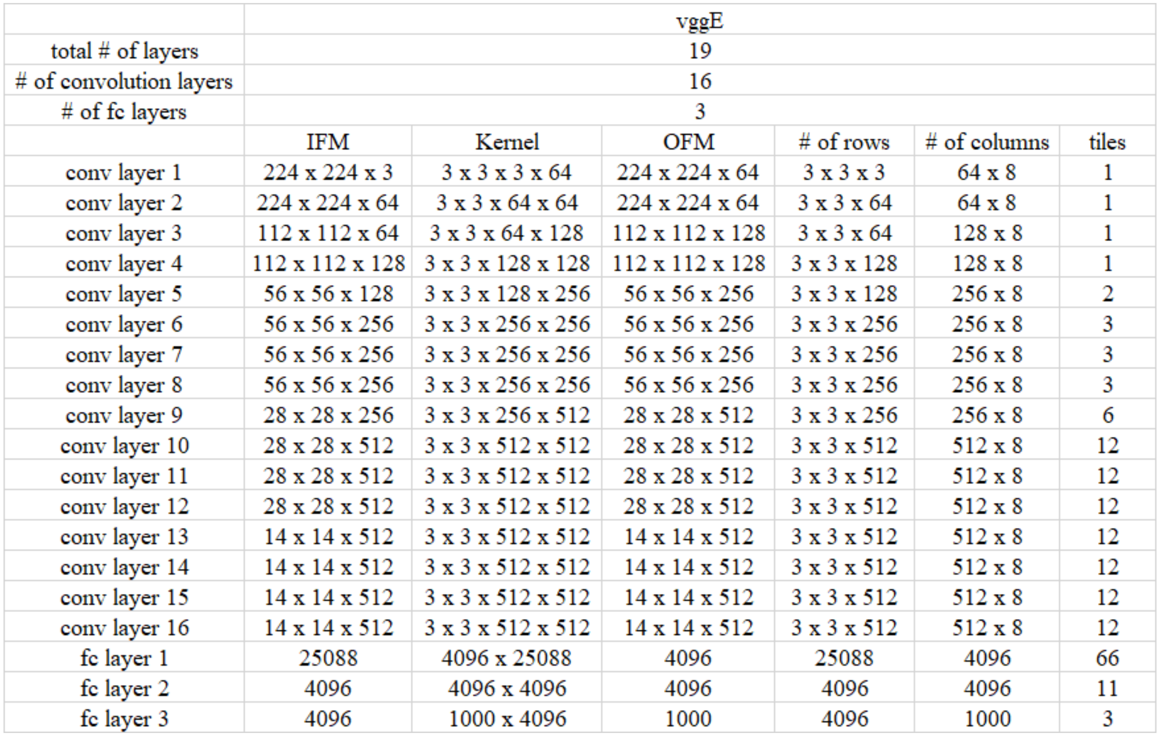
	vggC					
total # of layers	16					
# of convolution layers	13					
# of fc layers	3					
	IFM	Kernel	OFM	# of rows	# of columns	tiles
conv layer 1	224 x 224 x 3	3 x 3 x 3 x 64	224 x 224 x 64	3 x 3 x 3	64 x 8	1
conv layer 2	224 x 224 x 64	3 x 3 x 64 x 64	224 x 224 x 64	3 x 3 x 64	64 x 8	1
conv layer 3	112 x 112 x 64	3 x 3 x 64 x 128	112 x 112 x 128	3 x 3 x 64	128 x 8	1
conv layer 4	112 x 112 x 128	3 x 3 x 128 x 128	112 x 112 x 128	3 x 3 x 128	128 x 8	1
conv layer 5	56 x 56 x 128	3 x 3 x 128 x 256	56 x 56 x 256	3 x 3 x 128	256 x 8	2
conv layer 6	56 x 56 x 256	3 x 3 x 256 x 256	56 x 56 x 256	3 x 3 x 256	256 x 8	3
conv layer 7	56 x 56 x 256	1 x 1 x 256 x 256	56 x 56 x 256	1 x 1 x 256	256 x 8	1
conv layer 8	28 x 28 x 256	3 x 3 x 256 x 512	28 x 28 x 512	3 x 3 x 512	512 x 8	6
conv layer 9	28 x 28 x 512	3 x 3 x 512 x 512	28 x 28 x 512	3 x 3 x 512	512 x 8	12
conv layer 10	28 x 28 x 512	1 x 1 x 512 x 512	28 x 28 x 512	1 x 1 x 512	512 x 8	2
conv layer 11	14 x 14 x 512	3 x 3 x 512 x 512	14 x 14 x 512	3 x 3 x 512	512 x 8	12
conv layer 12	14 x 14 x 512	3 x 3 x 512 x 512	14 x 14 x 512	3 x 3 x 512	512 x 8	12
conv layer 13	14 x 14 x 512	1 x 1 x 512 x 512	14 x 14 x 512	1 x 1 x 512	512 x 8	2
conv layer 14	N/A	N/A	N/A	N/A	N/A	N/A
conv layer 15	N/A	N/A	N/A	N/A	N/A	N/A
conv layer 16	N/A	N/A	N/A	N/A	N/A	N/A
fc layer 1	25088	4096 x 25088	4096	25088	4096	66
fc layer 2	4096	4096 x 4096	4096	4096	4096	11
fc layer 3	4096	1000 x 4096	1000	4096	1000	3

