TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Electrical Engineering and Information Technology

Chair of Highly-Parallel VLSI Systems and Neuro-Microelectronics

Diploma Thesis

with Topic

Efficient Mapping of Convolutional Neural Networks on SpiNNaker2 Prototype

Author	Binyi Wu
Course of Study	Informationstechnik, Jg. 2015
Matrikel-Nr.	4571474
Date of birth	12.05.1991 in Guangdong China

Supervisor:	Dipl.-Ing. Florian Kelber
	Dipl.-Phys. Bernhard Vogginger
Examiner:	Prof. Dr.-Ing. habil. Christian Mayr
	Dr.-Ing. Sebastian Höppner
Day Of Submission:	16.05.2019

TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Electrical Engineering and Information Technology

Chair of Highly-Parallel VLSI Systems and Neuro-Microelectronics

Abstract

The 2nd generation SpiNNaker (SpiNNaker2) [1] system contains 144 machine learning (ML) accelerators and 144 ARM M4F cores for the efficient computation of deep neural networks (DNNs). The ML accelerator composes of a multiply-accumulate (MAC) array which can be used to calculate 2D convolutions or matrix multiplications.

In this thesis, a SpiNNaker2 simulator written in Python, SpiNNaker2Py, and different parallelization strategies for mapping convolutional neural networks (CNNs) onto the ARM cores and ML accelerators were developed to estimate the overall processing clocks and memory utilization on inference phase as well as analysis the performance bottlenecks of SpiNNaker2. The mapping strategies involve how to chain several layers, how to decompose each layer into multiple operations, split them into numerous small tasks and distribute them on SpiNNaker2 efficiently. The state-of-the-art neural network model, VGG [2] and residual network [3], are applied for the work.

Furthermore, during the validation of SpiNNaker2Py, some of HDL design flaws of SpiNNaker2 are discovered.

Keywords: SpiNNaker2, machine learning accelerator, SpiNNaker2Py, convolutional neural networks, inference, parallelization strategy

Statement of Authorship

I hereby certify that I have authored this Diploma thesis entitled *Efficient Mapping of Convolutional Neural Networks on SpiNNaker2 Prototype* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. They were no additional persons involved in the spiritual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, May 16, 2019 .....

Binyi Wu

Acknowledgements

I thank my supervisors Dipl.-Ing. Florian Kelber and Dipl.-Phys. Bernhard Vogginger for their constant help and patient guidance. During the graduation project, they discussed a lot with me, let me familiar with SpiNNaker2 step by step. And they also gave me a lot of advice on thesis writing. In this project, I learned a lot of new knowledge and once again thanked my supervisors for encouraging and accepting me to use my ideas for the project. Finally, I would like to thank all the scholars who have made valuable research in related fields.

Contents

List of Figures

List of Tables

List of Abbreviations

CNNs	Convolutional Neural Networks
CONV	convolutional layer
DMA	Direct Memory Access
DNNs	Deep Neural Networks
DRAM	Dynamic Random Access Memory
FC	fully-connected layer
GIL	Global Interpreter Lock
HDL	Hardware Description Language
LOSS	loss layer
MAC	Multiply-Accumulate/MultiplierAccumulator
MLA	Machine Learning Acclerator
ML	Machine Learning
NoC	Network-on-Chip
PE	Process Element
POOL	pooling layer
QPE	Quad PE
ReLU	Rectified Linear Unit
ReLU	Rectified Linear Units layer
SNN	Spiking Neural Network
SRAM	Static Random Access Memory

1 Introduction

Deep neural networks (DNNs) are currently widely applied to image recognition, natural language processing, and other fields, achieving near or even beyond-human performance. Because of their high computational complexity, especially for the convolutional neural network (CNN), which is a class of DNNs, much specific hardware has emerged. SpiNNaker2 (the 2nd generation of SpiNNaker), dedicated to spiking neural networks (SNNs), could also be employed to accelerate DNNs with an add-on machine learning accelerator (MLA).

SpiNNaker2 [1] has 144 processing elements (PE), which contains one machine learning accelerator (MLA) and one ARM M4F core, for the efficient computation of neural networks. Because the state-of-art CNNs are quite large and each PE in SpiNNaker2 have not enough memory to hold one layer of the networks, moreover in order to fully utilize all the computing resources, each layer of networks needs to be decomposed into small tasks. After decomposition, three different distribution algorithms for how to allocate them into 144 PEs, are developed and compared. On this basis, what the performance bottlenecks of SpiNNaker2 are and how serious they are will be analyzed against the overall processing clocks. In addition to this, memory utilization is also one of the research targets. Furthermore, the overall attainable performance of SpiNNaker2 using MLAs with different number of MAC units will be studied to seek for a higher power efficient system. In the thesis, the state-of-art models, VGG-16 [2] and ResNet-50 [3], are employed throughout all simulations and experiments to estimate the performance of SpiNNaker2.

As the SpiNNaker2 is still under development, only Quad PE (QPE) unit, which composes of four PEs and one network on chip (NoC), is available. Therefore, a SpiNNaker2 simulator (SpiNNaker2Py), written in Python, is developed to accomplish the research mentioned above. To ensure the cycle accuracy of the SpiNNaker2Py, the existing QPE Verilog model is used to validate the cycle accuracy of SpiNNaker2Py's QPE.

This thesis is organized as follows. Chapter 2 provides an introduction of convolutional neural networks and an overview of the architecture of SpiNNaker2. Chapter 3 introduce the software SpiNNaker2 simulator SpiN-

Naker2Py. Chapter 4 illustrates how to map CNNs into QPE or SpiNNaker2. The mapping strategy includes the solutions for how to chain several layers, how to decompose each layer into multiple operations, split them into multiple small tasks and how to distribute them on QPE or SpiNNaker2 efficiently. Chapter 5 mainly focuses on how to validate the feasibility of the simulator and the decomposition scheme, the simulations results as well as the experiments for MLA with different computing power. The last chapter summarizes this thesis.

2 Convolutional Neural Networks and SpiNNaker2

Convolutional neural network (CNN) and SpiNNaker2 are the two research objects of the thesis. In Section 2.1, the network architecture and various layers in CNN are introduced. Section 2.2 gives a brief explanation of two state-of-art CNN models, VGG-16 [2] and ResNet-50 [3]. Whereas the last section 2.3 illustrates SpiNNaker2's architecture and the characteristics of its components.

2.1 Convolutional Neural Networks

CNN is a class of DNNs and is widely applied to visual imagery processing, for example, image classification, object detection and so on. The input of CNN usually is an image. Its output is various for different purposes (e.g., for image classification, it is a class label).

A typical CNN has multiple layers. The input layer usually is the raw pixel values of the image. The hidden layers could contain the convolutional layer (CONV), non-linearity layer (ACTI), pooling layer (POOL) and fully-connected layer (FC) [4]. The output layer is always a fully-connected layer (FC) with a non-linearity layer. Figure 2.1 shows the architecture of typical CNN.

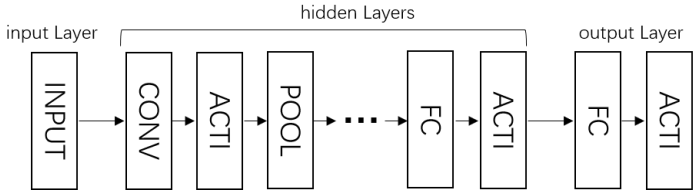

Figure 2.1: Architecture of CNN

As recent research implies that the DNN can be computed with low precision numerical represented data, which decreases the amount of data transfer and accelerate the computation as well as makes it possible to run DNN in

devices without floating point unit. In order to apply the low precision computation in DNN, an additional layer named quantization layer (QUAN), need to be added after the non-linearity layer [5]. For CNN, its typical architecture with quantization is shown in Figure 2.2.

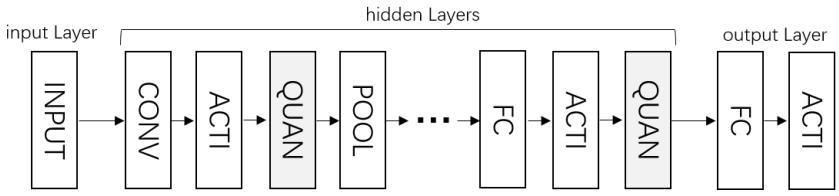

Figure 2.2: Architecture of CNN with quantization

2.1.1 Convolutional Layer

The convolutional layer is the core part with the heaviest computation of a CNN. The input and output of convolutional layer are both feature map. In the convolutional layer, there are two operations: padding and convolution. The padding is priors to the convolution.

The padding operation is used to retain the border information of the input feature map, which helps to preserve the spatial size and enable the network to be deeper. Zero-padding, which pads the border with zeros, is the most popular padding method [4].

The convolution operation has two multi-dimension operands. One is called feature map, which could be an image for an input layer or an output of the previous layer for a hidden layer. Another operand is called filter weight, which is learnable and is used to extract more profound and more abstract features from the input feature map. The extraction is realized by the convolution operation [4].

Both operands of convolution operation are 3-dimensions with $[W,H,D]$, where W, H, and D represent width, height, and depth respectively. The D of both operands must be the same. The other two dimensions of the filter weight are always smaller than those of the feature map. Besides this, the convolution involves the concept of stride S, which represents the number of pixels the filter weight moves during computation. Convolution can be decomposed into three steps:

1. do multiplier-accumulator (MAC) operation over the filter weight and part of the feature map captured by the filter weight;

2. sliding the filter weight according to the stride;

3. repeating the above two steps until the filter weight traverses the entire feature map.

If it is assumed that the dimension of the input feature map $\mathbf{I}$ and the filter weight $\mathbf{F}$ are $[W_i, H_i, D]$ and $[W_f, H_f, D]$ respectively. Then the output feature map $\mathbf{O}$ is $[W_o, H_o, 1]$, where $W_o = \frac{W_i - W_f}{S} + 1$, $H_o = \frac{H_i - H_f}{S} + 1$. The padding size P of each border is $P = \frac{W_i - W_o}{2} = \frac{H_i - H_o}{2}$. Given the dimension parameter, the convolution can be defined as equation 2.1 and is shown in Figure 2.3.

$$\mathbf{O}[w][h] = \sum_{k=0}^{W_f} \sum_{j=0}^{H_f} \sum_{i=0}^{D} \mathbf{I}[Sw + k][Sh + j][i] * \mathbf{F}[k][j][i] \tag{2.1}$$

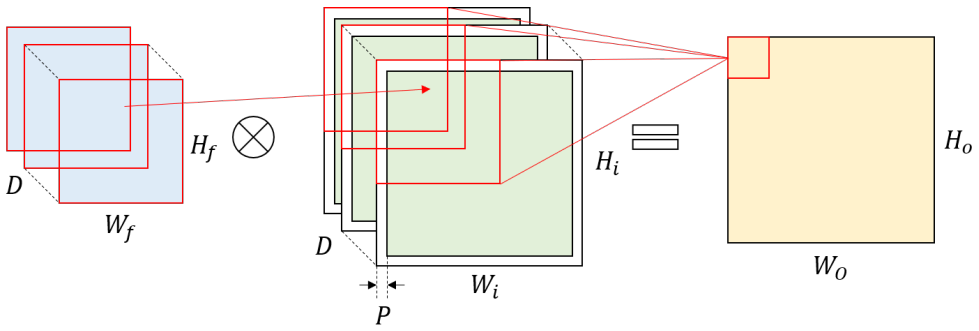

Figure 2.3: Convolution

If there are multi-filters, then the shape of filter weight is represented as $[W_f, H_f, D, C]$, where C represents the number of filters in filter weight. Correspondingly, the shape of output feature map $\mathbf{O}$ is $[W_o, H_o, C]$. Equation 2.1 turns to be equation 2.2

$$\mathbf{O}[w][h][c] = \sum_{k=0}^{W_f} \sum_{j=0}^{H_f} \sum_{i=0}^{D} \mathbf{I}[Sw + k][Sh + j][i] * \mathbf{F}[k][j][i][c] \tag{2.2}$$

There is a lot of data reuse during convolution [6]. One is called filter-weight-reused, where the filter weight is reused for each sliding. Another is called feature-map-reused, where each filter reuses the feature map. Furthermore, convolution operation has a high computational intensity (at least

tens, usually hundreds), given by equation 2.3, which causes the computing power of a computing unit is much more critical than its memory bandwidth.

$$\begin{aligned} computational\_intensity_{conv} &= \frac{2 * W_f * H_f * D * W_o * H_o * C}{W_i * H_i * D + W_f * H_f * D * C} \\ &= \begin{cases} 2 * W_f * H_f * C, & W_i * H_i \gg W_f * H_f * C \\ W_f * H_f * C, & W_i * H_i \approx W_f * H_f * C \\ 2 * W_o * H_o, & W_i * H_i \ll W_f * H_f * C \end{cases} \end{aligned} \tag{2.3}$$

2.1.2 Non-Linearity Layer

The non-linearity layer is also referred to as the activation layer, which is used to introduce non-linearity into the neural networks. The non-linearity layer always appears with a convolution layer or a fully-connected layer. The rectifier is the most commonly used nonlinear functions. It has been proven that it brings more performance improvements than other non-linearity function [7]. The definition of rectified linear units (ReLU) is $f(x) = max(0,x)$, as shown in Figure 2.4 [4].

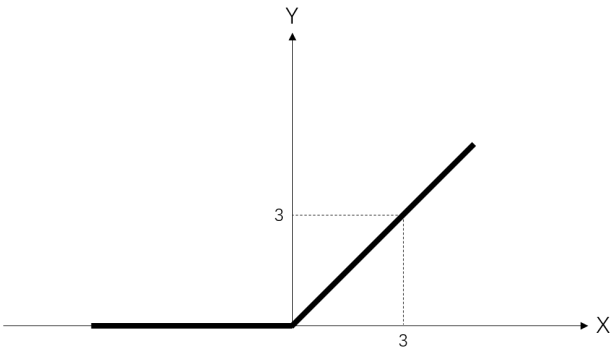

Figure 2.4: Rectified linear units (ReLU)

2.1.3 Pooling Layer

The pooling layer is standard in CNN, especially in large neural network models. It aims to progressively reduce the spatial size of the feature map, decreasing the computation amount in the next layer and the number of parameters needed to be trained over the whole neural network. The input

and output of the pooling layer are both feature maps. The pooling layer contains padding operation and pooling operation.

The pooling operation operates on every channel (depth) of the input feature map independently and resizes every channel. Although every channel is operated independently, the same pooling filter must be applied over them, ensuring every channel has the same size after resizing [4]. The pooling operation also has parameter stride S. The maximum (MAX) pooling is the most commonly used pooling operation. Beside MAX pooling, there are still other operations, like average operation. Pooling can be decomposed into three steps:

1. applying MAX operation over part of the feature map captured by the pooling filter;

2. sliding the pooling filter according to the stride;

3. repeating the above two steps until the pooling filter traverses the entire feature map.

If the dimensions of the input feature map and pooling filter are $[W_i, H_i, D]$ and $[W_{pf}, H_{pf}]$ respectively, the shape of the output feature map is $[W_o, H_o, D]$, where $W_o = \frac{W_i - W_{pf}}{S} + 1$ and $H_o = \frac{H_i - H_{pf}}{S} + 1$. An example of a pooling with pooling size of $[W_{pf}, H_{pf}] = [2,2]$ and $S = 2$ is shown in Figure 2.5.

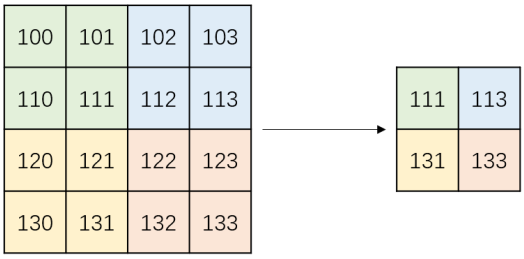

Figure 2.5: Example of MAX pooling

2.1.4 Fully-connected Layer

Different from the convolution layer's local connectivity, connecting neurons to part of the neurons in the previous layer, each neuron in the fully-connected layer is connected to all the neurons in the previous layer. Because of more complicated connections, the fully-connected layer is more complex

than the convolution layer, having more parameters, and thus is more difficult to train, and easier suffered from overfitting [4].

The computation of fully-connected layers is matrix multiplication. The output of the previous layer is reshaped into a vector with dimensions $[W_A, H_A] = [P,1]$ firstly, then multiplied by a weight matrix with dimension $[W_B, H_B] = [Q,P]$ and finally summed with a bias with dimension $[W_C, H_C] = [Q,1]$, where H_x and W_x respectively represent the height and the width. Figure 2.6 shows an example of computing a fully-connected layer.

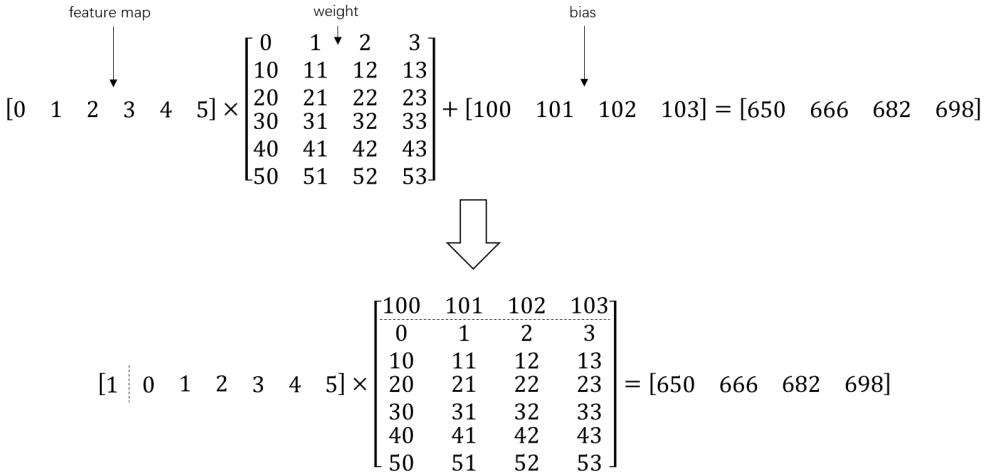

Figure 2.6: Example of computing a fully-connected layer

Matrix multiplication has less data reuse than convolution. Its computational strength is very low, given by equation 2.4, which makes the memory access bottleneck more prominent.

$$computational\_intensity_{mm} = \frac{P * Q * 2}{P + P * Q} \approx 2 \tag{2.4}$$

2.1.5 Quantization Layer

A quantization layer is used to convert the data with high precision representation to be the data with low precision. If a quantized network works on data with 8-bit, after the convolutional layer with the non-linearity layer or the fully-connected layer with the non-linearity layer, the result is represented by value with more than 8 bits, e.g., 32 bits. The quantization layer is used to convert the result into 8-bit again, letting the next layer can be

computed with the 8-bit data. The quantization operation can be realized by bit shifting [5].

2.2 VGG-16 and ResNet-50

In this section, two state-of-art CNN models, VGG-16 [2] and ResNet-50 [3], will be briefly introduced.

The VGG was developed in 2014 with a very uniform architecture, which proves that deeper networks with smaller filters can lead to better results for image classification. The name, VGG-16, indicates that there is a total of 16 convolutional layers and fully-connected layers. Figure 2.7 shows the architecture of VGG-16.

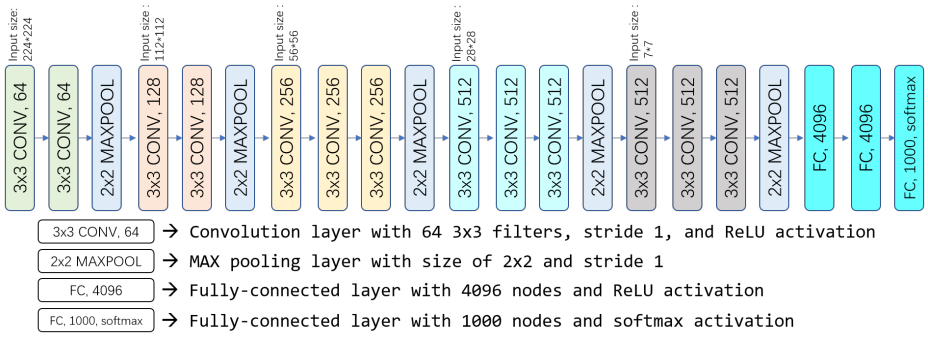

Figure 2.7: VGG-16

The ResNet, shorted for residual neural network, was proposed in 2015. In ResNet, shortcut connections are introduced, enabling the neural network to be much deeper. There are several variations of ResNet. The ResNet used in the thesis is with 50 layers. Figure 2.8 reveals its structure.

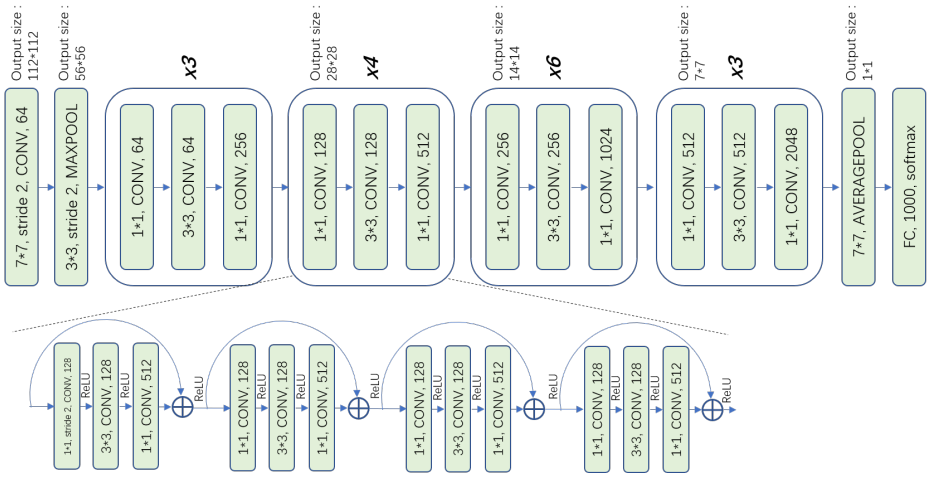

Figure 2.8: ResNet-50

2.3 SpiNNaker2

SpiNNaker2, the 2nd generation of SpiNNaker, is a multi-core SoC, which is dedicated to simulating the spiking neural networks (SNNs). With add-on machine learning accelerators (MLAs), it can also be employed to accelerate the neural network.

Since SpiNNaker2 was still under development, and due to the simple to complex development concept, the following description of the spinnaker architecture could be different from the latest version.

In the following sections, we will introduce the SpiNNaker2 in a bottom-up approach.

2.3.1 Processing Element (PE)

The basic building block of SpiNNaker2 is the processing element (PE) as shown in Figure 2.9. In a PE, there is an ARM M4F processor, a machine learning accelerator (MLA), a 128 KByte SRAM, a direct memory access unit (DMA), and a neuromorphic mathematic unit (NMU). The ARM M4F processor is the control unit of the PE as well as the computation unit whereas the MLA is used to accelerate matrix multiplication and convolution. To increase data bandwidth, four 32 KByte SRAMs make up a 128 KByte SRAM, providing 128 bits of data bandwidth. The 128 KByte SRAM is used for storing instructions and data. The neuromorphic mathematic unit (NMU) is a neuromorphic accelerator for computing logarithm and exponent

as well as generating random numbers. With dynamic voltage and frequency scaling technology, the supply voltage and frequency can be adjusted based on the calculated load to provide a more energy efficient system [8][1].

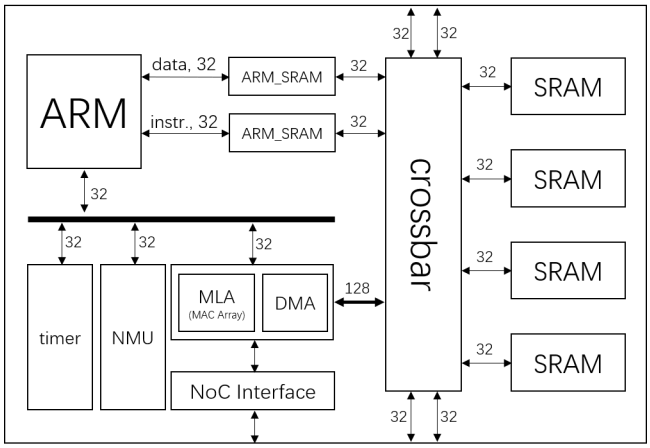

Figure 2.9: Architecture of PE. The number on the arrow represents the bit width.

2.3.2 Machine Learning Accelerator (MLA)

The ML accelerator (MLA) contains a 16 x 4 MAC array as shown in Figure 2.10. The operand A, which fetches 128-bit data each time from SRAM or NoC (referred to section 2.3.3), is 32 bits. Each MAC row (16 MAC units) shares each 8-bit value of the operand A. The operand B, which fetches 128-bit or 32-bit data each time from the SRAM, is 128 bits and each 8-bit value is shared by one MAC column(4 MAC units). To preserve accuracy, the output of each MAC unit is 29 bits. After computation, the MLA will store the calculated results into the SRAM row by row, and each clock outputs 128-bit data, where the 29 bits result of the MAC unit is converted to 32 bits [8].

Convolution In convolution operation, the operand A can fetch data from any PE SRAM through the NoC. However, even if it fetches data from the local PE SRAM, the fetching process still needs to be done through the NoC. Different from the operand A, the operand B fetches data from its local PE SRAM and does not need to be through the NoC.

The operand A is responsible for fetching filter weight, and each 8-bit value is from one filter. As there are 32 bits, MAC array is capable of computing

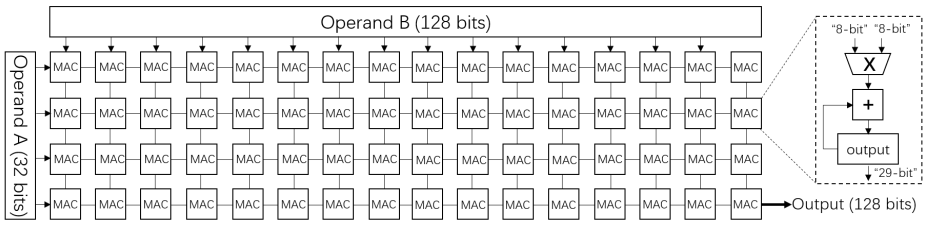

Figure 2.10: Architecture of MAC Array. Each MAC unit performs one multiplication and one addition in one clock cycle.

the number of rows of MAC array filters (four filters) at one time. The operand B is responsible for fetching input feature maps. At the beginning of fetching one row of the input feature map, the operand B fetches 128-bit data in one clock cycle. After this, the operand B only fetches 32 bits data, and the 32 bits data is shifted into the operand B in four times (each time 8 bits data), which limits the MAC array to only support convolution with $stride = 1$.

The dimension of the MAC array causes data alignment. The size of the input feature map $size_{ifmap\_align}$, the size of the filter weight $size_{filters\_align}$ and the size of the output feature map $size_{ofmap\_align}$ after alignment are expressed by equation 2.5.

$$\begin{cases} size_{ifmap\_align} = align_{16}(W_i) * H_i * D \\ size_{filters\_align} = align_{16}(W_f * H_f * D * align_{rows}(C)) \\ size_{ofmap\_align} = align_{4'}(W_o) * H_o * C \end{cases} \tag{2.5}$$

where $align_{16}()$, $align_{rows}()$ and $align_{4'}()$ respectively represent alignment to 16 bytes (128 bits), alignment to the number of rows bytes and alignment to 4 32-bit (128 bits).

Figure 2.11 shows an example of the convolution computing process in MLA.

Matrix Multiplication In matrix multiplication operation, the operand A can fetch data from any PE SRAM through the NoC. However, different from in convolution operation, if data comes from local PE SRAM, the NoC is not needed anymore. Same as in convolution, operand B must fetch data from its local PE SRAM.

When the local SRAM of PE is the source of the operand B and at least one operand A, the SRAM bandwidth will become the performance bottleneck

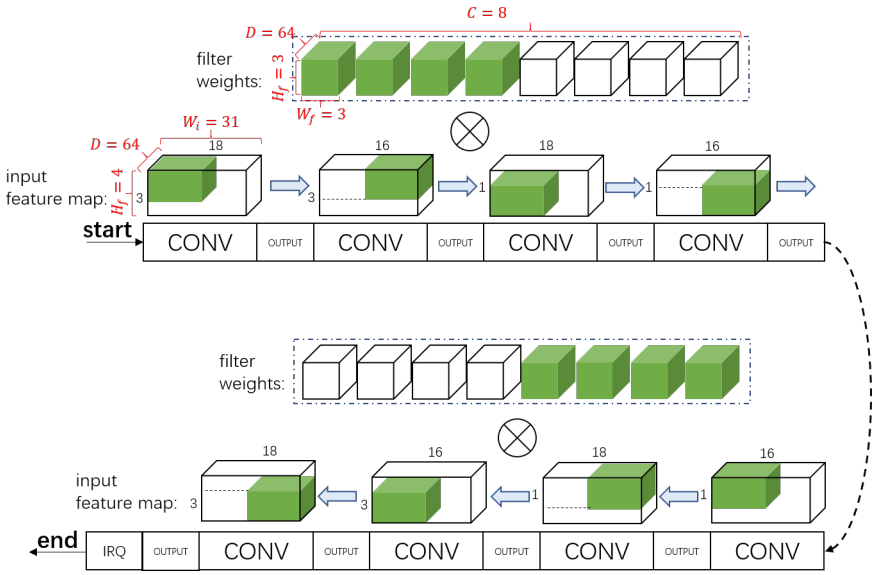

Figure 2.11: Example of the convolution process in MLA with the input feature map $[W_i,H_i,D] = [31,4,64]$, the filter weight $[W_f,H_f,D,C] = [3,3,64,8]$ and stride $S = 1$. The **CONV** stage represents the execution of the convolution whereas the **OUTPUT** stage represents outputting the results to the local SRAM. The green parts of the input feature map and filter weights are the currently used data for the convolution stage. MLA finishes the convolution with a total of 8 stages of **CONV** and **OUTPUT**. After then, the MLA generates an interrupt request (IRQ) to the ARM core.

of the MLA. Because when the MAC array is running with full utilization, 128-bit data is needed from the SRAM for operand A every four clocks and 512-bit data is needed for the operand B every four clocks. However, the SRAM can only maximum supply 512-bit data in 4 clocks, which cannot meet the required $512 + 128 = 640$ bits.

If it is assumed that the MLA does matrix multiplication: $A * B = C$, where A, B, and C are the matrix, the operand A is responsible for fetching rows from A. Each 8-bit channel of operand A corresponds to one column of A. As there are 32 bits, MAC array is capable of computing four rows of A at one time. The operand B is responsible for fetching columns from B. Each 8-bit channel of the operand B corresponds to one column of B. As there are 128 bits, MAC array is capable of computing 16 columns of A at one time.

The dimension of the MAC array causes data alignment. The size of the matrix A $size_{A\_align}$, the size of the matrix B $size_{B\_align}$ and the size of the

matrix C $size_{C\_align}$ after alignment are expressed by equation 2.6.

$$\begin{cases} size_{A\_align} = align_4(W_A) * align_{rows}(H_A) \\ size_{B\_align} = align_{16}(W_B) * align_4(H_B) \\ size_{C\_align} = align_{16}(W_C) * align_{rows}(H_C) \end{cases} \tag{2.6}$$

Figure 2.11 shows the computing process of a matrix multiplication.

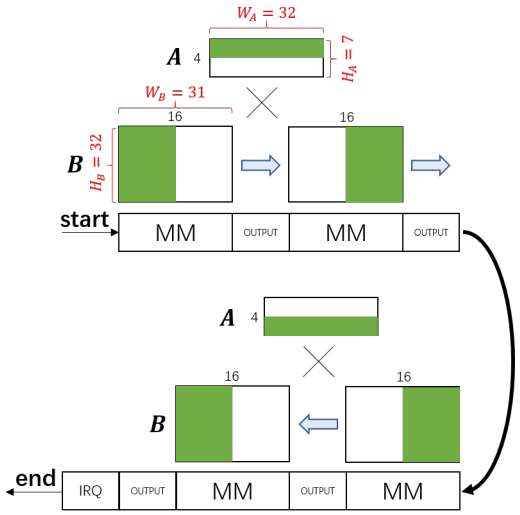

Figure 2.12: Example of the matrix multiplication process in MLA with $A[7, 32]$ and $B[32, 31]$. The **MM** stage represents the execution of the matrix multiplication whereas the **OUTPUT** stage represents outputting the results to the local SRAM. The green parts of matrix A and B are the currently used data for the matrix multiplication stage.

2.3.3 Network-on-Chip (NoC)

The network-on-chip (NoC) is a network-based communications subsystem used in SpiNNaker2, enabling the communication between different components. However, the NoC in SpiNNaker2 does not support broadcast or multicast. The NoC in SpiNNaker2 contains two independent NoCs. Data NoC (DNoC) is mainly used for transferring data with high performance while configuration NoC (CNOC) is purposed for configuration with low performance. Each PE is only connected to the DNoC via an asynchronous FIFO interface, with which DNoC and PE can work on different frequencies. The NoC also connects to other NoCs, building a mesh network. The NoC processes a NoC packet per clock but with four clocks delay. The block diagram of the NoC is shown in Figure 2.13 [8].

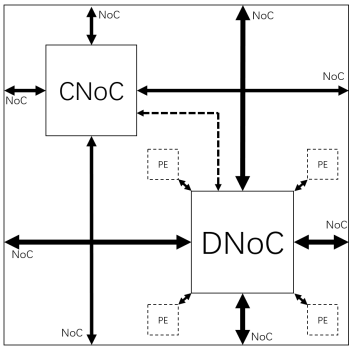

Figure 2.13: Architecture of NoC

2.3.4 Quad Processing Element (QPE)

Figure 2.14 shows the block diagram of QPE. Each QPE contains 4 PEs and 1 NoC router. Communication between PEs is done by the NoC[8].

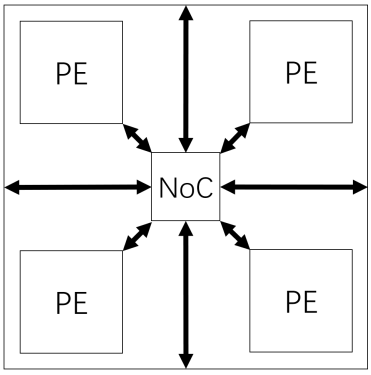

Figure 2.14: Architecture of QPE

With the building block QPE, the brief block diagram of SpiNNaker2 is shown in Figure 2.15. SpiNNaker2 has a total of 36 QPEs, four DRAM interfaces, and one host interface. Communications between QPEs and QPEs as well as QPEs and DRAMs are through the NoC. An NoC packet takes four clocks from QPE to neighbor QPE, whereas seven clocks from DRAM to neighbor QPE[8].

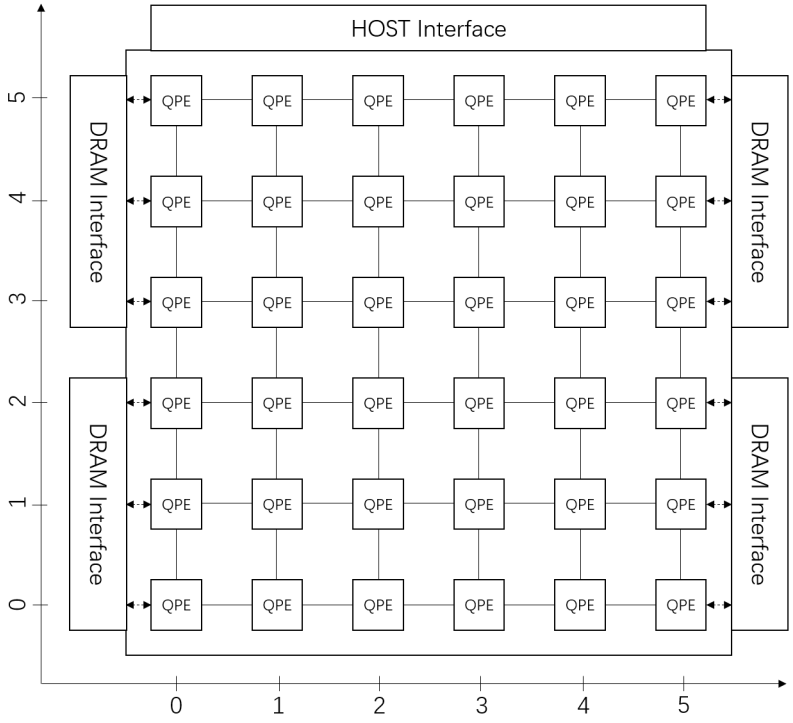

Figure 2.15: Architecture of SpiNNaker2

3 SpiNNaker2 Simulator: SpiNNaker2Py

As mentioned before, SpiNNaker is still under development. In order to develop efficient mapping algorithms and estimate their performances, a relatively high and straightforward precision simulator is needed. Moreover, a software-based simulator is more flexible, more comfortable to modify, and faster to find more reasonable hardware configurations for later hardware development.

The simulator SpiNNaker2Py is written in Python 3.6.4 and uses only built-in libraries. With object-oriented thinking, it is conducive to future maintenance. In addition to this, the multi-process library is used to speed up its simulation.

Next, we introduce the simulator architecture step by step in a bottom-up order.

3.1 Processing Element (PE) Simulator

The PE simulator simulates the PE in section 2.3 and includes an ARM core simulator, a machine learning accelerator (MLA) simulator, an SRAM simulator, a DMA simulator and an interface connecting to NoC, but without NMU. All the simulators have the same behavior as their hardware specifications. Figure 3.1 shows the block diagram of the PE simulator.

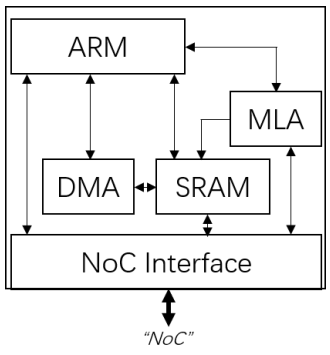

Figure 3.1: Block diagram of PE simulator

For each simulator, there is an executive function, calling it once is equivalent to running the corresponding hardware one clock. Furthermore, each simulator has a built-in task queue, when calling the execution function, the current task will be fetched from the task queue. If the task queue is empty, the corresponding simulator will be idle. Calling the executive function of the PE simulator once will trigger the execution of all the built-in simulators by once, which means that all simulators in the PE simulator operate at the same frequency.

The communication between components is realized by the signal packet. To handle the transmission of the signal packet, each simulator has a unique ID, which is included in the signal packet. The NoC packet is also one of the signal packets. Different packets contain different tasks and are handled differently. Generally, there are three types of signal packets: single-cycle single-task signal packets, single-cycle multi-tasking signal packets, and multi-cycle signal packets.

A single-cycle single-task signal packet means that it contains a single task and its processing time is one clock. This packet could be an interrupt signals from MLA or DMA, a single SRAM reading or writing request and an NoC packet route request.

Whereas a single-cycle multi-task signal packet is also processed in one clock, but multiple individual packets are generated and placed in the task queue of the corresponding simulator. A typical representation of this signal packet is the DMA request. When the ARM core sends the DMA task to DMA, the DMA converts the packet to be multiple single SRAM operation requests into its task queue within one clock. The DMA then sends the single SRAM operation requests to the SRAM one by one.

The multi-cycle single packet is a convolution or a matrix multiplication task of the MLA. When MLA receives the packet, it performs the task until the corresponding task is finished.

3.2 Quad PE (QPE) Simulator

Similar to the QPE in section 2.3, the QPE simulator contains four PE simulators and one NoC router simulator, as shown in Figure 3.2. The NoC simulator routes the signal packets from PEs and other NoCs through the packet destination indicated by the simulator ID. For NoC, it also has an executive function, which processes one packet per call. According to the NoC specification, the NoC simulator is with four clocks delay. Calling

the QPE simulator's execution function will automatically run the four PE simulators and the NoC simulator by once. By default, when the QPE runs once, the PE runs also once, but the NoC runs twice, which means that the NoC simulator is twice as frequent as the PE simulator.

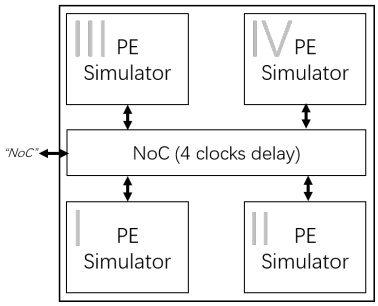

Figure 3.2: Block diagram of QPE simulator

3.3 QPE-DRAM Simulator

The QPE-DRAM simulator composes of one QPE and one DRAM. Figure 3.3 shows its block diagram. The host interface is the interface used to communicate with the host (e.g., distributor) and has a latency of 10 clocks. The DRAM simulator, whose frequency is equal to that of PE, has a delay of 7 clocks sending data to NoC.

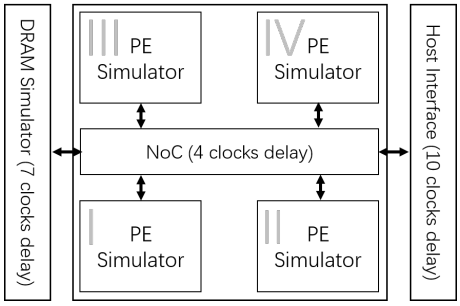

Figure 3.3: Block diagram of QPE-DRAM simulator

3.4 SpiNNaker2 Simulator

Figure 3.4 is the block diagram of the SpiNNaker2 simulator, which is according to the architecture of SpiNNaker2 in section 2.3. However, in the SpiNNaker2 simulator, by default, there are four host agent NoCs, which work as a traffic transfer station. On the one hand, all requests from the host will be transferred from the host interface to the agent NoCs, and the agent QPE-NoCs route them into corresponding QPEs or DRAMs. On the other hand, all response from QPEs or DRAMs will be transferred to the corresponding agent NoCs and redirected by the agent NoCs to the host through the host interface. Each agent NoC is responsible for one triple-QPE-block (3 x 3 QPE array). By default, QPE(2,2), QPE(3,2), QPE(2,3) and QPE(3,3) are respectively responsible for triple-QPE-block I, II, III and IV.

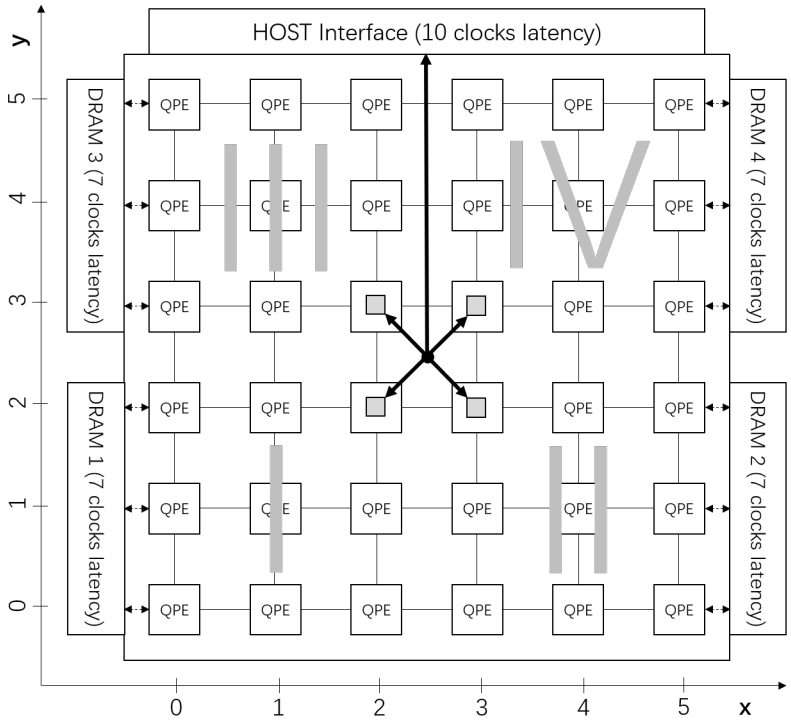

Figure 3.4: Block diagram of SpiNNaker2 simulator

Multiple Process Because all QPEs in SpiNNaker2 run in parallel, the SpiNNaker2 simulator can be accelerated by introducing multiple threads or

multiple processes. In the actual test, the multi-threading cannot improve the simulation speed but instead reduces it, due to the global interpreter lock (GIL) [9] feature of Python and the fact that the simulation is a CPU-intensive task. Therefore the multi-processing is applied to the simulator. Currently, this simulator is accelerated by five processes, as shown in Figure 3.5.