Figure 3.4: vggC.



	vggD					
total # of layers	16					
# of convolution layers	13					
# of fc layers	3					
	IFM	Kernel	OFM	# of rows	# of columns	tiles
conv layer 1	224 x 224 x 3	3 x 3 x 3 x 64	224 x 224 x 64	3 x 3 x 3	64 x 8	1
conv layer 2	224 x 224 x 64	3 x 3 x 64 x 64	224 x 224 x 64	3 x 3 x 64	64 x 8	1
conv layer 3	112 x 112 x 64	3 x 3 x 64 x 128	112 x 112 x 128	3 x 3 x 64	128 x 8	1
conv layer 4	112 x 112 x 128	3 x 3 x 128 x 128	112 x 112 x 128	3 x 3 x 128	128 x 8	1
conv layer 5	56 x 56 x 128	3 x 3 x 128 x 256	56 x 56 x 256	3 x 3 x 128	256 x 8	2
conv layer 6	56 x 56 x 256	3 x 3 x 256 x 256	56 x 56 x 256	3 x 3 x 256	256 x 8	3
conv layer 7	56 x 56 x 256	3 x 3 x 256 x 256	56 x 56 x 256	3 x 3 x 256	256 x 8	3
conv layer 8	28 x 28 x 256	3 x 3 x 256 x 512	28 x 28 x 512	3 x 3 x 512	512 x 8	6
conv layer 9	28 x 28 x 512	3 x 3 x 512 x 512	28 x 28 x 512	3 x 3 x 512	512 x 8	12
conv layer 10	28 x 28 x 512	3 x 3 x 512 x 512	28 x 28 x 512	3 x 3 x 512	512 x 8	12
conv layer 11	14 x 14 x 512	3 x 3 x 512 x 512	14 x 14 x 512	3 x 3 x 512	512 x 8	12
conv layer 12	14 x 14 x 512	3 x 3 x 512 x 512	14 x 14 x 512	3 x 3 x 512	512 x 8	12
conv layer 13	14 x 14 x 512	3 x 3 x 512 x 512	14 x 14 x 512	3 x 3 x 512	512 x 8	12
conv layer 14	N/A	N/A	N/A	N/A	N/A	N/A
conv layer 15	N/A	N/A	N/A	N/A	N/A	N/A
conv layer 16	N/A	N/A	N/A	N/A	N/A	N/A
fc layer 1	25088	4096 x 25088	4096	25088	4096	66
fc layer 2	4096	4096 x 4096	4096	4096	4096	11
fc layer 3	4096	1000 x 4096	1000	4096	1000	3

Figure 3.5: vggD.