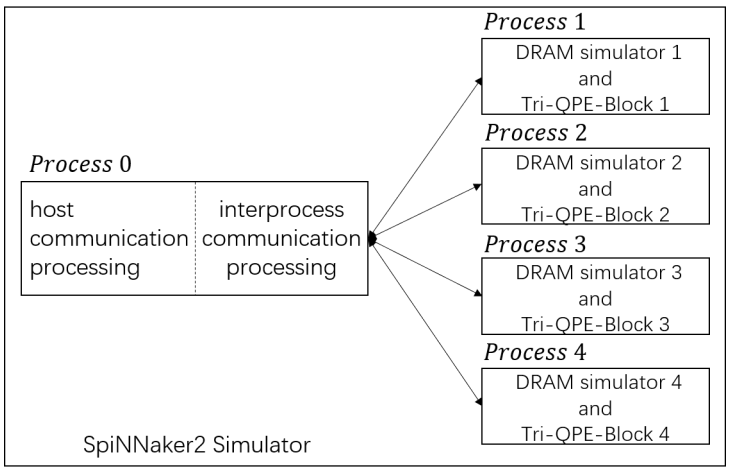

Figure 3.5: Multi-processing of SpiNNaker2

4 Mapping Strategy

In order to map the state-of-art neural networks into SpiNNaker2. The following three issues need to be addressed.

1. CNN has several types of layers and each of them contains a variety of operations, such as that a convolutional layer includes padding and convolution. In order to fit the compute primitives of SpiNNaker2, the neural network needs to be parsed into the basic computation supported by SpiNNaker2.

2. The state-of-art neural network is very large. For example, the shape of the input feature map, the filter weight and the output feature map of the first convolutional layer of VGG-16 are respectively $[W_i, H_i, D] = [224, 224, 3]$, $[W_f, H_f, D, C]$ = [3,3,3,64] and $[W_o, H_o, C] = [224, 224, 64]$. (After padding, the shape of the input feature map becomes $[W_i, H_i, D] = [226, 226, 3]$.) In order to accelerate the convolution through the MLA in SpiNNaker2, they must follow the data alignment illustrated in section 2.3.2. According to equation 2.5, the size of the input feature map, the filter weight and the output feature map after alignment are respectively 162720 Bytes, 1728 Bytes, and 12845056 Bytes. The total needed size is $162720 + 1728 + 12845056 \approx 12.41 MByte$. However, a PE has only 128 KBytes. In order to use SpiNNaker2 to accelerate this convolution, it need to be decomposed into multiple pieces, which fit the available SRAM. In fact, even though the SRAM is big enough to hold the whole layer, in order to utilize as much computing resource as possible, the convolution should also be decompose into multi-pieces.

3. Even the neural network is decomposed. An efficient distribution algorithm is needed to utilize the computing resource of SpiNNaker2 fully.

A mapper, as shown in Figure 4.1, which contains parser, splitter, and distributor, is created for performing the mapping strategies and addressing the issues mentioned above. The idea of mapper comes from Chen et al. [10].

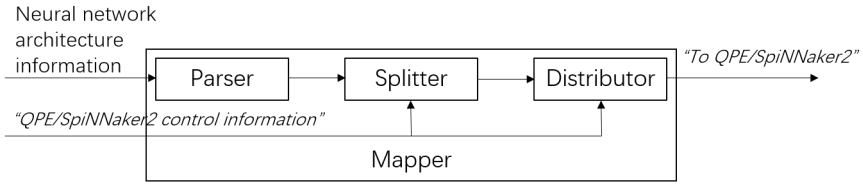

Figure 4.1: Block diagram of mapper

The input of the mapper is the neural network architecture information, which contains the kernel information of each layer. The data structure of the layer information is shown in Figure 4.2.

The data structure of each layer is *(layer_type, layer_parameter)*.
layer_type could be one of convolutional layer, pooling layer and fully-connected layer.
layer_parameter is specific to difference type of layer.

A convolutional layer with

```
[filter_width, filter_height, in_channel, out_channel]=[3,3,3,64],
                                    stride=1,
                                padding_type=SAME,
                                non-linearity=ReLU
```

is represented as

```
(LayerType.CONV, ([3, 3, 3, 64], 1, PaddType.SAME, ActivationType.ReLU)).
```

A pooling layer with

```
                        [filter_width, filter_height]=[2,2],
                                    stride=2,
                                pooling_type=MAXPOOL
```

is represented as

```
                    (LayerType.POOL, ([2,2], 2, PoolType.MAXPOOL)).
```

A fully-connected layer with

```
                                output_size=4096,
                                non-linearity=ReLU
```

is represented as

```
                    (LayerType.FC, (4096, ActivationType.ReLU)).
```

Figure 4.2: Example of layer information for Mapper

The parser, introduced in section 4.1, is used to address the first issue, parsing the neural network models to achieve inter-layer optimization, while in section 4.2, we explain how the splitter (for the 2nd issue) decomposes each layer to obtain intra-layer optimization. Lastly, for the 3rd issue, the distributor is developed to allocate the sub-tasks generated by the splitter to the corresponding simulator to achieve acceleration. Due to the difference in hardware characteristics between the QPE-DRAM simulator and

SpiNNaker2 simulator, the splitter and distributor performs the different strategies, whereas the parser applies the same strategy. Section 4.3 and section 4.4 are for the distributor of the QPE and SpiNNaker2, respectively.

4.1 Parser

The parser firstly attaches layers to layers, linking the output of the previous layer to the input of the next layer. Each layer parameter is converted into a full layer parameter, which contains the dimension of the input/output feature map. Furthermore, layer fusion, which is paving the way for operator fusion, is realized in this stage. An example of layer chaining and layer fusion is shown in Figure 4.3. In this step, the rationality of the network will be checked (e.g., if the input channel of the output feature map is equal to the input channel of the filter weight in the next layer).

If it is assumed that a convolutional layer is
`(LayerType.CONV, ([3, 3, 3, 64], 1, PaddType.SAME, ActivationType.ReLU))`,
and its previous layer's output feature map dimension is
`[output_width, output_height, output_channel, out_channel]=[224,224,3]`,
and its next layer is pooling layer with
`(LayerType.POOL, ([2, 2], 2, PoolType.MAXPOOL))`.
After chaining , the convectional layer's layer parameter becomes
`(Layer_type, ((input, convolution weight, convolution stride, output), (pool weight, pooling stride))) =`
`(LayerType.CONV, (([226, 226, 3], [3, 3, 3, 64], 1, [224, 224, 64]), ([2,2],2)))`.

Figure 4.3: Example of layer chaining and layer fusion

After filling the network information, the parser will break down each layer into several primitive operations, which fit the compute primitives of the accelerator. From section 2.3, it is known that SpiNNaker2 is capable of matrix multiplication (done by MAC array), convolution (done by MAC array) and element-wise operation (done by ARM core). In order to match these compute primitives of the accelerator, each layer of the neural network must be decomposed into these primitive operations. The decomposition of a convolutional layer, a pooling layer, and a fully-connected layer is shown in Figure 4.4.

Figure 4.5 is an example of how to decompose a convolutional layer, which decomposes a convolutional layer into a padding operation, a convolution operation, a non-linearity operation, and a quantization operation.

After layer decomposition, the parser will then reassemble all the decomposed operations into different operation blocks. The process is called operator fusion, which aims to decrease the data transfer between SRAM and

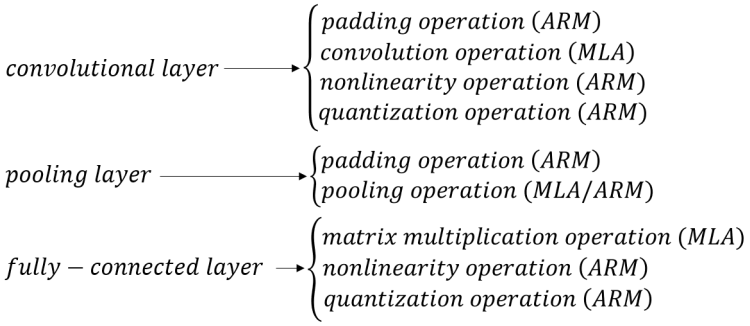

Figure 4.4: Decomposition of convolutional layer, pooling layer and fully-connected layer

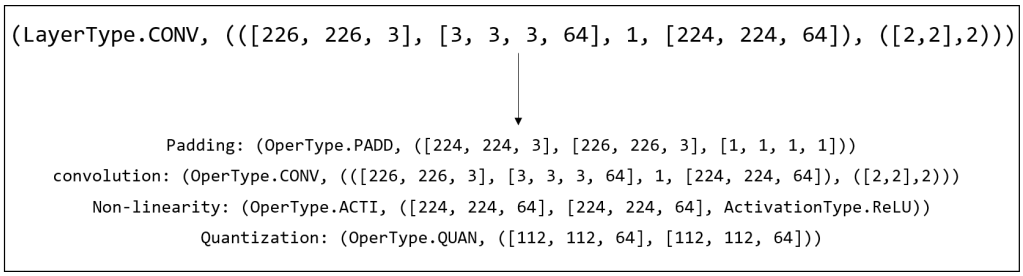

Figure 4.5: Example of decomposing convolutional layer

DRAM (while the computed result is stored in DRAM) or between SRAM and SRAM (while the computed result is stored in SRAM). When the former operation in an operation block is finished, the result will not be transferred out of the PE but left for the next operation in the operation block. Only when all the operations in the operation block are finished, the result is transferred out of PE. In addition to operator fusion, operator reorder, which changes the order of operations and thus can decrease the computation amount, can also be employed in some cases.

The operator fusion can be realized in one layer or multi-layers with the help of layer fusion mentioned before. Figure 4.6 shows the operator fusion of a convolutional layer, whereas Figure 4.7 shows the operator fusion of a convolutional layer and pooling layer when pooling filter's stride is equal to its width and height. These two operation blocks above are called convolution block, and the core operation is the convolution operation. When the pooling filter's stride is not equal to its width or height, the pooling layer cannot be fused into a convolution block and self forms a pooling block, whose core operation is the pooling operation, as shown in Figure 4.8. The operator fusion of a fully-connected layer is called matrix multiplication block, and

its core operation is the matrix multiplication operation. Figure 4.9 shows a matrix multiplication block.

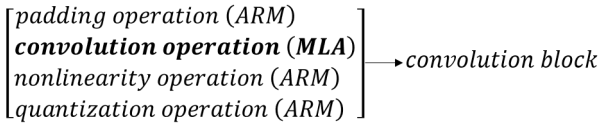

Figure 4.6: Operator fusion of convolutional layer

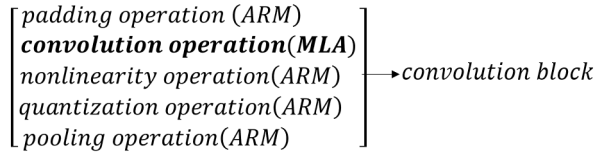

Figure 4.7: Operator fusion of convolutional layer and pooling layer

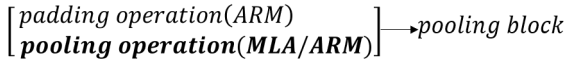

Figure 4.8: Operator fusion of pooling layer

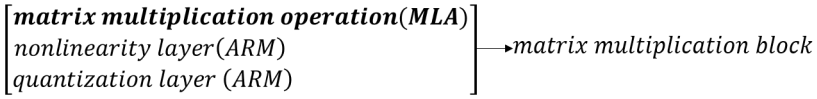

Figure 4.9: Operator fusion of fully-connected layer

4.2 Splitter

Based on the parse result from the parser, the splitter searches the best split scheme for the QPE-DRAM or SpiNNaker2. In the splitter, the core operation in the operation block is the primary processing object. In another word, the splitter splits the core operation, and the other primitive operations in the operation block follow its decomposition scheme. Furthermore, other primitive operations will be considered if necessary when splitting the core operation. In the following, we will explain how to split the core operation in units of operation blocks.

4.2.1 Convolution Block

For the convolution block, the core operation is the convolution operation, because the convolution operation requires the most storage space and takes the most of computation time. Moreover, if the pooling operation is included in the convolution block, the pooling operation is considered when splitting the convolution operation.

The convolution operation is very computationally intensive. In other words, the computational resource is much more important than memory bandwidth. Therefore, the basic rule of searching the split scheme for convolution is that keeping the maximum utilization of the MAC array comes first. This guarantees that the computation is accelerated as quickly as possible.

A convolution has two operands, input feature map and filter weight, as well as an output. The dimensions of the input feature map, filter weight and output feature map is $[W_i, H_i, D]$, $[W_f, H_f, D, C]$ and $[W_o, H_o, C]$ respectively, where W_i, H_i, W_f, H_f, D, C, W_o, and H_o respectively represent width of input feature map, height of input feature map, width of filter weight, height of filter weight, input channel, output channel, width of output feature map, and height of output feature map. It is possible to decompose the convolution operation along every dimension into multiple parts. However, splitting along different dimensions has different effects as shown in Table 4.1.

Dimension	Disadvantages
W_i	increase size of input feature map difficult to split due to overlapping
H_i	increase size of input feature map difficult to split due to overlapping
D	move part of computation work from MLA to ARM
W_f	move part of computation work from MLA to ARM
H_f	move part of computation work from MLA to ARM
C	difficult to reuse input feature map
W_o	increase size of input feature map
H_o	increase size of input feature map

Table 4.1: Effect by splitting convolution along different dimension

Splitting along W_f, H_f, D will result in a complete single filter being split into multiple parts and eventually need to accumulate their convolution result together to obtain the final result, which causes an additional element-wise operation. Since MLA cannot be used to speed up the element-wise addition operation, this operation can only be done by ARM, which means splitting a complete single filter will cause that parts of calculation are moved from MLA to ARM. As ARM has no advantage in computing performance against MLA, this splitting method will decrease the overall acceleration speed. Furthermore, considering that W_f and H_f are relatively small and SpiNNaker2 has enough storage space, splitting along them will not be the choice.

When the convolution stride is smaller than the width and height of filter, splitting along W_i and H_i will result in increasing the data amount of the input feature map. As after splitting, the split input feature maps overlap with each other. Splitting along W_o and H_o is the same as along W_i and H_i, because the dimension of input feature map is related to the output feature map via the filter weight. However, splitting along W_o and H_o makes the decomposition easier, since the split output feature maps do not overlap each other. Therefore, W_o and H_o are the split dimensions in the splitting process rather than W_i and H_i. After splitting, through the filter dimension and convolution stride, the dimension of the split input feature map can be obtained using equation 4.1.

$$Dim_i = (Dim_o - 1) * S + F \tag{4.1}$$

where Dim_i, Dim_o, F and S are the corresponding dimension size of the

input feature map, the output feature map, the filter weight and the stride.

Splitting along C is to regroup all filters into multiple filter clusters, which will not shift calculations to ARM or even increase the data amount of input feature map. However, it makes it difficult to reuse the input feature map.

In summary, this should be the best dimension order the splitter could follow to decompose a big convolution into multiple small convolutions: C, W_o/H_o, D. Figure 4.10 shows the splitting process.

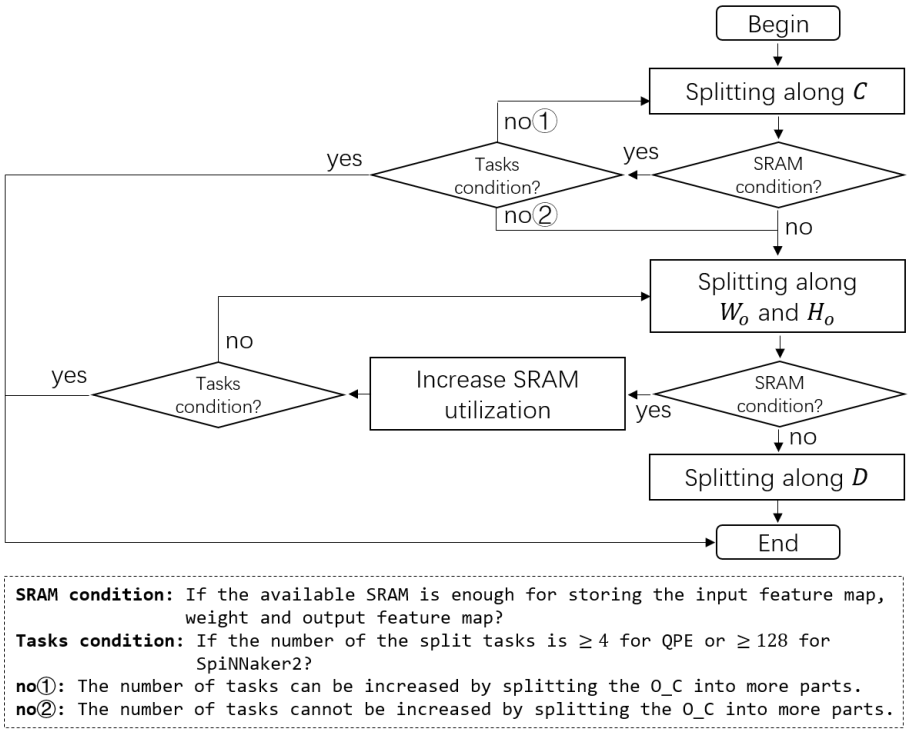

Figure 4.10: Convolution splitting process

When splitting along one dimension cannot meet the available SRAM limitation, the optimized split result along the dimension will be retained, and the next split step begins with this split result. The final split scheme is that the split scheme meets the SRAM limitation and the convolution is decomposed into at least 4 parts for QPE or 128 parts for SpiNNaker2.

For the dimension C, if the meted requirement split scheme could not be found, the optimized split result for the next step is that the filter weight is split into parts with $rows\_of\_MLA$ output channel (By default, $rows\_of\_MLA = 4$).

Splitting along W_o and H_o is the most complicated step. It not only determines how much data amount is increased but also decides how well the distributor could utilize the QPE-DRAM or SpiNNaker2. Splitting along W_o and H_o, therefore, is the critical step and several rules are set.

(1) If the pooling operation is fused into the convolution block, the pooling stride must be considered when splitting convolution.

(2) The MAC array utilization must be maximized.

(3) The splitter should choose a split scheme that does not increase the amount of data too much. See Appendix A for how to avoid increasing the data amount too much.

(4) The splitter must consider the computation balance problem, ensuring that all PEs have the same computation amount, which is essential in data reuse mode.

If the meted requirement split scheme could not be found, and it is assumed that the largest size of of the split input feature map, the split filter weight, and the split output feature map are $size_{ifmap}$, $size_{filter}$ and $size_{ofmap}$ respectively, then the optimized split result is the one whose $size_{ifmap} + size_{filter} + size_{ofmap}$ is closest to $N * available\_sram$, where N is an integer, a factor of D and as small as possible. This optimized split result ensures its relatively high SRAM utilization in subsequent steps. Because in the next step (splitting along D), dimension D can only be divided into integer parts.

If a decomposition scheme could not be found after splitting along C, W_o and H_o, the splitter is going to decompose D into N parts. However, splitting along D will increase the computation amount by the ARM core, which causes a bottleneck on the ARM core. Because of this, the splitter will try to avoid splitting along D.

After splitting, if it is assumed that X, Y, Z are the number of parts along C, W_o/H_o and D respectively, the large convolution operation is decomposed into $X * Y * Z$ small convolution operations. For SpiNNaker2, it should be as close to $2^P * 128$ as possible (P is an integer and $P >= 0$), which ensures the distributor to utilize SpiNNaker2 as much as possible. For more detail information, please refer to section 4.4.

4.2.2 Pooling Block

For the pooling block, the pooling operation is the split object, and its splitting scheme is followed by other primitive operations in the pooling block.

Because the pooling operation is not heavy on computation and splitting the input feature map increases the data amount, the splitter carries on the guideline that decomposing the input feature map into $N * 4$ parts for QPE-DRAM or $N * 128$ parts for SpiNNaker2, where N is an integer and as small as possible.

The pooling has only one operand and one output: input feature map and output feature map. The dimensions of them are the same as that for convolution, but $D = C$. Splitting along different dimensions has different effects. Splitting along W_i/H_i or W_o/H_o has the same effect as that for convolution; However, because pooling operation is a channel-independent operation, splitting along D has no disadvantages.

In summary, the best dimension order, which the splitter could follow to decompose a pooling operation, is D, then W_o/H_o. Figure 4.11 shows the splitting process.

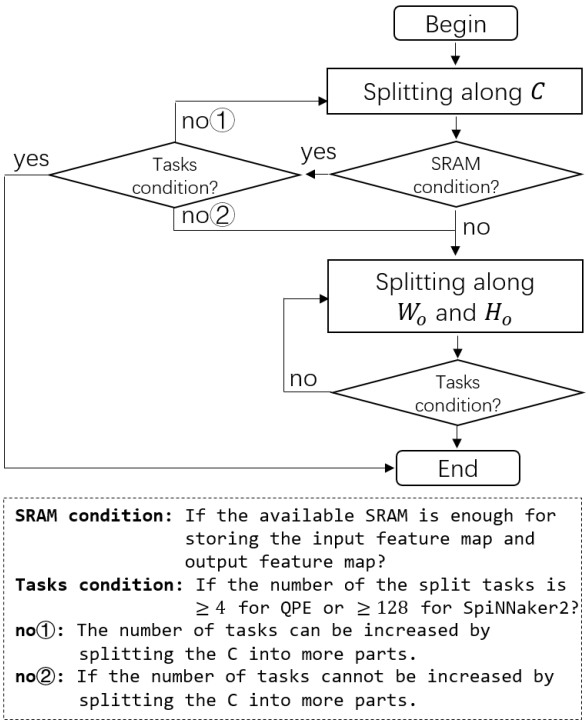

Figure 4.11: Pooling splitting process

Same as for the convolution splitting process, the optimized split result along the dimension will be retained for the next split step if its decomposition scheme does not meet the available SRAM limitation.

When splitting along D could not meet the available SRAM limitation, it will be decomposed into parts with one channel per part for the next split step.

Splitting along W_o/H_o is more complicated. The splitter should choose a split scheme that does not increase the amount of data too much (See Appendix A for more detail information). Furthermore, it decides how well the distributor could utilize the QPE-DRAM or SpiNNaker2. After splitting, if it is assumed that X, Y are the number of parts along D, W_o/H_o respectively, the large pooling operation is decomposed into $X * Y$ small pooling operations, and the value $X * Y$ should be equal to $N * 4$ for QPE-DRAM or $N * 128$ for SpiNNaker2.

4.2.3 Matrix Multiplication Block

The matrix multiplication operation is the split object of the matrix multiplication block.

The matrix multiplication operation is a low computationally intensive operation with only 2 operations per data, which means memory bandwidth is much more critical than the computational resource. The guideline of searching the split scheme for matrix multiplication is that full utilization of the memory bandwidth comes first.

Matrix multiplication has two operands, input M_A and weight M_B, as well as an output M_C. The dimension of them are $[W_A, H_A]$, $[W_B, H_B]$ and $[W_C, H_C]$ respectively, where W_A, H_A, W_B, H_B, W_C and H_C] represent width of matrix M_A , height of matrix M_A , width of matrix M_B, height of matrix M_B, width of matrix M_C and height of matrix M_C respectively, and $W_A = H_B$, $H_A = H_C$, $W_B = W_C$. The splitting can only be done along H_A, W_B and H_B.

For the fully-connected layer with that batch size is 1, $H_A = H_C = 1$. If it is assumed to split the corresponding matrix multiplication into N parts, this could be done by splitting along W_B and H_B.

Splitting along W_B causes the incrementation of the data amount of the input, whereas splitting along H_B will cause an additional element-wise operation, which can only be accelerated by the ARM core. Therefore, there should be a trade-off between splitting along W_B and H_B. However, because both QPE and SpiNNaker2 have enough computing resources against memory bandwidth for matrix multiplication, in order to improve the speed of computing, the splitting scheme that increases the data amount should be suppressed (Please refer to Appendix B for more detail information). Con-

sequently, the splitter tends to split along H_B as more parts as possible.

4.3 Distributor for QPE-DRAM

When the parser and the splitter finish, the distributor for QPE-DRAM distributes the tasks into QPE-DRAM. The distribution strategy is developed and improved based on the flow shown in Figure 4.12. The distribution strategy begins with the simplest one. Once the distribution result is obtained, then it is analyzed to find out the performance bottlenecks. Solving the bottlenecks will obtain a new, improved distribution strategy based on the old one.

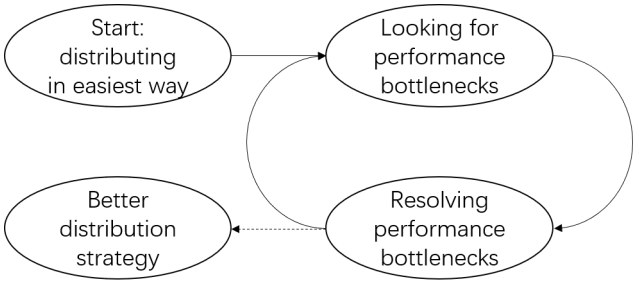

Figure 4.12: How to develop/improve distribution strategy

4.3.1 Convolution Block

For the convolution block, three distribution strategies has been developed:

- without operator fusion and without data reuse,
- with operator fusion and without data reuse,
- with operator fusion and with data reuse.

All of them try to utilize MAC array and PEs as high as possible, while the difference lays on that the latter two are trying to minimize the time on data transfer, which is realized with operator fusion or data reuse.

(i) Without operator fusion and data reuse Figure 4.13 shows this distribution process of a convolution block.

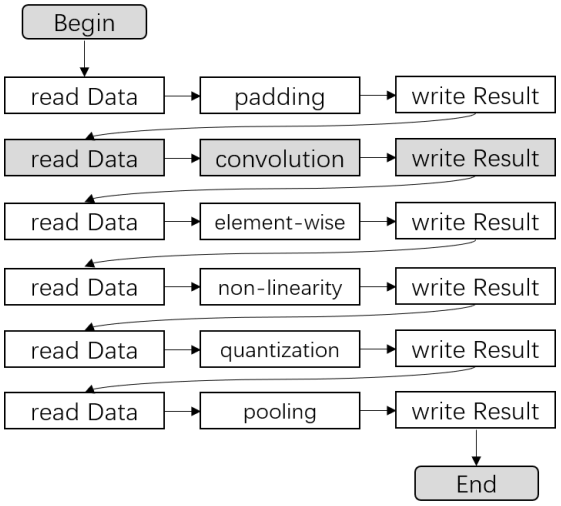

Figure 4.13: Distribution process of convolution block without operator fusion and without data reuse

In this mode, each operation is isolated from other operations. In other words, there is no concept of operating blocks in this mode (we use the operation blocks here to facilitate comparison with other distribution strategies). The next operation is accelerated only when all the subtasks of the current one is completed. During accelerating one of the operations, each PE runs entirely independently from other PEs. Once a PE has completed its work, it will output the results back to the DRAM and immediately get a new task, reading the data from the DRAM and compute it. Note that in simulation, except convolution, the execution of other operations (padding, element-wise, non-linearity, quantization and pooling) is assumed to be completed immediately. In other words, only gray parts in Figure 4.13 are simulated.

(ii) With operator fusion and without data reuse As it is apparent from the first mode, there is a lot of data movement between different operations. The new distribution process takes advantage of the operator fusion feature as shown in Figure 4.14. This new distribution process mainly obtains performance improvement from the decrease in the unnecessary data movement between different operations.

In this mode, the operation block is put in practice. When one operation is completed, the computed result is not transferred to DRAM. They are left for the next operation until all the operations in the operation blocks

are finished. Same as the first mode, PE runs entirely independently from other PEs. Note that in simulation, only the gray parts (the convolution operation) in Figure 4.14 are simulated.

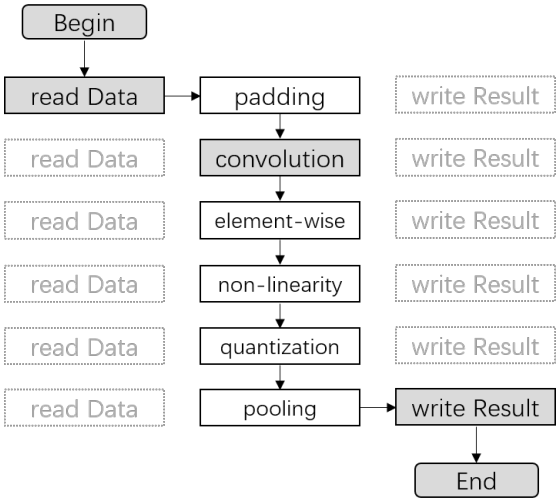

Figure 4.14: Distribution process of convolution block with operator fusion

(iii) With operator fusion and data reuse In this mode, the data reuse feature of convolution is employed to reduce data movement further. The data reuse is realized through changing the source PE for fetching operand A, which means that PEs in QPE are related to each other. Because of this, as mentioned before, the computation balance is essential for data reuse and must be considered in the splitter. If the PEs have tasks with different computational complexity, the PEs with lesser computation have to wait for the PEs with more computation, which makes the former PE idle for a while, wasting computing resource. Figure 4.15 shows the data reuse process. The source PEs of the two operands are called paired PEs. Through shifting the paired PEs, one QPE could accelerate $4 * 4 = 16$ convolution tasks with loading data into QPE only once.

Specifically, there are two types of data reuse, input feature map reuse and filter weight reuse. The reused data is unchanged until another operand is traversed. Practically, another operand is partially reused. For input feature map reuse, the input feature map is reused by all the filter weight, which means each part of the split input feature map is loaded into QPE only one time. Moreover, as MLA can fetch operand A from different PE SRAM,

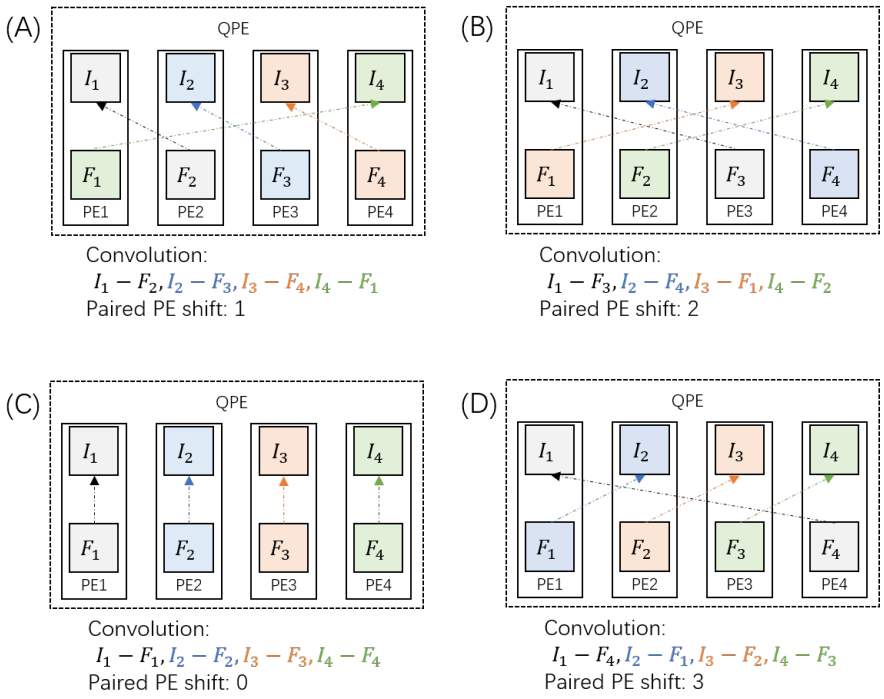

Figure 4.15: Data reuse through changing the source of operand A

therefore, if we change the source of operand A, then the filter weights could be reused by other PEs, which realizes partial data reuse of filter weights and the filter weight is loaded $ceil(P_{fmap}/4)$ times, where P_{fmap} is the number of parts of the split feature map. Whereas for filter weight reuse, each part of the split filter weight is loaded into QPE only once, while the input feature map is loaded $ceil(P_{filter}/4)$ times, where P_{filter} is the number of parts of the split filter. If the size of the split feature map and filter weight are $size_{fmap}$ and $size_{filter}$ respectively, the total data transfer size for the input feature map reuse and the filter weight reuse are

$$data\_size_{fmap\_reuse} = size_{fmap} + size_{filter} * ceil(P_{fmap}/4) \tag{4.2}$$

and

$$data\_size_{filter\_reuse} = size_{filter} + size_{fmap} * ceil(P_{filter}/4). \tag{4.3}$$

Comparing the result of equations 4.2 and 4.3 and select the data reuse with smaller data transfer size. As $ceil(P_{fmap}/4) \ll ceil(P_{filter}/4)$ or $size_{fmap} \gg$

$size_{filter}$ is true in most cases, the distributor tends to select input-feature-map-reuse.

The distribution process of the convolution block with operator fusion and data reuse is shown in Figure 4.14. The distribution algorithm of the convolution block using input feature map reuse is shown in Appendix C. Note that in simulation, only the gray parts (the convolution operation) in Figure 4.14 are simulated.

4.3.2 Pooling Block

Because the pooling operation has no property of data reuse, only two distribution strategies are developed for the pooling block, without operator fusion, and with operator fusion. The former is the same as the first distributed strategy (The one without operator fusion and data reuse) of the convolution block, the latter being the same as its second strategy (The one with operator fusion and without data reuse).

4.3.3 Matrix Multiplication Block

Matrix multiplication has the properties of operator fusion and data reuse. However, they cannot coexist: When using operator fusion, data reuse is not possible. The opposite is also exact. The acceleration brought by operator fusion is more significant than that by data reuse. Therefore, the data reuse will be deprecated (For the detail reasons, please refer to subsection 4.2.3 and Appendix B). Finally, two distribution strategies, without operator fusion and with operator fusion, are developed for accelerating matrix multiplication block. These two distribution strategies are the same as the first two for the convolution block.

4.4 Distributor for SpiNNaker2

For SpiNNaker2 distributor, the distribution strategy is also developed based on the flow as shown in Figure 4.12. The distribution strategy for the pooling block and the matrix multiplication block is the same as for QPE-DRAM, please refer to section 4.3 for more detailed information.

4.4.1 Convolution Block

For the convolution block, same as for QPE-DRAM simulator, three types of distribution strategy are developed:

- without operator fusion and data reuse,
- with operator fusion and without data reuse,
- with operator fusion and data reuse.

The first two distribution strategies are the same as the first two for QPE-DRAM, whereas the third distribution strategy is different. In the following, the third distribution strategy will be fully introduced.

Comparing to QPE-DRAM, which has a ratio of DRAM to QPE of 1:1, the ratio of DRAM to QPE in SpiNNaker2 is 1:9. It could be expected that the performance of SpiNNaker2 is heavily limited by DRAM (The simulation results of the first two allocation strategies also confirm this conjecture). Due to this, corresponding to 4 DRAMs, the distributor treats 4 QPEs (QPE(2,2), QPE(3,2), QPE(2,3) and QPE(3,3)) as storage QPEs. If the storage QPE has enough space, the result of the calculation is stored in the storage QPE SRAM instead of the DRAM. In this way, the performance bottleneck caused by DRAM is reduced. After that, there are still 32 computation QPEs (128 PEs) (88.9% of computing power) available for computation. Although some of the computing power was sacrificed, it is beneficial to improve the acceleration speed in most cases. In order to better use the 4 storage QPEs and 4 DRAMs, the computation QPEs are grouped into 4 groups, which owns one DRAM and one storage QPE. The 4 groups are called **double QPE block**s, and each **double QPE block** has 2 **QPE block**s, which contains 4 QPEs. Figure 4.16 shows this functional division.

After cutting some computational power, it is easier for the remaining QPEs (computation QPEs) to realize data reuse inside SpiNNaker2. For the data reuse in SpiNNaker2, the reused data is the operand fixed in the SRAM. Another operand is the migrated data, which needs to migrate in SpiNNaker2. For data reuse in QPR-DRAM, there are two types of data reuse, input feature map reuse and filter weight reuse. However, only the input feature map reuse is available for SpiNNaker2 due to the following two reasons. Firstly, filter tends to be split into 2^n parts, where n is an integer. This makes the weight migration in SpiNNaker more convenient. Secondly, the migrated data requires an extra memory space to prevent data overwriting during data migration. Because in each PE, the input feature map takes

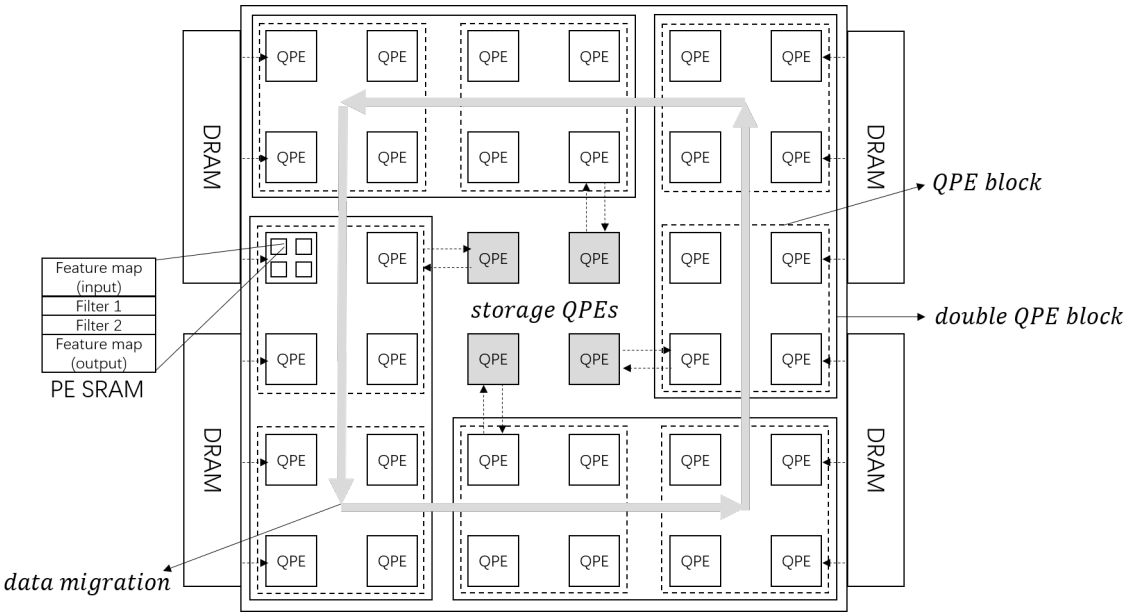

Figure 4.16: The functional division in SpiNNaker2 for data reuse

more space than filter weight, this makes filter weight reuse difficult to realized (In most cases, it is impossible to double its memory space). Hence filter weight reuse is not feasible.

Practically, in the input feature map reuse, the filter weight is partial reused. The partial filter weight reuse process in SpiNNaker2 is more complicated than that in QPE-DRAM. In addition to the weight partial reuse by changing the source of operand A, SpiNNaker2 can reuse weight through data migration via NoC. In data reuse mode, input feature maps are fixed in each PE SRAM, and filter weight migrates through the whole SpiNNaker2 letting that each input feature map is convoluted with all the filters. The filter weight partial reuse hierarchy, which separates different ways to reuse the weight into a hierarchy based on migration latency, is shown in Figures 4.17 and 4.18. The partial reuse hierarchy helps to move the bottleneck on DRAM towards NoC and improve the performance. The latency comes from the data movement between PEs.

Inside QPE: PE↔PE: This type of partial data reuse is also being applied in the QPE-DRAM distributor and has no latency.

Inside QPE block: QPE↔QPE: This type of partial data reuse is realized by swapping data between QPEs. After data migration, data reuse inside QPE is performed. The data movement bandwidth is up to $16 * BW_{NoC}$,

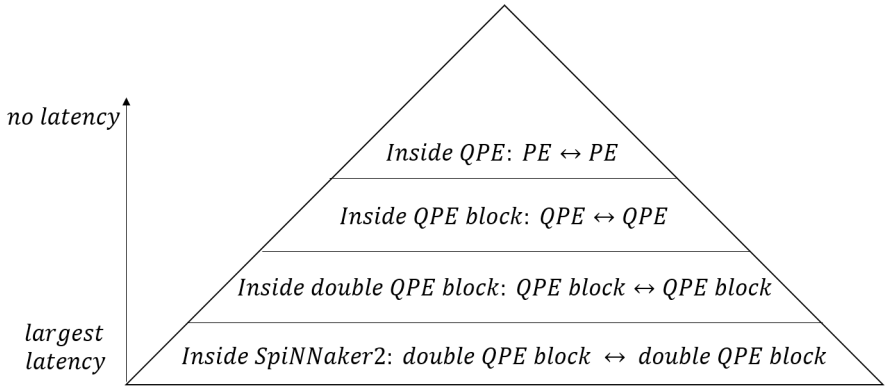

Figure 4.17: Partial data reuse hierarchy in SpiNNaker2

where BW_{NoC} is the memory bandwidth of NoC.
Inside double QPE block: QPE block↔QPE block: This type of data reuse is realized by swapping data between the QPE blocks. After data migration, data reuse inside QPE or QPE block is possible. The data movement bandwidth is up to $8 * BW_{NoC}$.
Inside SpiNNaker2: double QPE block↔double QPE block: This type of data reuse is realized by swapping data between double QPE blocks. After data migration, data reuse inside QPE or QPE block or double QPE block is possible. The data movement bandwidth is up to $8 * BW_{NoC}$.