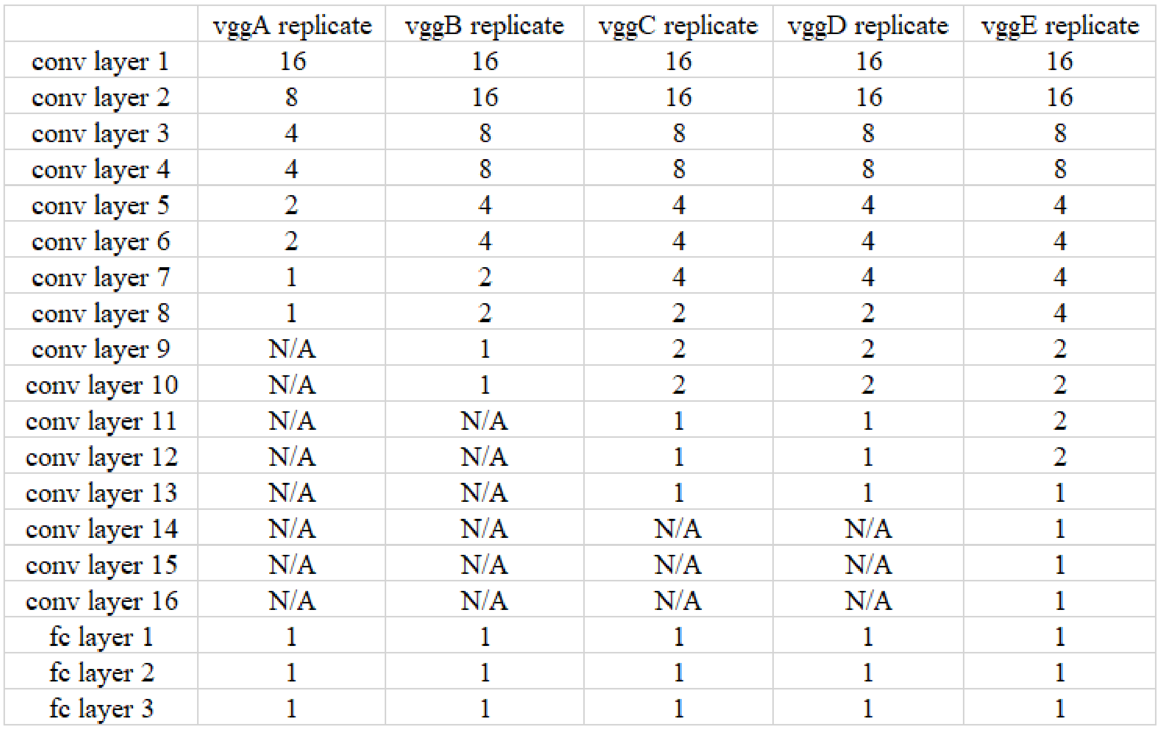
	vggE					
total # of layers	19					
# of convolution layers	16					
# of fc layers	3					
	IFM	Kernel	OFM	# of rows	# of columns	tiles
conv layer 1	224 x 224 x 3	3 x 3 x 3 x 64	224 x 224 x 64	3 x 3 x 3	64 x 8	1
conv layer 2	224 x 224 x 64	3 x 3 x 64 x 64	224 x 224 x 64	3 x 3 x 64	64 x 8	1
conv layer 3	112 x 112 x 64	3 x 3 x 64 x 128	112 x 112 x 128	3 x 3 x 64	128 x 8	1
conv layer 4	112 x 112 x 128	3 x 3 x 128 x 128	112 x 112 x 128	3 x 3 x 128	128 x 8	1
conv layer 5	56 x 56 x 128	3 x 3 x 128 x 256	56 x 56 x 256	3 x 3 x 128	256 x 8	2
conv layer 6	56 x 56 x 256	3 x 3 x 256 x 256	56 x 56 x 256	3 x 3 x 256	256 x 8	3
conv layer 7	56 x 56 x 256	3 x 3 x 256 x 256	56 x 56 x 256	3 x 3 x 256	256 x 8	3
conv layer 8	56 x 56 x 256	3 x 3 x 256 x 256	56 x 56 x 256	3 x 3 x 256	256 x 8	3
conv layer 9	28 x 28 x 256	3 x 3 x 256 x 512	28 x 28 x 512	3 x 3 x 256	256 x 8	6
conv layer 10	28 x 28 x 512	3 x 3 x 512 x 512	28 x 28 x 512	3 x 3 x 512	512 x 8	12
conv layer 11	28 x 28 x 512	3 x 3 x 512 x 512	28 x 28 x 512	3 x 3 x 512	512 x 8	12
conv layer 12	28 x 28 x 512	3 x 3 x 512 x 512	28 x 28 x 512	3 x 3 x 512	512 x 8	12
conv layer 13	14 x 14 x 512	3 x 3 x 512 x 512	14 x 14 x 512	3 x 3 x 512	512 x 8	12
conv layer 14	14 x 14 x 512	3 x 3 x 512 x 512	14 x 14 x 512	3 x 3 x 512	512 x 8	12
conv layer 15	14 x 14 x 512	3 x 3 x 512 x 512	14 x 14 x 512	3 x 3 x 512	512 x 8	12
conv layer 16	14 x 14 x 512	3 x 3 x 512 x 512	14 x 14 x 512	3 x 3 x 512	512 x 8	12
fc layer 1	25088	4096 x 25088	4096	25088	4096	66
fc layer 2	4096	4096 x 4096	4096	4096	4096	11
fc layer 3	4096	1000 x 4096	1000	4096	1000	3

Figure 3.6: vggE.

we need eight columns to encode one weight. More details about mapping can be found in Chapter 2.

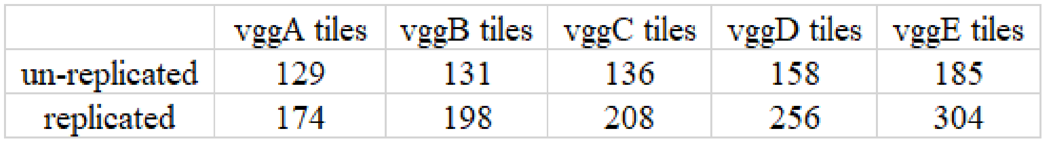
3.3 Weight Replication

Pooling layers degrade the performance of inter-layer pipelining because the next layer has to wait for the pooled results which come from different columns of the current feature map. This introduces extra pipeline bubbles, increases latency, and decreases throughput. In order to have a more balanced pipeline design, we replicate more weights for the first few layers while replicate less weights for the deep layers. Specifically, all five VGGs are down-sampled five times: 224×224, 112×112, 56×56, 28×28, 14×14, 7×7. Each time a grid of 2×2 is applied to the whole OFM. In order to satisfy this trend, we also replicate the weights 16 times, 8 times, 4 times, 2 times, and 1 time. Fig. 3.7 shows the number of times the weights are replicated in each layer for each VGG. Fig. 3.8 shows the total number of tiles required with and without weight replication for each VGG. All schemes



	vggA replicate	vggB replicate	vggC replicate	vggD replicate	vggE replicate
conv layer 1	16	16	16	16	16
conv layer 2	8	16	16	16	16
conv layer 3	4	8	8	8	8
conv layer 4	4	8	8	8	8
conv layer 5	2	4	4	4	4
conv layer 6	2	4	4	4	4
conv layer 7	1	2	4	4	4
conv layer 8	1	2	2	2	4
conv layer 9	N/A	1	2	2	2
conv layer 10	N/A	1	2	2	2
conv layer 11	N/A	N/A	1	1	2
conv layer 12	N/A	N/A	1	1	2
conv layer 13	N/A	N/A	1	1	1
conv layer 14	N/A	N/A	N/A	N/A	1
conv layer 15	N/A	N/A	N/A	N/A	1
conv layer 16	N/A	N/A	N/A	N/A	1
fc layer 1	1	1	1	1	1
fc layer 2	1	1	1	1	1
fc layer 3	1	1	1	1	1

Figure 3.7: Weight Replication of Each VGG.



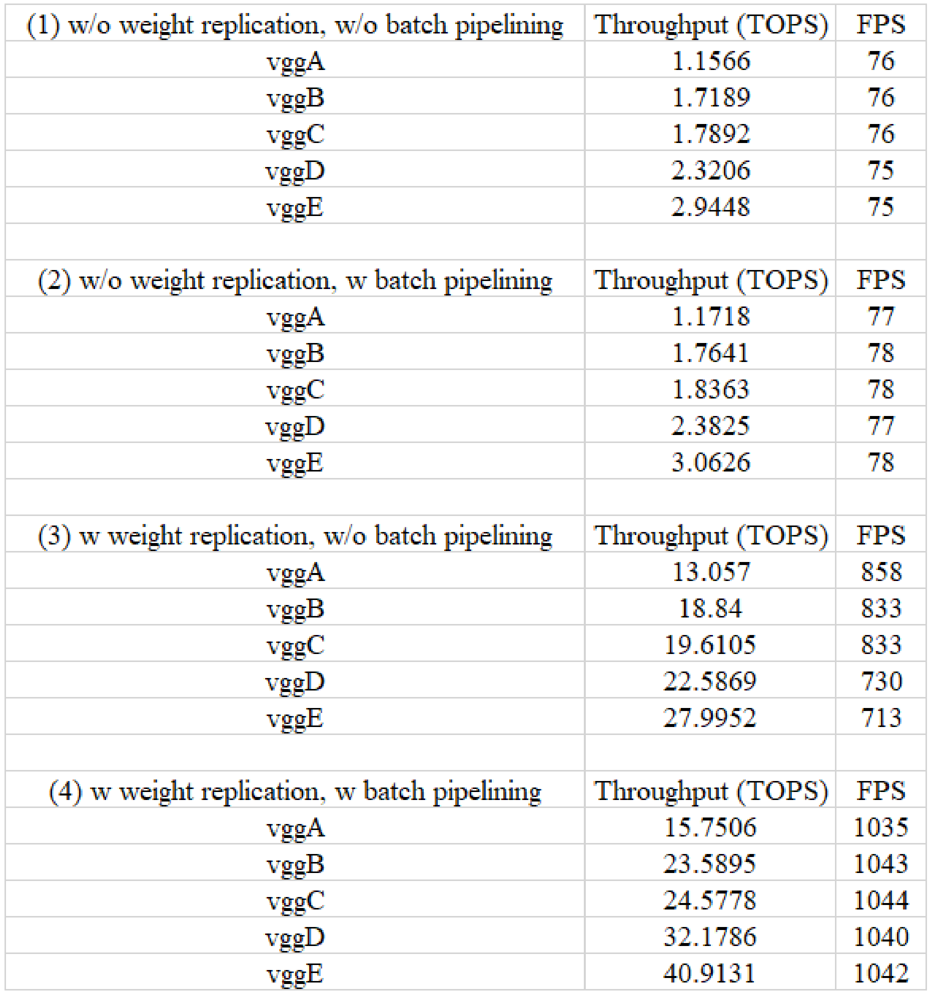
	vggA tiles	vggB tiles	vggC tiles	vggD tiles	vggE tiles
un-replicated	129	131	136	158	185
replicated	174	198	208	256	304

Figure 3.8: Total Number of Tiles of Each VGG.

meet the constraint that there are a maximum of 320 tiles available.