The distribution process of the convolution block is the same as that for QPE-DRAM, as shown in Figure 4.14. In order to support the data migration and fully utilize the computing resource, the splitter tends to decompose the input feature map into or close to 2^n parts, where n is an integer. Furthermore, in order to more efficiently manage the output feature map, the distributor will allocate an entire row of the split feature map into a **double QPE block**. If the height of the input feature map is split into 1 or 2 parts, the whole feature map will be copied 3 or 1 times so that each storage QPE or DRAM has a copy. Two examples are shown in Figure 4.19. When the input feature map's height and width are split into 7 and W parts, which means the input feature map is split into $7 * W$ parts. The first 2 rows with total $2 * W$ parts will be assigned to the first **double QPE block**, the next 2 rows with $2 * W$ parts will be assigned to the second **double QPE block** and so on to the third **double QPE block**. However, for the fourth **double QPE block**, there is only one row with total W parts. If the input feature map's height is split into 2 parts, the whole input feature map will be copied by 1 time. The first row with total W parts is assigned to the first and second

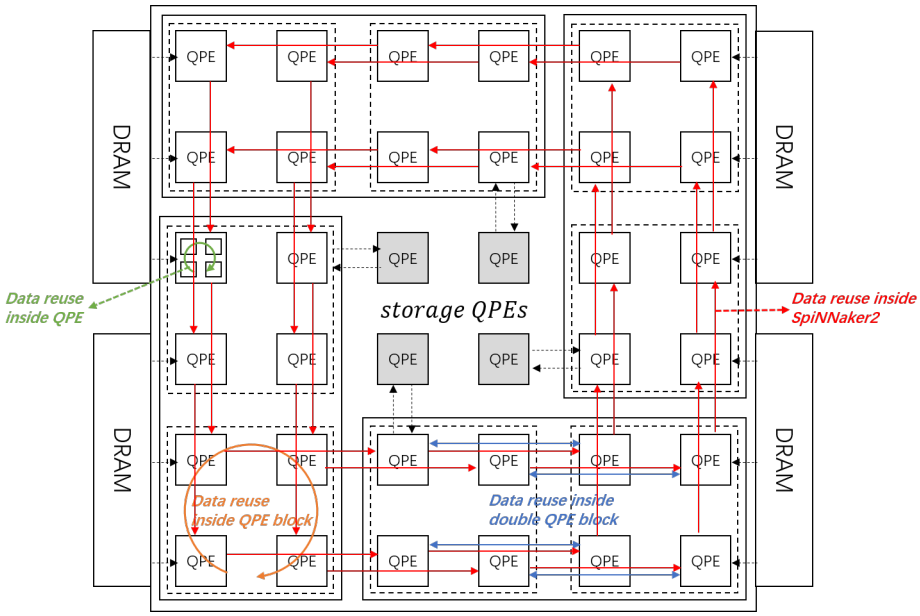

Figure 4.18: Partial data reuse in SpiNNaker2

double QPE block, which means the first and second **double QPE block** have the same input feature maps. The second row with total W parts is assigned to the third and fourth **double QPE block**. Each **double QPE block** will load the input feature map from its storage QPE or DRAM, and the loading process of each **double QPE block** is entirely independent of other **double QPE block**s. Moreover, the distributor will make sure that each PE has a part of the input feature map. In case that some PEs own the same feature map, this is accomplished by data migration through NoC to reduce the bottleneck by the DRAM. Appendix D has some examples of how the distributor loads the feature map into a **double QPE block**.

For the filter weight, the splitter will split it into 2^n parts, where $n >= 4$. The distributor makes sure that each PE has a part of the filter weight. In the case of $n < 7$, some PEs own the same part of the filter weights, and this is also done by data migration through NoC. Appendix E has some examples of how the distributor load the filter weight into a **double QPE block**.

Furthermore, Appendix F uses an example to show the data reuse and data migration process in SpiNNaker2.

Because of the data reuse, all the PEs in SpiNNaker are related to each other, which means that all PEs work or are idle almost simultaneously. Figure 4.20 shows the acceleration time frame. Since each time period is

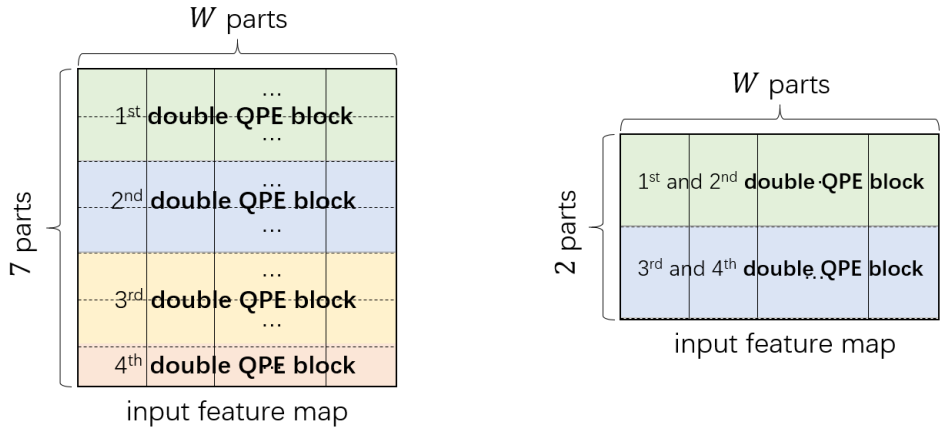

Figure 4.19: Feature map assignment for SpiNNaker2

well isolated, it is easy to analyze where performance bottlenecks occur.

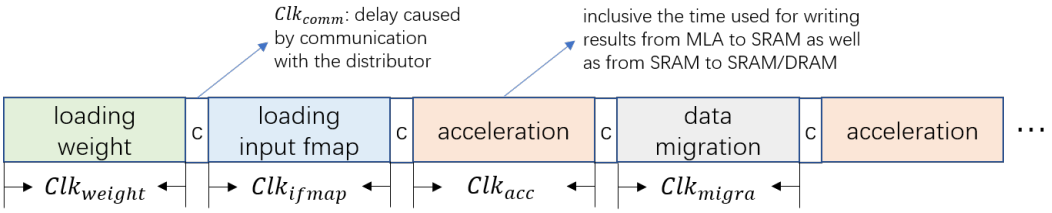

Figure 4.20: Time frame for data reuse strategy in SpiNNaker2

5 Validation, Simulation and Experiment

In this chapter, the mapper and the simulator will be chained together to give an evaluation of the SpiNNaker2. In order to ensure the accuracy of SpiNNaker2Py and the feasibility of the splitter, the mapper and the simulator will be verified in section 5.1. The performance that different distribution strategies can bring will be shown in section 5.2 and 5.3 respectively for QPE and SpiNNaker2. Moreover, in section 5.3, the overall processing time of VGG-16 and ResNet-50 on SpiNNaker2 with operator fusion and data reuse will be estimated. Further, we will study the comparison between the performance of MLAs with different numbers of MAC units in section 5.4. A comprehensive discussion is placed in the last section 5.5.

In order to better represent the various performance indicators, the various components in the simulator will run as shown in the Table 5.1. Note that most of the work in this chapter is based on these assumptions and it will be explained if there are changes.

Components	Frequency (MHz)	Clocks per operation
NoC	500	1
DRAM	250	2
HOST Interface	250	1
PE	250	1
ARM in PE	250	1
SRAM in PE	250	1
DMA in PE	250	1

Figure 5.1: Frequency of components in SpiNNaker2Py

5.1 Validation

In this thesis, the mapper and SpiNNaker2Py were developed to estimate or analyze the performance of SpiNNaker2. Ensuring that the final simulation results are convincing, the feasibility of the splitter and the accuracy of the SpiNNaker2Py will be verified.

5.1.1 Validation of Splitter

As shown in chapter 4, the mapper is composed of a parser, a splitter and a distributor. Because the mapper will change the computation of neural network, and this change mainly comes from the splitter, the splitter's decomposition scheme will be verified whether it can keep the calculation results unchanged.

Only the convolution, pooling and matrix multiplication operation are the verified objects, as other calculation operations are element-wise and splitting them will not cause distortion of the calculation result. Figure 5.2 shows the validation process. Given with the CNN architecture information, the parser and the splitter decompose the network into pieces. Simultaneously, random data (for input feature map and weight) are generated, and the reference convolution result is calculated using TensorFlow [11]. On the other side, the generated random input data is split according to the splitting scheme from the splitter and the convolution results based on the split random input data are obtained through TensorFlow. The compared convolution result is acquired by recombining the convolution results based on the split input data. Finally, the feasibility of the splitter can be verified by comparing the reference convolution result and the compared convolution result. The verification results indicate that both the parser and the splitter passed.

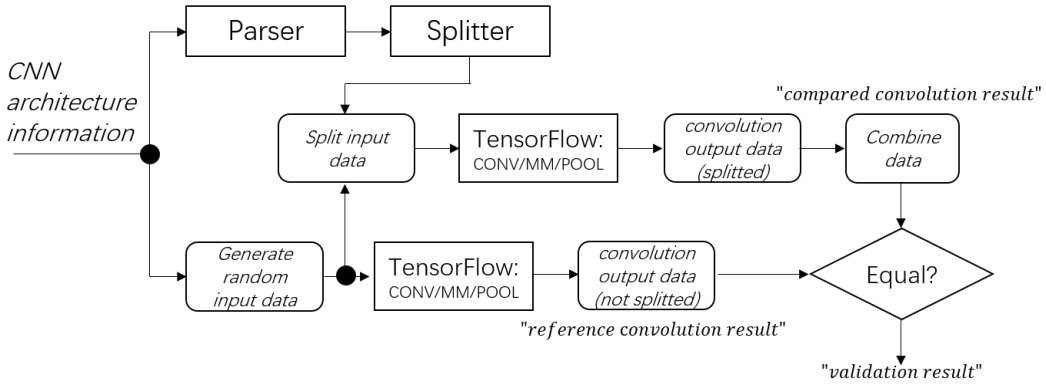

Figure 5.2: Validation process of splitter

5.1.2 Validation of SpiNNaker2Py

As mentioned before, SpiNNaker2 was still under development, and only the QPE HDL (Hardware Description Language) prototype was available.

Therefore only the QPE simulator is validated. Because the QPE is the basic building block of SpiNNaker2 with NoC and PEs, if the QPE simulator is verified, the accuracy of the SpiNNaker2 simulator can be approximately derived.

The QPE simulator will be verified in two ways. One is to get operand A from the local SRAM, and the other is to get the operand A from the neighbor PE SRAM. When validating the QPE simulator, the QPE HDL prototype is used as a reference model for verification. The verification process without reading input data into SRAM and without writing computed results from SRAM is as follows.

(1) Continuously trigger 4 PEs to execute convolution or matrix multiplication with the same tasks. The tasks are mainly taken from the split result of VGG-16 and ResNet-50. MLA fetches operand A from its local PE SRAM or neighbor PE SRAM. (This step is done both on the QPE HDL prototype and the QPE simulator)

(2) Waiting the 4 PEs to finish the convolution or matrix multiplication tasks and recording the entire clock cycles, which include the clocks for writing the computed results from MLA to SRAM. (This step is done both on QPE HDL prototype and QPE simulator)

(3) Calculate the clock difference between the QPE HDL prototype and the QPE simulator. The clock deviation δ determines the accuracy of the QPE simulator with equation 5.1.

$$\delta = \frac{CLK_{simulator} - CLK_{ICPRO}}{CLK_{ICPRO}} \tag{5.1}$$

, where $CLK_{simulator}$ and CLK_{ICPRO} are the clock cycles needed for the simulator and the HDL prototype. If the absolute value of the clock deviation is less than 10%, the simulator is considered to meet the requirements.

Note that in the validation, the QPE-DRAM simulator is employed instead of the QPE simulator, because the QPE-DRAM simulator has a host interface. The additional component DRAM does not affect the validation result because it is assumed that all required data is already in the SRAM. Note that there were some problems with the current QPE HDL prototype, where the SRAM requires two clocks for per reading or writing operation instead of one clock, so the SRAM simulator is set to show the same behavior during validation.

(i) Operand A: local PE SRAM The above verification process with fetching operand A from local PE SRAM is applied to the QPE HDL prototype and the QPE-DRAM simulator. The validation result for the convolution and the matrix multiplication is shown in Tables 5.1 and 5.2 respectively. The deviations of the clock cycles are between -7.12% and 5.04%, whose absolute values are below 10%, meeting the requirements. For convolution, the large deviations happen on the convolution with larger filters (e.g., 5x5, 7x7 and 9x9), which are not common in the state-of-art models.

Convolution task input feature map dimension: $[W_i, H_i, D]$, filter dimension: $[W_f, H_f, D, C]$, stride: 1	**Clocks** (HDL prototype)	**Clocks** (Simulator)	**Clock deviation** $(\frac{CLK_{simulator}-CLK_{ICPRO}}{CLK_{ICPRO}})$
fmap: [226,22,3] filter: [3,3,3,4]	27748	25771	-7.12%
fmap: [114,9,64] filter: [3,3,64,4]	61186	59543	-2.69%
fmap: [18,18,128] filter: [3,3,128,4]	38626	38264	-0.94%
fmap: [30,9,256] filter: [3,3,256,4]	66244	66342	0.15%
fmap: [56,14,64] filter: [1,1,64,4]	10822	10599	-2.06%
fmap: [28,10,256] filter: [1,1,256,4]	13162	13395	1.77%
fmap: [28,14,128] filter: [5,5,128,4]	116215	109402	-5.86%
fmap: [28,10,128] filter: [7,7,128,4]	82139	86275	5.04%
fmap: [16,16,128] filter: [9,9,128,4]	31648	32883	3.90%

Table 5.1: Clock accuracy of QPE simulator for convolution (local PE SRAM)

Matrix Multiplication task Matrix **A** dimension: $[W_A, H_A]$, Matrix **B** dimension: $[W_B, H_B]$	**Clocks** (HDL prototype)	**Clocks** (Simulator)	**Clock deviation** $(\frac{CLK_{simulator}-CLK_{ICPRO}}{CLK_{ICPRO}})$
fmap: [64,1] weight: [1024, 64]	13276	12619	-4.95%
fmap: [128,1] weight: [512,128]	11908	11435	-3.97%

Table 5.2: Clock accuracy of QPE simulator for matrix multiplication (local PE SRAM)

(ii) Operand A: neighbor PE SRAM In this validation, PE fetching operand A from neighbor PE SRAM will be validated. Additionally, it is expected that their clock cycles should be close to that of fetching operand A from local PE SRAM.

There were some design flaws of the QPE HDL prototype discovered during validation: when fetching too much data from neighbor PE SRAM, the MLA does not work well.

For convolution, when the weight is too large, the execution of MAC array in some PEs will be aborted in advance before completing the task, and the remaining PEs will run for an unusually long time to complete the task. The issue has nothing to do with the size of the input feature map. The limit value of the weight size is currently unknown. In the case of $[W_f, H_f, C] = [3,3,4]$, the problem does not occur when $C = 3$ but occurs when $C \geq 64$.

For matrix multiplication, when the width of matrix A is too large, all PEs will keep running without stop. The issue has nothing to do with other dimensions of matrix A and matrix B. The limit value of the width of matrix A is also unknown. In the case of $W_A = 64$, the problem does not occur, but occurs when $W_A = 128$.

Tables 5.3 ande 5.4 show the validation results for convolution and matrix multiplication.The clock differences between the HDL prototype and the simulator are located at $[-0.8\%, -9.51\%]$, while the clock differences between different PE shifts of the HDL prototype and the simulator are at $[0.00\%, 2.88\%]$ and $[-1.00\%, 6.22\%]$ respectively. All of them are within the acceptable range.

Neighbor PE SRAM	Clocks (Error to shift 0) (HDL prototype)	Clocks (Error to shift 0) (Simulator)	Clock deviation $(\frac{CLK_{simulator}-CLK_{ICPRO}}{CLK_{ICPRO}})$
Neighbor PE shift: 0	27748 (0.00%)	25771 (0.00%)	-7.12%
Neighbor PE shift: 1	27735 (0.00%)	25772 (0.00%)	-7.08%
Neighbor PE shift: 2	28482 (2.65%)	25772 (0.00%)	-9.51%
Neighbor PE shift: 3	27726 (0.00%)	25773 (0.00%)	-7.04%

Table 5.3: Clock accuracy of QPE simulator for convolution with $[W_i, H_i, D] = [226,22,3]$ and $[W_f, H_f, D, C] = [3,3,3,4]$ (Fetching operand A from neighbor PE SRAM). The neighbor PE shift indicates how the PEs are paired (Details in subsection 4.3.1). The *error to shift 0* is calculated using $\frac{CLK_{pe\_shift\_x}-CLK_{pe\_shift\_0}}{CLK_{pe\_shift\_0}}$.

Neighbor PE SRAM	Clocks (Error to shift 0) (HDL prototype)	Clocks (Error to shift 0) (Simulator)	Clock deviation $(\frac{CLK_{simulator}-CLK_{ICPRO}}{CLK_{ICPRO}})$
Neighbor PE shift: 0	13276 (0.00%)	12619 (0.00%)	-4.95%
Neighbor PE shift: 1	13563 (2.16%)	13454 (6.62%)	-0.80%
Neighbor PE shift: 2	12893 (2.88%)	12493 (-1.00%)	-3.10%
Neighbor PE shift: 3	13577 (2.27%)	12974 (2.81%)	-4.44%

Table 5.4: Clock accuracy of QPE simulator for matrix multiplication with $[W_A, H_A] = [64,1]$ and $[W_B, H_B] = [1024,64]$ (Fetching operand A from neighbor PE SRAM). The neighbor PE shift indicates how the PEs are paired (Details in subsection 4.3.1). The *error to shift 0* is calculated using $\frac{CLK_{pe\_shift\_x}-CLK_{pe\_shift\_0}}{CLK_{pe\_shift\_0}}$.

5.2 Mapper on QPE-DRAM Simulator

In this section, we will show the mapping results on the QPE-DRAM simulator with the three different parallelization strategies, illustrated in section 4.3. The mapping objects are VGG-16 and ResNet-50 (refer to section 2.2). We firstly introduce the split result for the QPE-DRAM, then comes to the mapping result on the QPE-DRAM.

5.2.1 VGG-16 and ResNet-50 on Splitter for QPE-DRAM

This subsection analyzes the split result scheme, the SRAM utilization, and the MAC unit utilization in each PE. By default, the splitter assumes there are 96 KByte SRAM available for MLA.

The SRAM utilization r_{sram} indicates how much memory the valid data (data before alignment) occupies and is calculated by equation 5.2.

$$r_{SRAM} = \frac{size_{input} + size_{weight} + size_{output}}{96KBytes} \tag{5.2}$$

where $size_{input}$, $size_{weight}$ and $size_{output}$ represent the size of input, weight, and output before alignment. Moreover, every single data of the input and weight takes 8 bits but 32 bits for the output.

The MAC unit utilization represents how many MAC units in MLA are used during computation. It is calculated by equation 5.3.

$$r_{MAC} = \frac{W_o * C}{ceil_{rows}(W_o) * ceil_{columns}(C)} \tag{5.3}$$

where $ceil_{rows}()$ and $ceil_{columns}()$ respectively represent the alignment to the number of rows of MLA and the alignment to the number of columns of MLA.

(i) VGG-16 In the VGG-16 network, since the pooling stride of the pooling operation always equals to the width and height of the pooling filter, all the pooling operation will be integrated into previous convolution operation to form the convolution block. Therefore, there is no pooling block for VGG-16. The split result is shown in Table G.1 in Appendix G.

The SRAM and MAC unit utilization for the convolution operation and the matrix multiplication operation are shown in Table G.2 in Appendix G. For convolution operations, the MAC unit utilization is at least 0.88, while the SRAM utilization is between 0.53 and 0.95, and the SRAM utilization for most operations exceeds 0.80. For the matrix multiplication, the MAC unit utilization is only 0.25 (Because the batch size is 1), and the SRAM utilization is 0.68.

(ii) ResNet-50 Different from VGG-16, the pooling stride of the pooling operation in ResNet-50 is not equal to the width and the height of the pooling filter, so the pooling operation will not be integrated into a convolution operation and forms a pooling block. Figure H.1 in Appendix H shows the split result.

The SRAM and MAC unit utilization of different operations are shown in Table H.2 in Appendix H. The MAC utilization for most operations is 0.88. The SRAM utilization is in the range of 0.23 and 0.97, and most operations have SRAM utilization below 0.50.

5.2.2 Mapping Convolution Operation and Matrix Multiplication Operation on QPE-DRAM Simulator

For VGG-16, the mapping results of the convolution operation and the matrix multiplication operation are shown in Table 5.5. Whereas Table 5.6 is for ResNet-50. Since most of the convolution operation decomposition schemes are similar, only the representative convolution operations are listed in the tables. Their corresponding bar charts are shown in Figures 5.3 and 5.4 for better intuitive visual performance improvement. Comparing to the distribution algorithm without operator fusion and data reuse, the one with operator fusion but without data reuse has an improvement up to around 3 times, but only for a few operations, and there is only little improvement for the most

operations. The third algorithm with operator fusion and data reuse has a better improvement (up to 5 times) but is also limited to a few operations.

Furthermore, in order to provide an intuitive visual gap with the bandwidth and performance ceiling of the QPE, Roofline models[12] are employed and shown in Figure 5.5 and 5.6 respectively for VGG-16 and ResNet-50. The X-axis of the Roofline model represents the operational intensity, while the Y-axis is the attainable performance. In the Roofline model, the diagonal solid blue line and the diagonal solid red line represent the peak bandwidth of DRAM and NoC respectively, whereas the horizontal solid black line is the ceiling of the peak performance of QPE. The attainable performance of the algorithm with operator fusion or data reuse is increased and moved towards the bandwidth ceiling or performance ceiling of QPE. However, the achievable performance on CONV_42 and CONV_SC_14 is quite far from the ceiling, because the convolution stride is 2 so that only around half of the used MAC units are valid. For the matrix multiplication, as we analyzed before, its bottleneck mainly comes from the DRAM rather than the computing resources, and it touches the bandwidth ceiling.