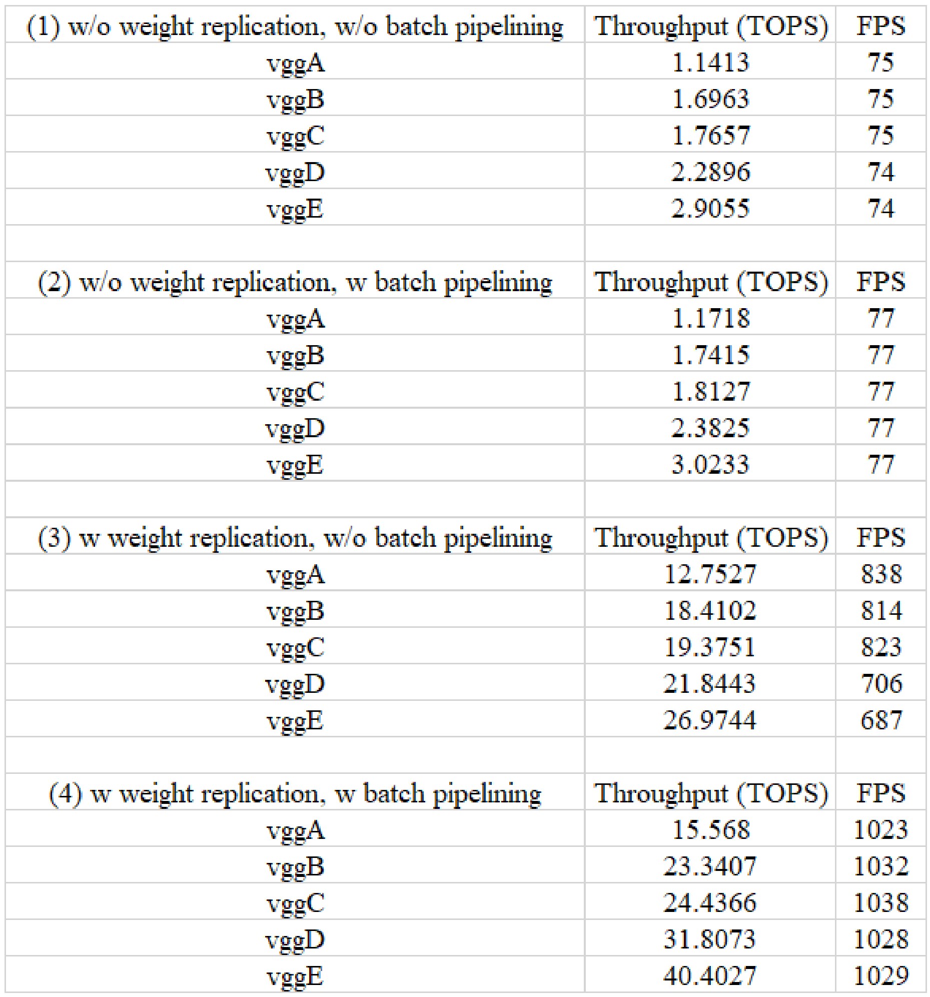
3.4 Performance Results

The workloads include $3 \times 4 \times 5 = 60$ cases. There are five different CNNs: VGG (A-E), three different NoCs: ideal, SMART, wormhole, and four different pipelining scenarios:



(1) w/o weight replication, w/o batch pipelining	Throughput (TOPS)	FPS
vggA	1.1566	76
vggB	1.7189	76
vggC	1.7892	76
vggD	2.3206	75
vggE	2.9448	75
(2) w/o weight replication, w batch pipelining	Throughput (TOPS)	FPS
vggA	1.1718	77
vggB	1.7641	78
vggC	1.8363	78
vggD	2.3825	77
vggE	3.0626	78
(3) w weight replication, w/o batch pipelining	Throughput (TOPS)	FPS
vggA	13.057	858
vggB	18.84	833
vggC	19.6105	833
vggD	22.5869	730
vggE	27.9952	713
(4) w weight replication, w batch pipelining	Throughput (TOPS)	FPS
vggA	15.7506	1035
vggB	23.5895	1043
vggC	24.5778	1044
vggD	32.1786	1040
vggE	40.9131	1042

Figure 3.9: Performance of Each VGG using Ideal NoC.



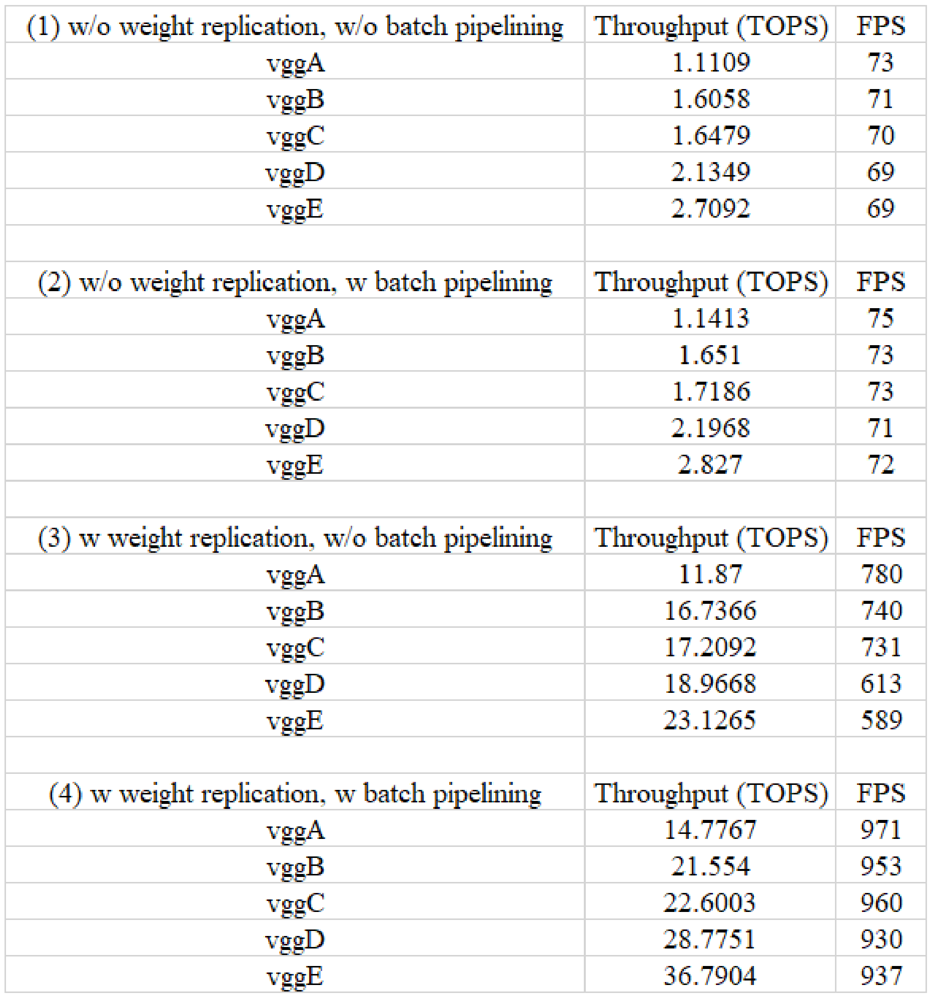
(1) w/o weight replication, w/o batch pipelining	Throughput (TOPS)	FPS
vggA	1.1413	75
vggB	1.6963	75
vggC	1.7657	75
vggD	2.2896	74
vggE	2.9055	74
(2) w/o weight replication, w batch pipelining	Throughput (TOPS)	FPS
vggA	1.1718	77
vggB	1.7415	77
vggC	1.8127	77
vggD	2.3825	77
vggE	3.0233	77
(3) w weight replication, w/o batch pipelining	Throughput (TOPS)	FPS
vggA	12.7527	838
vggB	18.4102	814
vggC	19.3751	823
vggD	21.8443	706
vggE	26.9744	687
(4) w weight replication, w batch pipelining	Throughput (TOPS)	FPS
vggA	15.568	1023
vggB	23.3407	1032
vggC	24.4366	1038
vggD	31.8073	1028
vggE	40.4027	1029

Figure 3.10: Performance of Each VGG using SMART NoC.

without weight replication and without batch pipelining (1), without weight replication and with batch pipelining (2), with weight replication and without batch pipelining (3), with weight replication and with batch pipelining (4). For the processing side, we build a cycle-accurate simulator from scratch in C++. For the interconnect side, we simulate the performance of the NoC in the cycle-accurate simulator garnet2.0 using trace-based input.

Fig. 3.9 shows the throughput (TOPS) and frame rate (FPS) of the four scenarios for each VGG using ideal NoC. Fig. 3.10 shows the throughput (TOPS) and frame rate (FPS) of the four scenarios for each VGG using SMART NoC. Fig. 3.11 shows the throughput (TOPS) and frame rate (FPS) of the four scenarios for each VGG using wormhole NoC.

To explore the effect of different pipelining schemes on the performance, we calculate



(1) w/o weight replication, w/o batch pipelining	Throughput (TOPS)	FPS
vggA	1.1109	73
vggB	1.6058	71
vggC	1.6479	70
vggD	2.1349	69
vggE	2.7092	69
(2) w/o weight replication, w batch pipelining	Throughput (TOPS)	FPS
vggA	1.1413	75
vggB	1.651	73
vggC	1.7186	73
vggD	2.1968	71
vggE	2.827	72
(3) w weight replication, w/o batch pipelining	Throughput (TOPS)	FPS
vggA	11.87	780
vggB	16.7366	740
vggC	17.2092	731
vggD	18.9668	613
vggE	23.1265	589
(4) w weight replication, w batch pipelining	Throughput (TOPS)	FPS
vggA	14.7767	971
vggB	21.554	953
vggC	22.6003	960
vggD	28.7751	930
vggE	36.7904	937

Figure 3.11: Performance of Each VGG using Wormhole NoC.

the speedup of each scenario by normalizing the throughput to scenario (1). Fig. 3.12 shows the speedup in all four scenarios for each VGG in all three different NoCs. The geometric mean of (2) compared to (1) is $1.0309\times$, (3) compared to (1) is $10.1788\times$, and (4) compared to (1) is $13.6903\times$. Note that for the best pipelining setup in scenario (4), it achieves a speedup close to $16\times$. We don't need to replicate the weights in all layers by 16 times to achieve this speedup. Instead, we replicate weights decreasingly as the layers become deeper and the size of OFM decreases to make a balanced pipeline design. Note that the results in Fig. 3.12 are projected results, which are not directly from garnet2.0.