Operation	Clocks (No operator fusion) (No data reuse)	Clocks (Operator fusion) (No data reuse)	Clocks (Operator fusion) (Data reuse)
CONV_1	2294223	835667	1065837
CONV_5	16207531	11050716	9697264
CONV_9	12616201	12417980	10838832
CONV_11	6369574	6262524	5419584
CONV_15	3261186	3254902	2918496
FC_21	546718	532670	—-

Table 5.5: VGG-16 on QPE: Mapping result of convolution operation and matrix multiplication operation

Operation	Clocks (No operator fusion) (No data reuse)	Clocks (Operator fusion) (No data reuse)	Clocks (Operator fusion) (Data reuse)
CONV_1	2402170	2304196	2320116
CONV_5	2246963	1717913	471968
CONV_7	813317	785374	716836
CONV_SC_14	13356221	5776704	2722496
CONV_19	806278	788695	700808
CONV_25	761451	753440	739465
CONV_26	422420	383365	369810
CONV_42	4145252	3675771	879624
CONV_43	1644412	1527487	1598560
FC_52	276382	269502	—-

Table 5.6: ResNet-50 on QPE: Mapping result of convolution operation and matrix multiplication operation

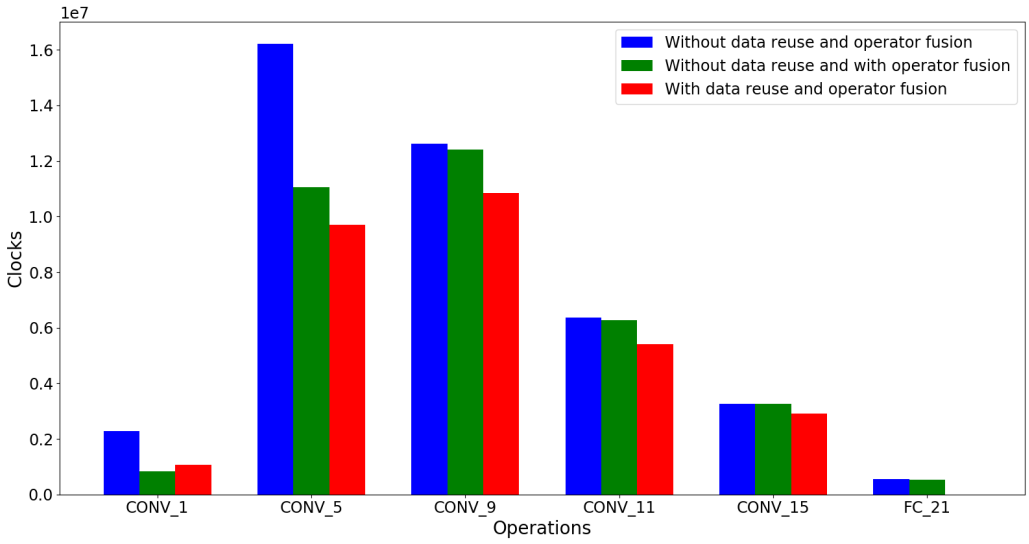

Figure 5.3: Comparison of the distribution strategies one VGG-16 and QPE

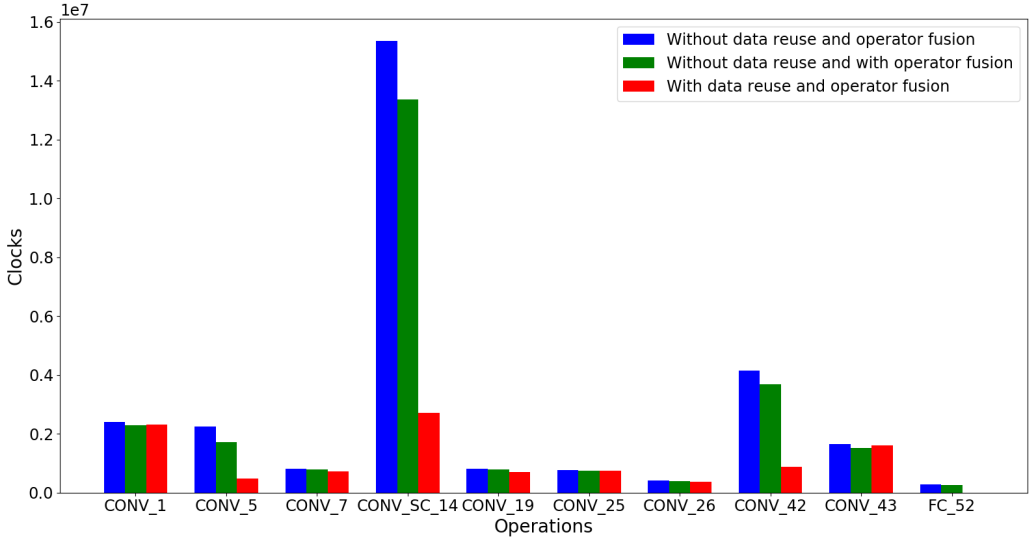

Figure 5.4: Comparison of the distribution strategies one ResNet-50 and QPE

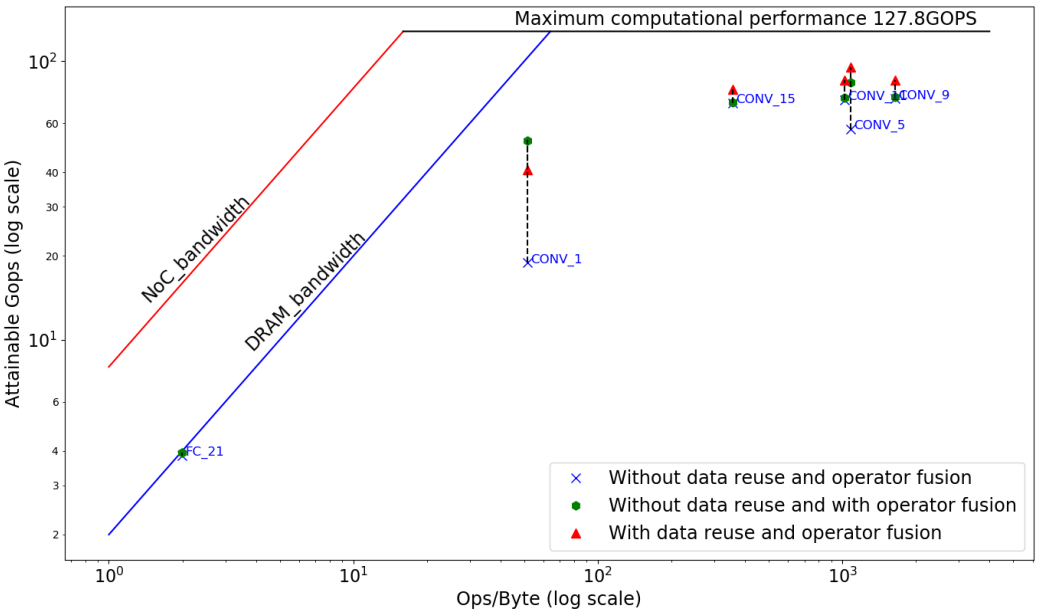

Figure 5.5: Roofline model for convolution and matrix multiplication of VGG-16 on QPE

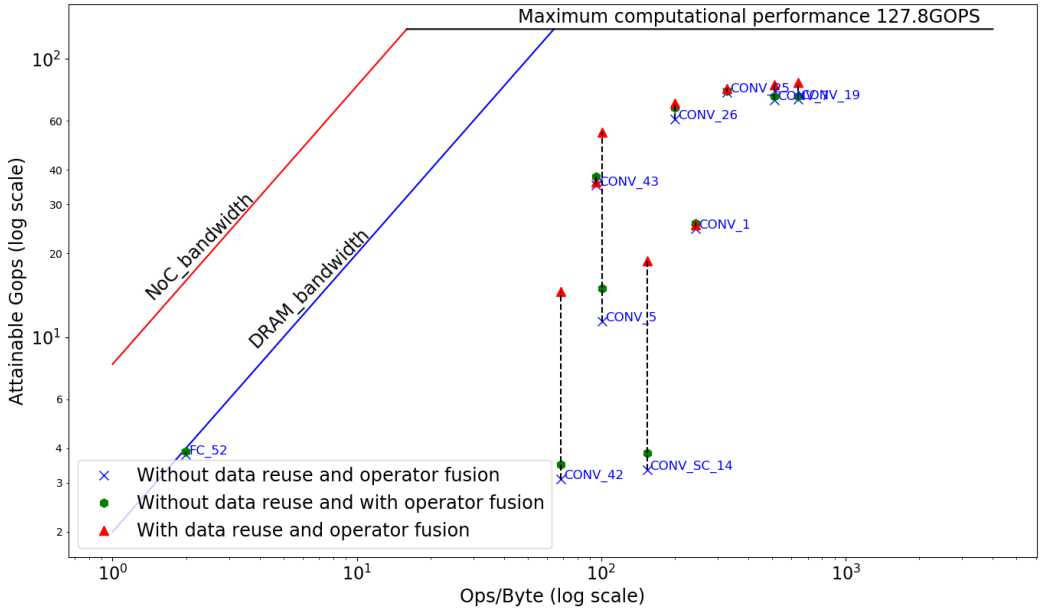

Figure 5.6: Roofline model for convolution and matrix multiplication of ResNet-50 on QPE

5.3 Mapper on SpiNNaker2 Simulator

5.3.1 VGG-16 and ResNet-50 on Splitter for SpiNNaker2

Significantly different from the hardware specification of the QPE, SpiNNaker2 has 36 QPEs (144 PEs), in order to utilize the computation resources, the splitter performs a slightly different split strategy in order to utilize SpiNNaker2 fully. The split result of VGG-16 as well as the SRAM and MAC unit utilization reports are listed in Appendix I, whereas the reports for ResNet-50 are in Appendix J. Same as for the QPE-DRAM simulator, the splitter assumes there are 96 KByte SRAM available for MLA.

In VGG-16, the MAC unit utilization of the convolution operations is at least 0.88, while the SRAM utilization of the convolution operations is between 0.51 and 0.85. For the matrix multiplication, its MAC unit utilization is 0.25, and its SRAM utilization is pretty low, only up to 0.34.

In ResNet-50, the MAC utilization for most operations is 0.88. The SRAM utilization is in the range of 0.08 and 0.92, and most operations have SRAM utilization below 0.50.

5.3.2 Mapping Convolution Operation and Matrix Multiplication Operation on SpiNNaker2 Simulator

For VGG-16 and ResNet-50, their mapping results of the convolution operation and matrix multiplication operation on SpiNNaker2 are shown in Tables 5.7 and 5.8, respectively. Their corresponding bar charts are shown in Figures 5.7 and 5.8. Comparing to the distribution algorithm without operator fusion and data reuse, the one with operator fusion but without data reuse has an improvement up to around 5 times, but only for a few operations. Whereas the algorithm with operator fusion and data reuse has a significant improvement, up to 10 times, and the improvement of most operations is significant, beyond 5 times.

The Roofline model figures for the distribution results are respectively shown in Figures 5.9 and 5.10 for VGG-16 and ResNet-50. It can be seen that both operator fusion and data reuse help to increase the overall performance towards the bandwidth ceiling or the performance ceiling of QPE. However data reuse brings much more performance than operator fusion. Same as in the QPE-DRAM simulator, the bottleneck of the matrix multiplication also appears on the DRAM of the SpiNNaker2 simulator.

Operation	Clocks (No operator fusion) (No data reuse)	Clocks (Operator fusion) (No data reuse)	Clocks (Operator fusion) (Data reuse)
CONV_1	494729	114249	119254
CONV_5	4139243	3557807	381753
CONV_9	2538019	2437194	413733
CONV_11	1409122	1302970	263877
CONV_15	1457186	1446968	189908
FC_21	150355	132044	—-

Table 5.7: VGG-16 on SpiNNaker2: Mapping result of convolution operation and matrix multiplication operation

Operation	Clocks (No operator fusion) (No data reuse)	Clocks (Operator fusion) (No data reuse)	Clocks (Operator fusion) (Data reuse)
CONV_1	216184	157468	99623
CONV_5	332016	228080	45312
CONV_7	169620	154441	37874
CONV_SC_14	1088654	939526	115207
CONV_19	190469	177575	44987
CONV_25	206821	204440	56688
CONV_26	159985	128559	32182
CONV_42	356132	325505	47116
CONV_43	408563	278969	133136
FC_52	75575	66513	—-

Table 5.8: ResNet-50 on SpiNNaker2: Mapping result of convolution operation and matrix multiplication operation

5.3.3 Overall processing clocks on SpiNNaker2

As mentioned in chapter 4, the clock cycles for the operation accelerated by ARM core is not simulated in SpiNNaker2Py, as the ARM core is difficult to simulate. In this subsection, we are going to estimate their clocks using the average experimental value given by the supervisor, as shown in Table 5.9. Furthermore, the estimation is done under the distribution algorithm with operator fusion and data reuse, illustrated in section 4.4, because each time period is well isolated and the clocks by the ARM core can directly add to the clocks used by MLA. Note that according to Figure 4.14, the data transfer clocks are already included in the convolution and matrix multiplication operations.

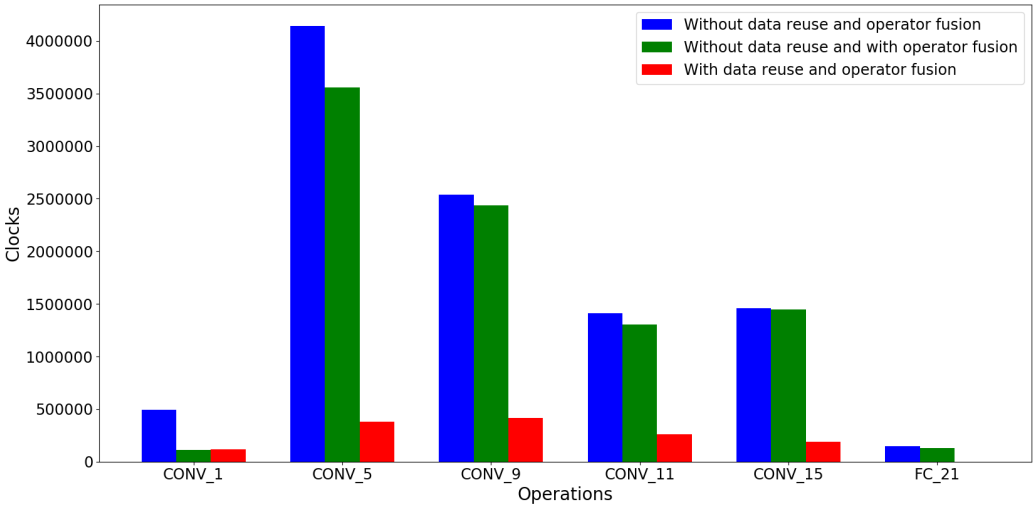

Figure 5.7: Comparison of the distribution strategies one VGG-16 and SpiNNaker2

Operation	Clocks	Comments
padding	2	per 32-bit
quantization	8	per input pixel
non-Linearity	8	32-bit ReLU, per input pixel
non-Linearity	2.5	8-bit ReLU, per input pixel
MAX-pooling	18.75	32-bit, per input pixel
MAX-pooling	12	8-bit, per input pixel
matrix element-wise addition	8	per input pixel

Table 5.9: Average experimental clocks for operations accelerated by ARM

Figure 5.11 shows the overall processing clocks of VGG-16 and ResNet-50. The detail clock information for each operation is shown in Table 5.10. For VGG-16, clocks for matrix multiplication takes almost the same time as for convolution, and they use 73.20% of the time. Whereas for ResNet-50, the convolution takes the most of time, around 59.54%. The overall clocks of VGG-16 is 2.2 times of ResNet-50. In theoretical analysis, the convolution of VGG-16 and ResNet-50 occupies more than 90% of the computation. However, their running time in SpiNNaker2 did not reach such a high ratio, which indicates that SpiNNaker2 uses too much time on other operations.

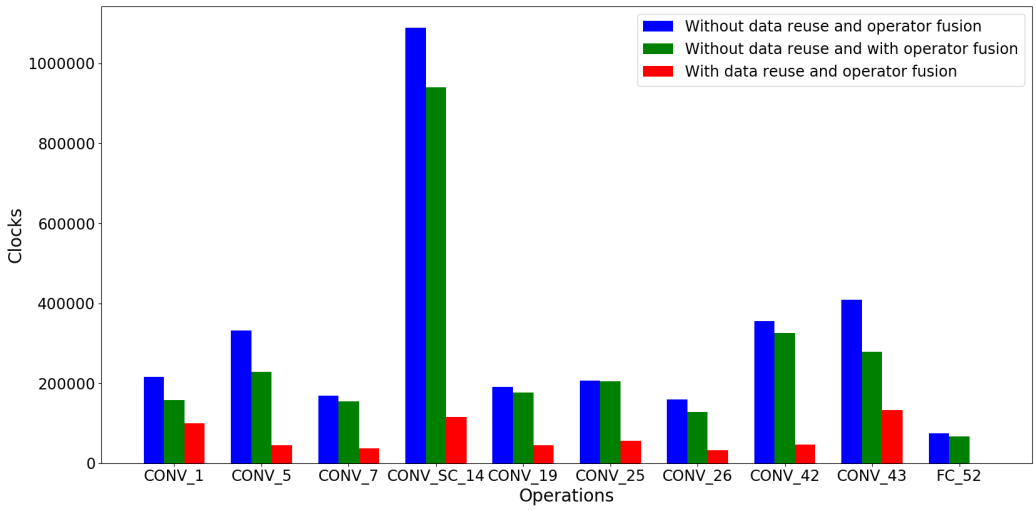

Figure 5.8: Comparison of the distribution strategies one ResNet-50 and SpiNNaker2

Operation	Clocks (VGG-16)	Clocks (ResNet-50)
CONV	4026732	2901614
FC	3894744	66513
PADD	340608	355240
MAT_ELE	0	351232
ACTI	932352	585312
QUAN	932352	585312
POOL	694272	28224
sum	10821060	4873447

Table 5.10: Clocks for different operations for VGG-16 and ResNet-50. CONV: convolution, FC: matrix multiplication, PADD: padding, MAT_ELE: matrix element-wise addition, ACTI: ReLU, QUAN: quantization, POOL: pooling.

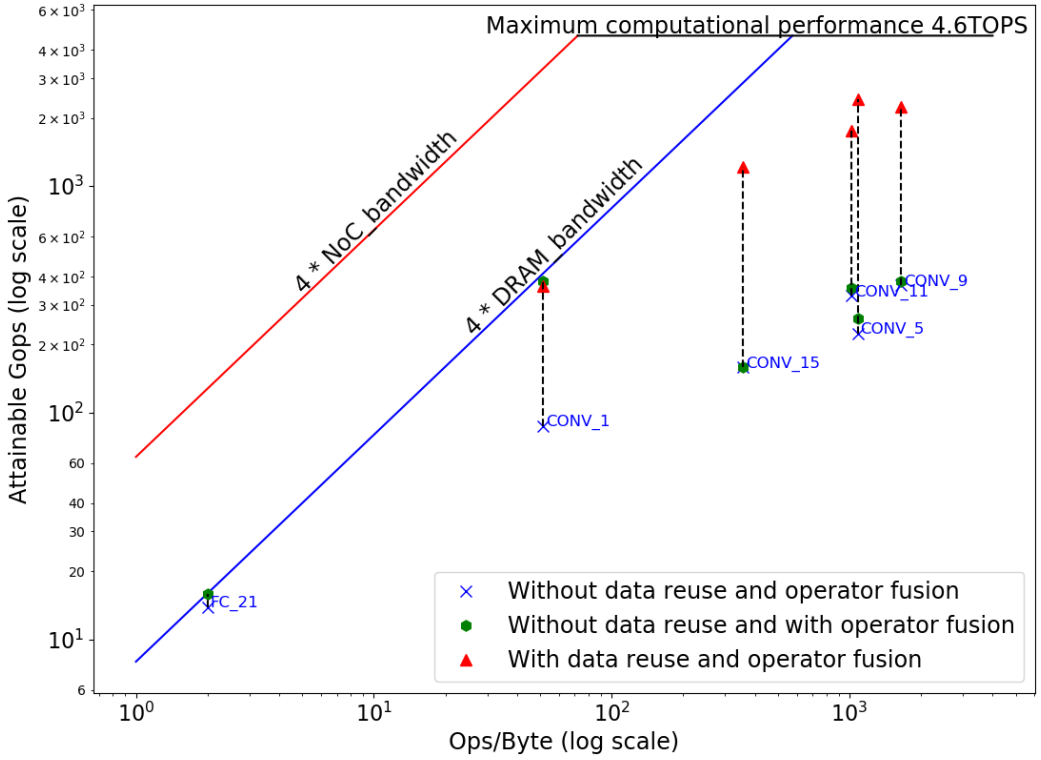

Figure 5.9: Roofline Model for VGG-16 on SpiNNaker2

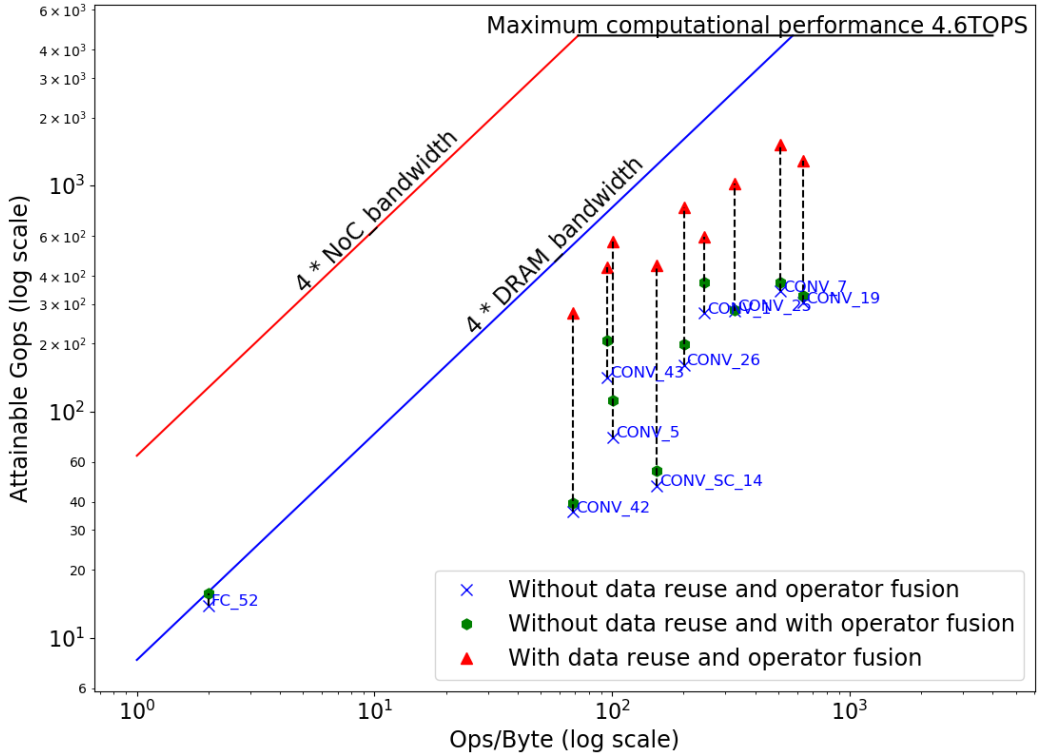

Figure 5.10: Roofline Model for ResNet-50 on SpiNNaker2

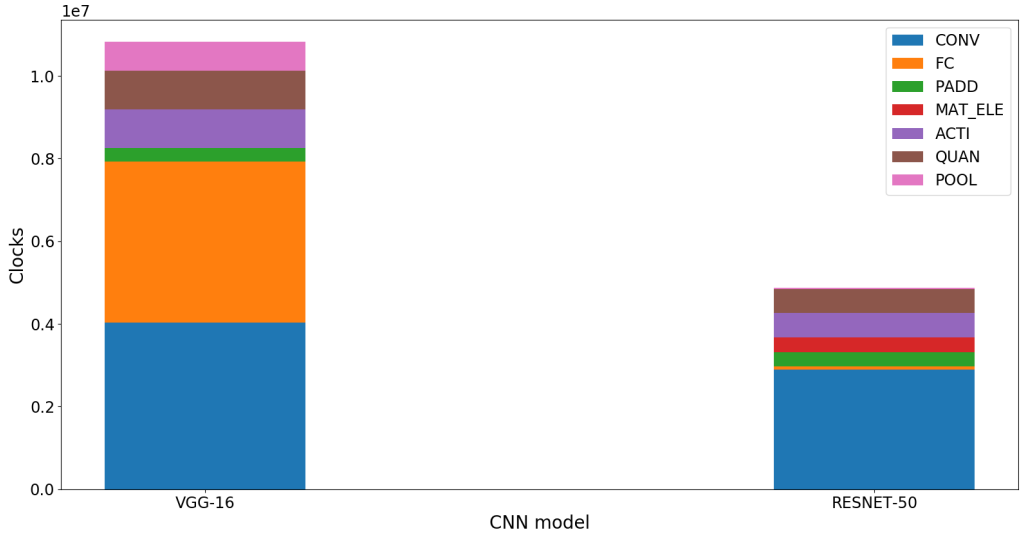

Figure 5.11: Overall processing clocks of VGG-16 and ResNet-50. CONV: convolution, FC: matrix multiplication, PADD: padding, MAT_ELE: matrix element-wise addition, ACTI: ReLU, QUAN: quantization, POOL: pooling.

5.4 Comparison between MLA with different number of MAC units

Cutting the number of MACs might reduce chip power consumption as well as the chip area. However, it might also decrease the overall performance. There should be a compromise between them. Due to this, in this section, we will investigate the performance changes of SpiNNaker2 when reducing the size of the MAC array.

The default size of the MAC array is 16*4. It is convenient to reduce it to be half, and there are two choices, size of 16*2 and 8*4. When the size of the MAC array is changed, the fetching process of the operands also changes. For MAC array with 16*2, the operand A still fetches 128-bit every time and but the 128-bit data is used for 8 times for computation instead of 4 times (for convolution operation). For MAC array with 8*4, the operand B is changed to fetch 128-bit data per 2 clocks for the first time, and later 32-bit data per clock (for convolution operation). Whereas for matrix multiplication operation, the operand B always fetches 64-bit data.

Several representative convolution operations and matrix multiplications from VGG-16 and ResNet-50 are applied to compare their performance difference. Besides, the SpiNNaker2 simulator, the distribution algorithm using operator fusion and data reuse, and the same split scheme are used in this performance comparison.

Tables 5.11 and 5.12 show the clocks comparison results with different size of MAC array, whereas Tables 5.13 and 5.14 are the attainable performance results. For the attainable performance, Figures 5.12 and 5.13 provide a better intuitive visual on the performance degradation and comparison.

From the comparison result, it is clear that decreasing computing resources to be half, the attainable computing power is not half, and the degradation is below 1.5 times. In particular, there is almost no degradation when accelerating vector-matrix-multiplication. Instead, there is a slight improvement in the case of 16*2, because, for the MAC array with the size of 16*2, the output data amount increased by the alignment is decreased comparing to MAC array with 16*4, which saving the times on outputting the result from MLA to SRAM. Generally, MAC array with the size of 16*2 and 8*4 has higher computing resource utilization. They could alleviate the problem of insufficient bandwidth of SRAM and NoC, which makes difference components match each other better.

Operation	Clocks/Computation clocks (16*4)	Clocks/Computation clocks (16*2)	Clocks/Computation clocks (8*4)
CONV_1	119254/110383	124267/115398	125476/116608
CONV_7	214575/171892	327914/285233	376141/333456
CONV_9	413733/326286	639999/552554	736468/649021
FC_21	132044/-	131864/-	132816/-

Table 5.11: VGG-16: Clocks comparison between MLA with different number of MAC units

Operation	Clocks/Computation clocks (16*4)	Clocks/Computation clocks (16*2)	Clocks/Computation clocks (8*4)
CONV_1	99623/87591	168451/156422	170628/158599
CONV_SC_14	180906/89363	180906/154412	176935/150474
CONV_19	44987/21799	59216/36025	65233/42040
CONV_42	47116/10703	55317/18901	57366/20950

Table 5.12: ResNet-50: Clocks comparison between MLA with different number of MAC units

Operation	Operational intensity (operation/bytes)	Performance[16*4] (Gops)	Performance[16*2] (Gops)	Performance[8*4] (Gops)
CONV_1	51.51	363.53	348.86	345.50
CONV_7	1210.28	2155.06	1410.20	1246.84
CONV_9	1641.38	2235.36	1445.07	1267.11
FC_21	2.00	15.88	15.90	15.79

Table 5.13: VGG-16: Attainable performance comparison between MLA with different MAC units

Operation	Operational intensity (operation/bytes)	Performance[16*4] (Gops)	Performance[16*2] (Gops)	Performance[8*4] (Gops)
CONV_1	243.44	592.30	350.29	350.01
CONV_SC_14	153.91	445.98	284.02	290.03
CONV_19	636.93	1284.88	976.13	898.47
CONV_42	68.50	272.63	232.21	223.96

Table 5.14: ResNet-50: Attainable performance comparison between MLA with different MAC units

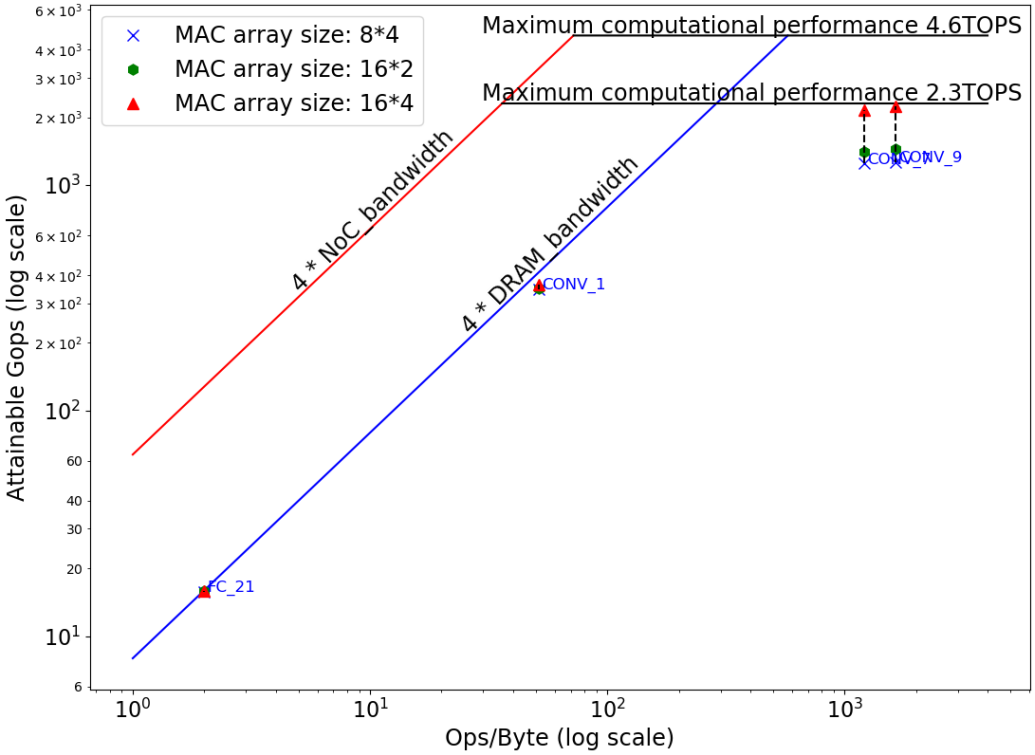

Figure 5.12: VGG-16: Roofline model for MLA with different number of MAC units

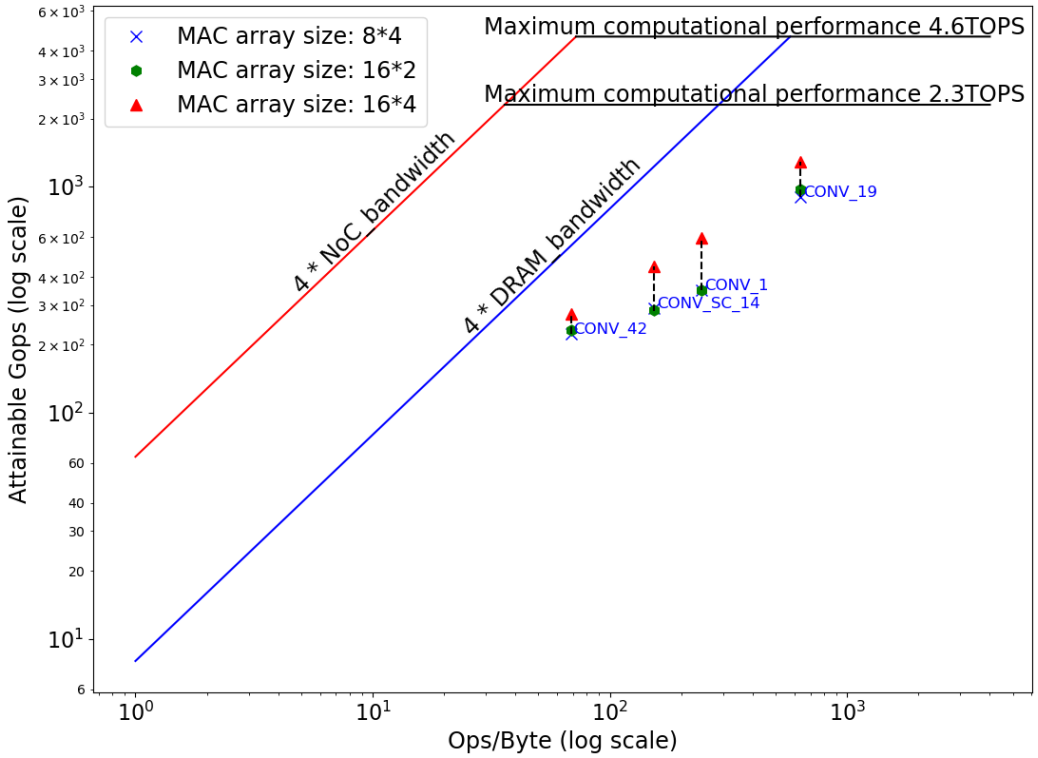

Figure 5.13: ResNet-50: Roofline model for MLA with different number of MAC units

5.5 Discussion

As subsection 5.2.2 and 5.3.2 show, the distribution algorithm with operator fusion and data reuse can improve the overall performance of SpiNNaker2 or QPE.

Comparing the simulation results of the algorithm without operator fusion and data reuse between QPE and SpiNNaker2, it could be found that the performance of SpiNNaker2 has only an improvement up to 4 times against QPE. This improvement is equivalent to the ratio of the number of DRAMs between SpiNNaker2 and QPE-DRAM. This indicates that DRAM bandwidth severely limits the performance of SpiNNaker2.

To reduce the limitations imposed by DRAM bandwidth, operator fusion is introduced to reduce large amounts of data movement. However, the performance of SpiNNaker2 is still mainly restricted by DRAM, because the overall attainable performance of SpiNNaker2 is only several times (about 3-10 times) that of QPE, while the computing resources of SpiNNaker2 are 36 times that of QPE.

In order to squeeze more performance out of SpiNNaker2, the data reuse algorithm was proposed. In data reuse, two methods are applied to solve the bandwidth problem of DRAM. One is that 4 QPEs in SpiNNaker2 are converted to be the storage QPEs to store the computed results, which can reduce the data transfer between DRAM and SRAM, and increase data transfer between SRAM and SRAM. The introduction of the storage QPEs moves the overall system memory bandwidth bottleneck from DRAM towards NoC (Data transfer between SRAM and SRAM is done by the NoC). The NoC bandwidth is 4 times to the DRAM bandwidth. Therefore the total memory bandwidth of these 4 storage QPEs is equal to that of 16 DRAMs, which means there are a total of 20 DRAMs in data reuse mode(considering the original 4 DRAMs). Another one is to further reduce the data transfer from DRAM to SRAM by partial data reuse through the NoC. As we have seen in Appendixes D and E, some PEs in SpiNNaker2 have the same input feature map or filter weight, which is done by copying data from SRAM to SRAM through the NoC. Moreover, weight migration is also done by NoC. Same to the first method, the second method improves the overall performance by moving the overall system memory bandwidth bottleneck from DRAM towards NoC. Finally, with operator fusion and data reuse, the overall attainable performance of SpiNNaker2 is about 20 - 30 times that of QPE. However, this performance improvement comes at a price, which is to sacrifice the computational power of the 4 storage QPEs.

From Figures 5.9 and 5.10, we can also see that the later proposed neural network (ResNet-50) has lower computational complexity, and the ceiling of the most operations is DRAM bandwidth rather than computing resources. Therefore, according to the comparison results in section 5.4, reducing the number of MAC units might be the right choice if we need to reduce the chip area or even power consumption.

Further, comparing the SRAM utilization in QPE, the SRAM utilization in SpiNNaker2 is lower, that is because every operation is split into as many parts as possible to use more computing resources parallel. Besides, there is a dummy space preserved for filter weight avoiding overwriting during data migration. However, the MAC utilization in each PE of them is the same, because the splitter always maximizes it during splitting.

6 Conclusion

SpiNNaker2 has multiple processing elements, and the processing elements are connected through the NoC to form a mesh network. In order to develop targeted parallel algorithms and evaluate the performance of SpiNNaker2, a SpiNNaker2 simulator, SpiNNakerPy, and different mapping algorithms for efficiently mapping neural networks to SpiNNaker2 are developed. The simulation results show that DRAM performance is the bottleneck of SpiNNaker, which is also consistent with hardware specifications. After applying the operator fusion and data reuse strategies, the limitations imposed by DRAM can be reduced, and the overall attainable performance has a significant improvement. However, for different computational intensities, the degree of improvement is not the same. In particular, for high computational intensities, such as convolution, the attainable performance can be significantly improved by utilizing NoC and SRAM. Whereas, for low computational intensities, such as matrix multiplication, the attainable performance is primarily limited by DRAM and cannot be improved by NoC or SRAM.

The mapper (responsible for the mapping strategy) and SpiNNaker2Py still have room for improvement. The mapper can be improved in three ways.

(1) The parser currently does not support the computational graph generated by the existing open source software libraries, such as TensorFlow. In order to speed up the deployment of the latest neural networks, the parser should seamlessly support the existing deep learning libraries.

(2) The splitter obtains the optimized split result based on a traversal algorithm, which searches for large spaces, consumes more time and requires manual adjustment of the results. In the future, a machine learning based search algorithm should be developed for searching for a better split result and saving time.

(3) The distributor is currently unable to squeeze the performance of spinnaker thoroughly, and for some operations, the PE utilization is still relatively low. New distribution should be developed based on the current bottlenecks.

For the simulator, SpiNNaker2Py, the shortcoming is the relatively long simulation time, and its code or software architecture needs to be optimized.

Appendix

A Input feature map splitting strategy

If we assume that the input feature map is splitted into totally C parts, where C is a constant, and the height and width of input feature map are H and W and splitted into h and w parts respectively, then the size of input feature map before splitting $SIZE_{before}$ is indicated with equation A.1.

$$SIZE_{before} = H * W. \tag{A.1}$$

Whereas the size after splitting $SIZE_{after}$ is

$$\begin{aligned} SIZE_{after} =& H * W \\ &+ (h-1) * W * (F\_H - S) \\ &+ (w-1) * H * (F\_W - S) \\ &- ((h-1) * (w-1)) * (F\_H - S) * (F\_W - S) \end{aligned} \tag{A.2}$$

where F_H, F_W and S are the height of filter, width of filter and stride respectively. Let equation A.2 be subtracted from equation A.1 to get the increased data size $SIZE_{increased}$.

$$\begin{aligned} SIZE_{increased} =& (h-1) * W * (F\_H - S) \\ &+ (w-1) * H * (F\_W - S) \\ &- ((h-1) * (w-1)) * (F\_H - S) * (F\_W - S) \end{aligned} \tag{A.3}$$

If it is assumed that $F\_H = F\_W = F$ and $W = H$, furthermore with the help of $C = w * h$, we get equation A.4 from equation A.3.

$$\begin{aligned} SIZE_{increased} =& (\frac{C}{w} - 1 + w - 1) * H * (F - S) \\ &- ((\frac{C}{w} - 1) * (w-1)) * (F - S)^2 \end{aligned} \tag{A.4}$$

The goal is to find the optimized w that minimize the $SIZE_{increased}$. Setting the gradient of $SIZE_{increased}$ with respect to w to be zero, gives the equation A.5.

$$w - \frac{C}{w^2} = 0 \tag{A.5}$$

Solving equation A.5, w has two values: $\sqrt{C}$ and $-\sqrt{C}$, where $w = \sqrt{C}$ is the only valid result. Furthermore, from $w = \sqrt{C}$, we get $h = \sqrt{C}$. This result indicates that in the case where you need to split it into K shares, if the width and height of feature map are tend to splitted into the same number of parts, the final increased data will be avoid to be too much.

B Matrix multiplication splitting strategy

It is assumed that the matrix multiplication is $A*B = C$, and the dimensions of matrix A, B and C are respectively $[H_A, W_A]$, $[H_B, W_B]$ and $[H_C, W_C]$, where H_x is the height, W_x represents the width, $W_A = H_B$, $H_A = H_C$ and $W_B = W_C$.

Furthermore, the matrix multiplication is assumed to be split into N parts, in addition to split along H_A, this could also be realized by splitting along H_B and W_B. However, splitting along W_B decreases the data reuse of A, whereas splitting along W_H will cause some computation to be accelerated by the ARM core, lowering the acceleration speed.

If W_B is split into X parts, then H_B will be split into $Y = \frac{N}{Y}$ parts. On the one hand, the decrement of the data reuse causes A need to be fetched into QPE or SpiNNaker2 more times. Explicitly, the matrix A need to be fetched for X times. On the other hand, the element-wise operation need to be done by the ARM core is the accumulate operation (or addition operation), and the computation amount of this addition operation is $W_B * (Y - 1)$.

The decrement of the data reuse can be overcome by employing the data reuse mode during distribution. However, it brings drawbacks. Using data reuse, the accumulate operation needs to be done after all the matrix multiplication subtasks are finished, and because the ARM core is weak in terms of computational performance, the accumulation will take a relatively long time. Moreover, in order to accomplish the accumulation, all data need to be re-transferred, which takes extra time.

When the operator fusion mode is used during distribution, the accumulation can be done in place after each matrix multiplication subtask without data movement. However, in operator fusion mode, the time used for the accumulation still cannot be hidden and the matrix A need to be fetched for X times.

For $H_A = 1$, as shown in equation 2.4, the computational intensity of the matrix multiplication is 2, which is pretty low. For accelerating, the PE loads data from DRAM and takes $T_{load} = \frac{W_B}{X} * \frac{H_B}{Y} / 16 * 2$ clocks, where 16 is 16 bytes (the bandwidth of DRAM), and 2 represent 2 clocks per DRAM operation. After then, the computation needs $T_{compute} = \frac{W_B}{X} * \frac{H_B}{Y} / columns\_of\_MLA$ clocks, where $columns\_of\_MLA$ represents that how many MAC units are

used (16 by default). It is obviously that $T_{load} = 2 * T_{compute}$. In operator fusion mode, if the clocks needed for moving computed results from MLA to SRAM T_{move} and the accumulation T_{acc} are less than $T_{compute}$, then 2 PEs are enough to utilize one DRAM memory bandwidth fully. If $T_{move} + T_{acc} > T_{compute}$, with more PEs, the DRAM memory bandwidth can also be fully utilized. Although the matrix A still need to be fetched for X times and cannot be hidden, it does not slow the acceleration too much due to that the size of A is pretty small compared to B.

Because both QPE and SpiNNaker2 have enough computing resources against memory bandwidth for matrix multiplication, in order to improve the speed of computing, the operator fusion mode is preferred. In order to reduce the slowdown due to the reduction of data reuse of A, the splitting scheme with larger X should be suppressed. Consequently, the splitter tends to split along H_B as more parts as possible.

C Input feature map reuse algorithm for QPE

Algorithm 1 QPE-DRAM distribution algorithm with operator fusion and feature map data reuse for convolution block

Input: feature map after splitting in DRAM: fmaps; filter after splitting in DRAM: filters;

$roundFmaps = ceil(length(fmaps)/4)$
$roundFilters = ceil(length(filters)/4)$
$remainingFmaps = length(fmaps)$
for $i \in [0, roundFmaps)$ **do**
    if remainingFmaps >= 4 **then**
        Loading 4 parts of fmaps to 4 PEs
        $dataReuseTimes = 4$
    else if remainingFmaps == 3 **then**
        Loading 3 parts of fmaps to the first 3 PEs
        $dataReuseTimes = 3$
    else if remainingFmaps == 2 **then**
        Loading 2 parts of fmaps to the first 2 PEs
        Copy fmaps from PE[0] to PE[2]
        Copy fmaps from PE[1] to PE[3]
        $dataReuseTimes = 2$
    else if remainingFmaps == 1 **then**
        Loading 1 parts of fmaps to the first 1 PE
        Copy fmaps from PE[0] to PE[1]
        Copy fmaps from PE[0] to PE[2]
        Copy fmaps from PE[1] to PE[3]
        $dataReuseTimes = 1$
    end if
    for $j \in [0, roundFilters)$ **do**
        Loading $filters[j * 4 : (j + 1) * 4]$ from DRAM into 4 PEs
        for $k \in [0, dataReuseTimes)$ **do**
            if $dataReuseTimes == 3$ **then**
                Copy fmaps from PE[k] to PE[3]
            end if
            Run 4 PEs to accelerate convolution block
            repeat
                Receiving response from QPE-DRAM
            until All PEs finish the work
        end for
    end for
end for

D Loading input feature map into SpiNNaker2

For the input feature map shown in Figure D.1, whose width and height are splitted into 2 and 4 parts, its final loading result is as shown in Figure D.2, where IX means the $X^t h$ part of the input feature map.