To explore the effect of different NoCs on the performance, we calculate the speedup of all three NoCs (ideal, SMART, wormhole) by normalizing the throughput to worm-

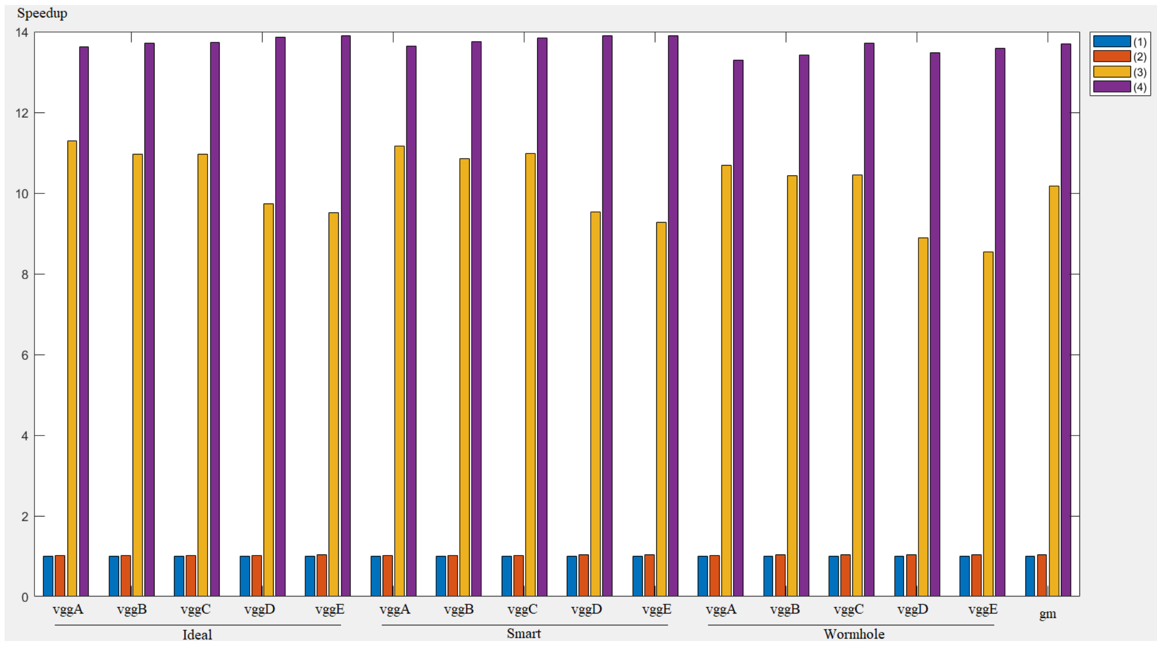


Figure 3.12: Speedup of Each VGG due to Different Pipelining.

hole. Fig. 3.13 shows the speedup in all three NoCs for each VGG in all four pipelining scenarios. The geometric mean of ideal compared to wormhole is $1.0809\times$ and SMART compared to wormhole is $1.0965\times$. Note that SMART NoC achieves better speedup for

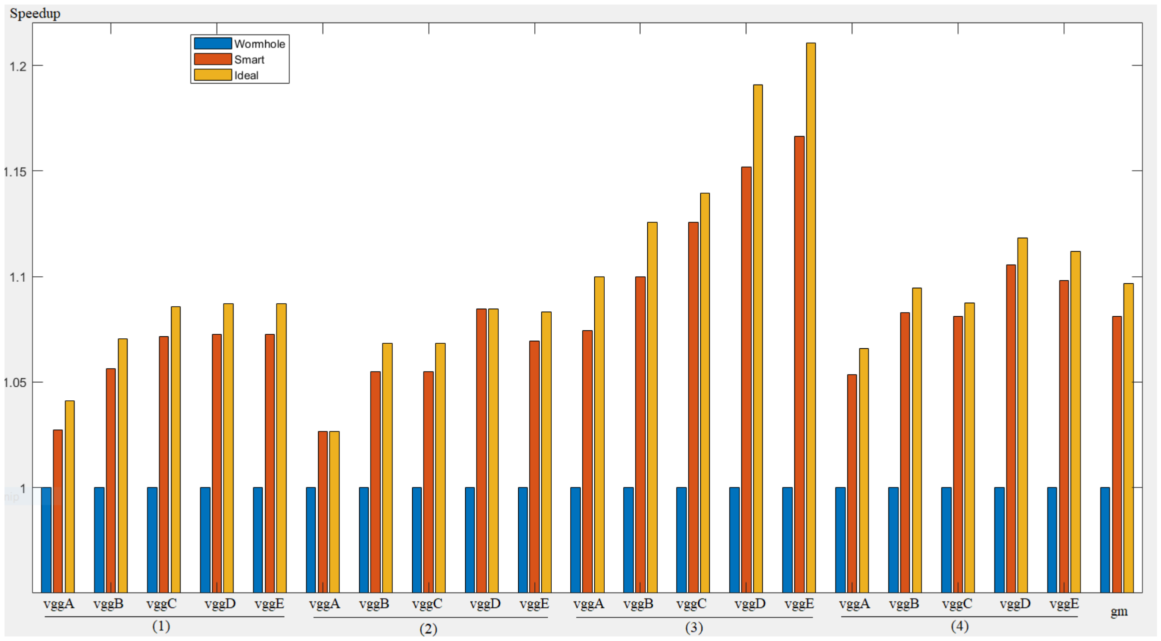
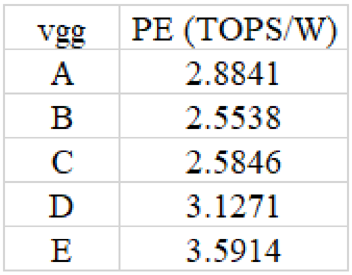


Figure 3.13: Speedup of Each VGG due to Different NoC.



vgg	PE (TOPS/W)
A	2.8841
B	2.5538
C	2.5846
D	3.1271
E	3.5914

Figure 3.14: Energy Efficiency of Each VGG.

more aggressive pipelining because more aggressive pipelining has heavier traffic in the NoC, so the performance of SMART NoC improves effectively while the performance of wormhole NoC degrades performance even more. Note that the results in Fig. 3.13 are projected results, which are not directly from garnet2.0.

In addition, we also report the energy efficiency (TOPS/W) for processing each VGG, as shown in Fig. 3.14. Note that weight replication, batch pipelining, and different flow control algorithms don't affect energy efficiency much because with the total amount of energy consumed depends mostly on the amount of operations in the workload.

CHAPTER 4
CONCLUSION

This research work presents a design of an analog ReRAM-based PIM (processing-in-memory) architecture for fast and efficient CNN (convolutional neural network) inference. For the overall architecture, we use the basic hardware hierarchy such as node, tile, core, and subarray. On the top of that, we design intra-layer pipelining, inter-layer pipelining, and batch pipelining to further exploit parallelism in the architecture and increase overall throughput for the inference of an input image stream. Our simulator also optimizes the performance of the NoC (network-on-chip) using SMART (single-cycle multi-hop asynchronous repeated traversal) flow control. Finally, we experiment with weight replications for different CNN layers and report throughput, energy efficiency, and speedup of VGG (A-E) for large-scale data set ImageNet.

4.1 Observation

For ASIC designs for AI/ML/NN acceleration, NoC still represents a small portion of bottleneck within PIM because most computations are done within a tile and inter-tile communications are not quite often. Therefore, most speedup is achieved from the processing side by designing efficient pipelining and leveraging weight replications.

4.2 Future Work

ReRAM-based PIM is a very good architecture for ASIC designs for AI/ML/NN applications, but how to increase its programmability and reconfigurability can be further explored. One idea to make ReRAM-based architecture more generate-purpose for all kinds of NNs is to design domain-specific ISAs for ReRAM-based architectures.

REFERENCES

[1] Z. Zhang, "Derivation of backpropagation in convolutional neural network (cnn)," *University of Tennessee, Knoxville, TN*, 2016.

[2] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, IEEE, 2017, pp. 19–24.

[3] 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 *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2017, pp. 1–12.

[4] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai, "Metal–oxide rram," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.

[5] G. W. Burr, R. M. Shelby, S. Sidler, C. Di Nolfo, J. Jang, I. Boybat, R. S. Shenoy, P. Narayanan, K. Virwani, E. U. Giacometti, *et al.*, "Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element," *IEEE Transactions on Electron Devices*, vol. 62, no. 11, pp. 3498–3507, 2015.

[6] A. F. Vincent, J. Larroque, W. Zhao, N. B. Romdhane, O. Bichler, C. Gamrat, J.-O. Klein, S. Galdin-Retailleau, and D. Querlioz, "Spin-transfer torque magnetic memory as a stochastic memristive synapse," in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2014, pp. 1074–1077.

[7] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy, *et al.*, "Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2019, pp. 715–731.

[8] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[9] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *ACM SIGARCH Computer Architecture News*, IEEE Press, vol. 44, 2016, pp. 27–39.

[10] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2017, pp. 541–552.

[11] M. Cheng, L. Xia, Z. Zhu, Y. Cai, Y. Xie, Y. Wang, and H. Yang, "Time: A training-in-memory architecture for memristor-based deep neural networks," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ACM, 2017, p. 26.

[12] T. Krishna, "Enabling dedicated single-cycle connections over a shared network-on-chip," PhD thesis, Massachusetts Institute of Technology, 2014.

[13] T. Krishna, C.-H. O. Chen, W. C. Kwon, and L.-S. Peh, "Breaking the on-chip latency barrier using smart," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2013, pp. 378–389.

[14] I. Pérez, E. Vallejo, and R. Beivide, "Smart++ reducing cost and improving efficiency of multi-hop bypass in noc routers," in *Proceedings of the 13th IEEE/ACM International Symposium on Networks-on-Chip*, 2019, pp. 1–8.

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

[16] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.