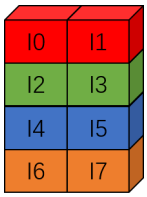

Figure D.1: Input feature map with 8 parts

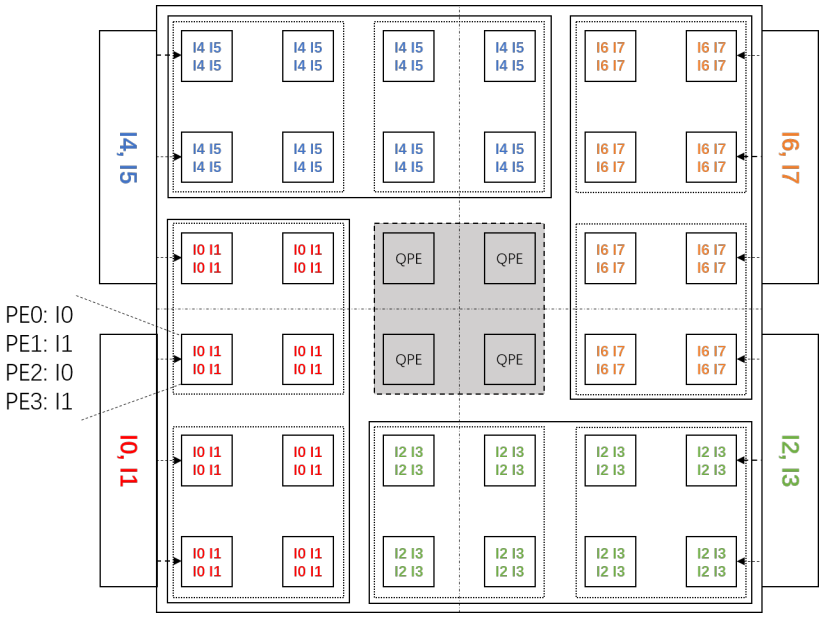

Figure D.2: Loading result of input feature map with 8 parts

For the input feature map shown in Figure D.3, whose width and height are splitted into 8 and 7 parts, its final loading result is as shown in Figure D.2, where IX-IY represents from the $X^t h$ part to the $Y^t h$ part of the input

feature map. Note that, in this case, there are 4 QPEs (16 PEs) have not input feature map, and they will idle even they have filter weight.

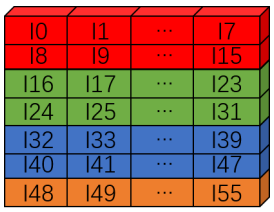

Figure D.3: Input feature map with 56 parts

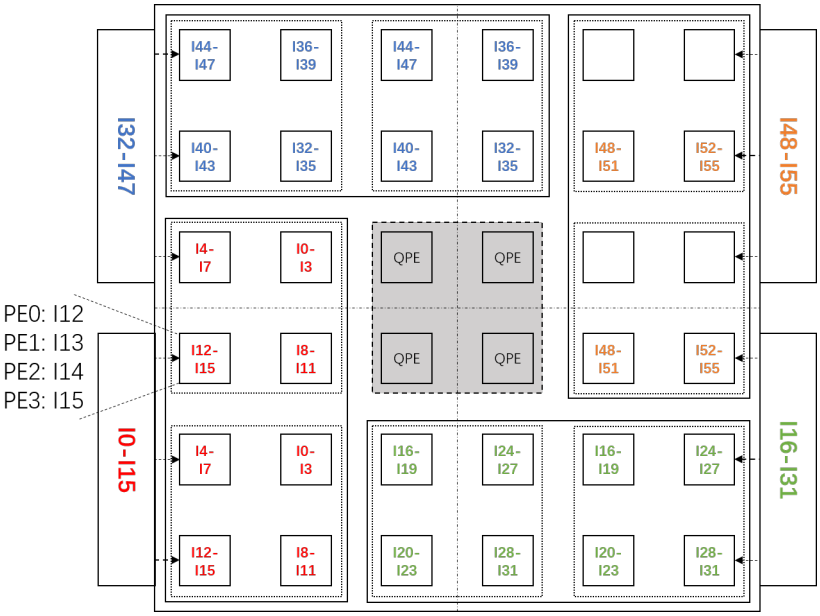

Figure D.4: Loading result of input feature map with 56 parts

E Loading filter weight into SpiNNaker2

Figure E.1, E.2, E.3, E.4 show how to load 16, 32, 64 and 128 parts of filter weight into SpiNNaker2 respectively, FX-FY represents from the X^th part to the Y^th part of the filter weight. Filter weights are evenly distributed across 4 DRAMs, ensuring that they are read into SpiNNaker2 as quickly as possible. For the cases, where the number of filter weights is less than 128, the distributor will instruct the SpiNNaker2 to filling the empty PEs through copying the filter weight from the non-empty PEs. This process is called filling-process, which ensures each PE has a filter weight.

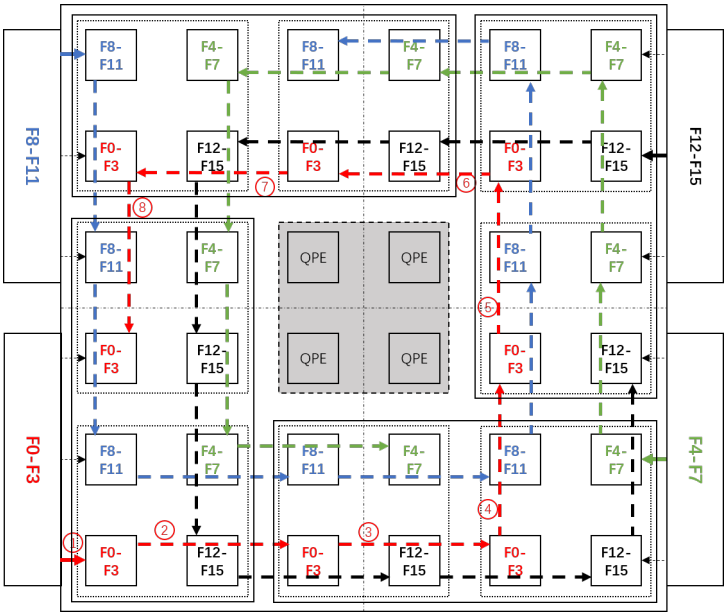

Figure E.1: Loading 16 parts of weight into SpiNNaker2

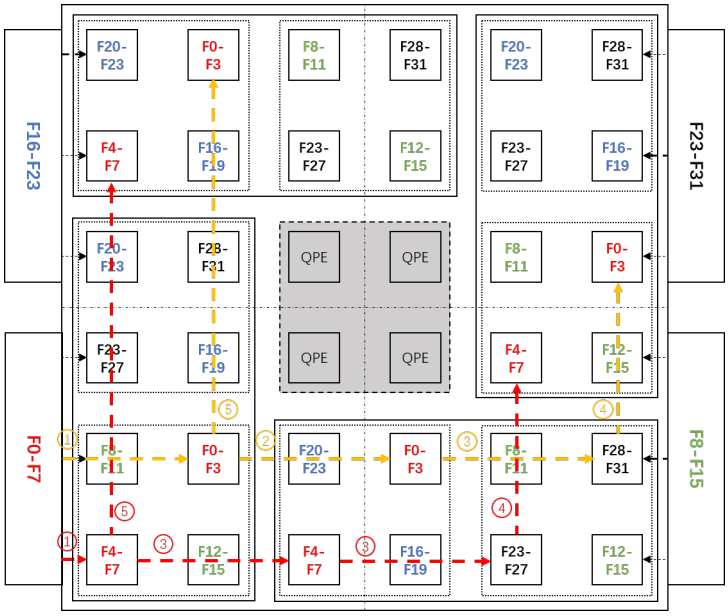

Figure E.2: Loading 32 parts of weight into SpiNNaker2

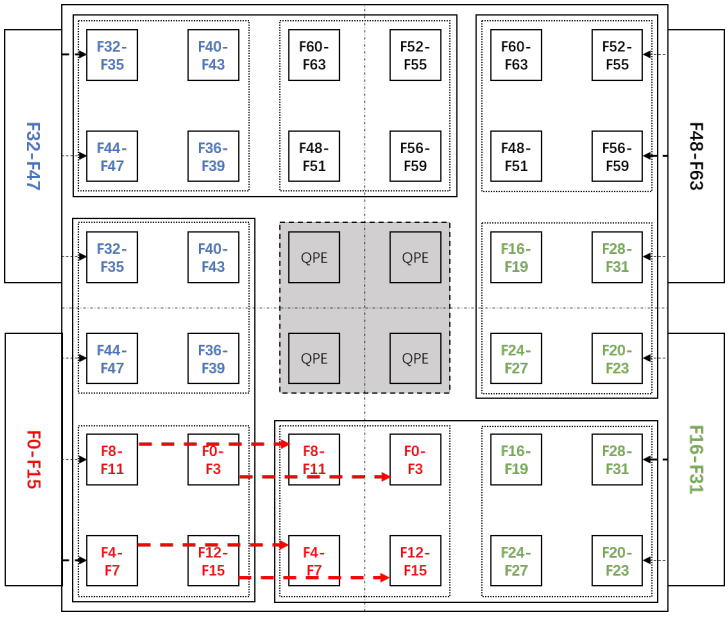

Figure E.3: Loading 64 parts of weight into SpiNNaker2

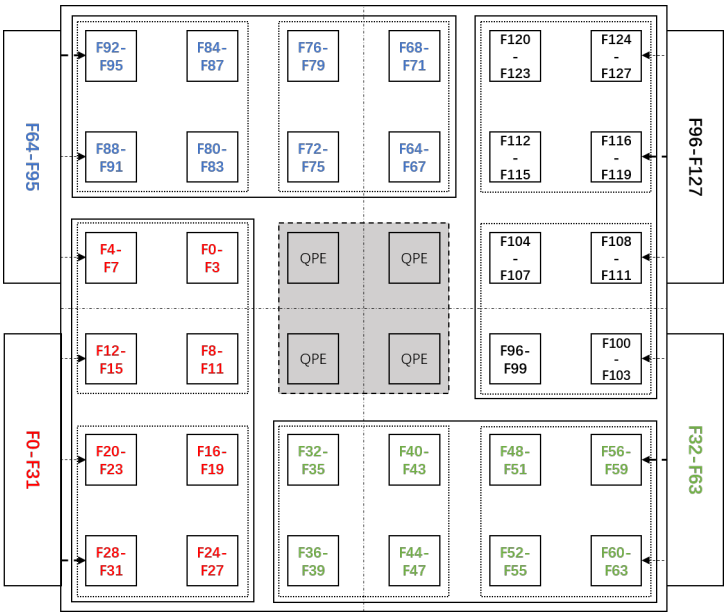

Figure E.4: Loading 128 parts of weight into SpiNNaker2

F Data reuse in SpiNNaker2

This appendix shows the data reuse and data migration process in SpiNNaker2 through a convolution example, as shown in Figure F.1. The input feature map **I** is split into 56 parts whereas the filter weight **F** with 256 output channels is split into 64 parts. This large convolution is decomposed into $56 * 64 = 3584$ small convolution tasks.

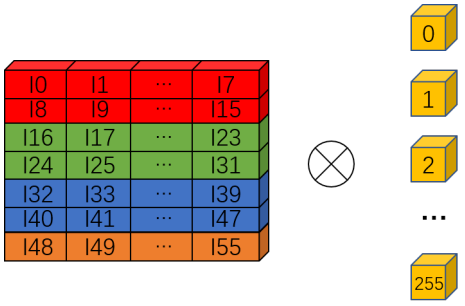

Figure F.1: A convolution with data reuse in SpiNNaker2. The filter weight with $C = 256$ is split into 64 parts, each part has 4 filter.

After loading the input feature map and the filter weight, the distribution of the input feature map **I** and filter weight **F** in SpiNNaker2 is shown in Figure F.1. There are 4 QPEs (16 PEs) have not input feature maps, which means they will be idle and there are a total of $28 * 4 = 112$ PEs are utilized to accelerate the convolutions. In this state (Step 1), through inside QPE partial data reuse, $112 * 4 = 448$ convolution tasks are accelerated.

After the first step, data migration inside QPE block is employed as shown in Figure F.3. After data migration, through inside QPE partial data reuse, another $112 * 4 = 448$ convolution tasks are accelerated (Step 2). Then data migration is employed again. The same as in step 2, another $112 * 4 = 448$ convolution tasks are finished (Step 3).

After Step 3, data migration is employed the third time, and the allocation of the data is shown in Figure F.4. In this state (Step 4), one more again, through inside QPE partial data reuse, another $112 * 4 = 448$ convolution tasks are accelerated. After 4 step acceleration, a total of $448 * 4 = 1792$ convolution task are accelerated.

Figure F.2: Data reuse step 1

After Step 4, data migration inside SpiNNaker2 is employed as shown in Figure F.5. With inside QPE partial data reuse, SpiNNaker2 can accelerate 448 convolution tasks (Step 5). Next, following the Step 2,3 and 4, the remaining convolution tasks can be finished.

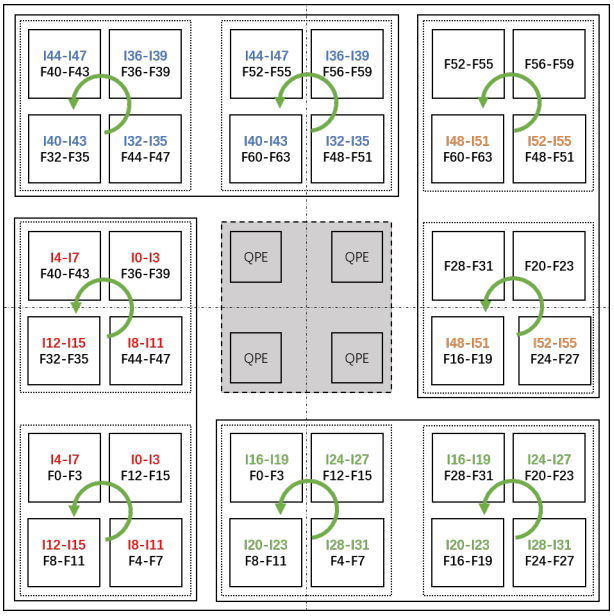

Figure F.3: Data reuse step 2

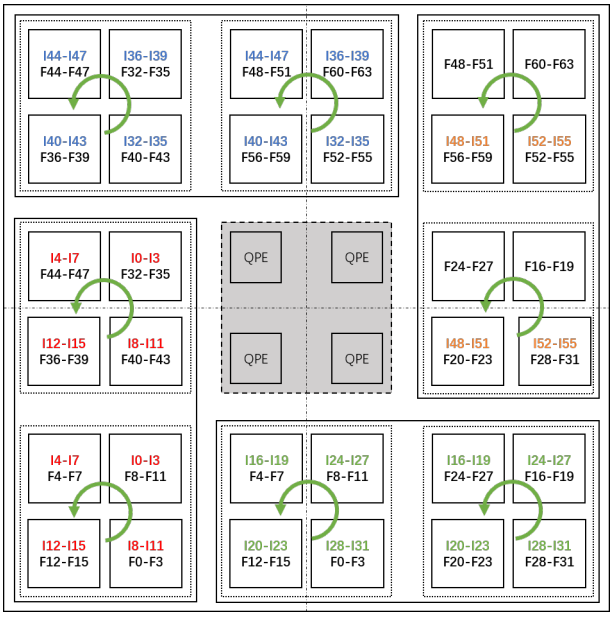

Figure F.4: Data reuse step 4

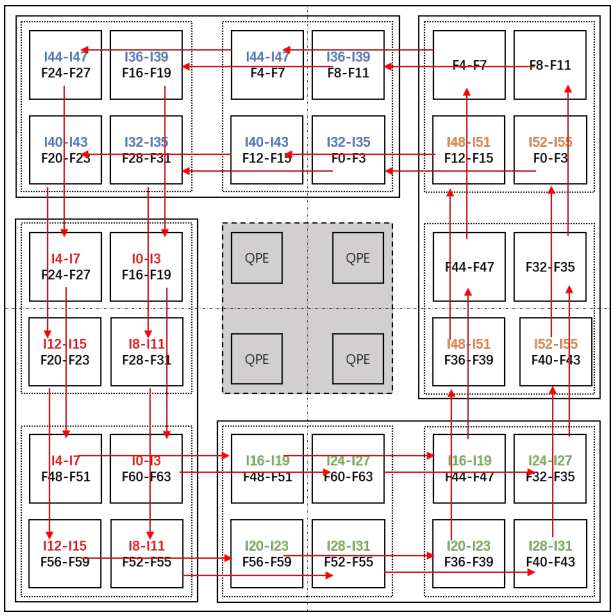

Figure F.5: Data reuse step 5

G VGG-16 split report for QPE-DRAM Simulator

Table G.1 is the split result of VGG-16 for QPE. The interpretations of the split results for different operation are showed below.

1. Padding operation: Represented by *PADD_x*. The input/output feature map dimension is [*width*, *height*, *channel*]. Each dimension will be splitted into multiple parts and represented as [$size_1$, $parts\ of\ size_1$, $size_2$, $parts\ of\ size_2$, ...]. The weight of padding is represented as *([padding size of left border, padding size of right border, padding size of upper border, padding size of below border], overlap size after splitting)*. Column *pieces* is not available for padding.

2. Convolution operation: Represented by *CONV_x*. The input/output feature map dimension is [*width*, *height*, *channel*] while the weight is represented as [[*width*, *height*, $channel_{in}$, $channel_{out}$], *stride*]. If one of the dimensions is decomposed, the decomposition result is represented as [$size_{large}$, $parts\ of\ size_{large}$, $size_{small}$, $parts\ of\ size_{small}$]. Column *pieces* means the number of decomposed tasks.

3. Matrix multiplication operation: Represented by *FC_x*. The dimension of input/output feature map and weight is [*width*, *height*]. When any dimension is decomposed, the decomposition result is represented as [$size_{large}$, $parts\ of\ size_{large}$, $size_{small}$, $parts\ of\ size_{small}$], where $size_{large} > size_{small}$. Column *pieces* means the number of decomposed tasks.

4. Quantization operation: Represented by *QUAN_x*. Column *Weight* has no meaning. The other columns has the same representations with convolution operation or matrix multiplication operation, depending on whether it is located in convolution block or matrix multiplication block.

5. Non-linearity operation: Represented by *ACTI_x*. Column *Weight* has no meaning. The other columns has the same representations with

convolution operation or matrix multiplication operation, depending on whether it is located in convolution block or matrix multiplication block.

6. Pooling operation: Represented by *POOL_x*. Column *Weight* is represented as $[[weight_{width}, weight_{height}], stride]$. The other columns has the same representations with convolution operation.

Table G.2 is the SRAM and MAC units utilization for the convolution operation and the matrix multiplication operation.

After splitting, convolution operation and matrix multiplication operation are decomposed into groups, which are shown in column *Partial Operation*. For convolution, after splitting, according to the feature map splitting result along the width and height, the split feature maps are divided into four groups, as shown in figure G.1. For the matrix operations, this grouping rule is applied to the weight. If the feature map or the weight is not split, there is no groups. This naming rules are applied in column *Partial Operation*.

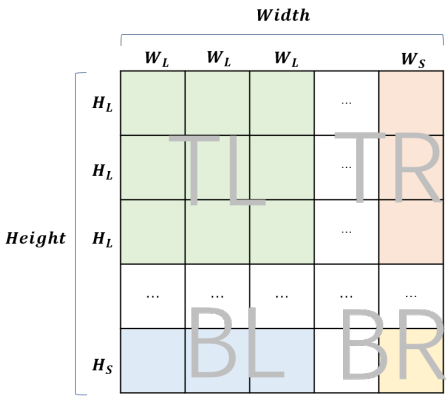

Figure G.1: Partial operation. TL: top left. TR: top right. BL: bottom left. BR: bottom right.

The valid data utilization of the input feature map, the weight and the output feature map are represented as below.

$$size\_after\_alignment[size\_before\_alignment(\frac{size\_before\_alignment}{size\_after\_alignment})]$$

Operation	Input Feature Map	Weight	Output Feature Map	Pieces
PADD_1	[[224, 1], [22, 1, 23, 3, 22, 6, 21, 1], 3]	([1, 1, 1, 1], 2)	[[226, 1, 0, 0], [23, 4, 22, 7], 3]	—
CONV_1	[[226, 1, 0, 0], [23, 4, 22, 7], 3]	[[3, 3, 3, [4, 16, 0, 0]], 1]	[[224, 1, 0, 0], [21, 4, 20, 7], [4, 16, 0, 0]]	176
ACTI_1	[[224, 1, 0, 0], [21, 4, 20, 7], [4, 16, 0, 0]]	[0]	[[224, 1, 0, 0], [21, 4, 20, 7], [4, 16, 0, 0]]	176
QUAN_1	[[224, 1, 0, 0], [21, 4, 20, 7], [4, 16, 0, 0]]	[0]	[[224, 1, 0, 0], [21, 4, 20, 7], [4, 16, 0, 0]]	176
PADD_2	[[113, 1, 113, 1], [7, 1, 8, 35, 6, 1, 5, 1], 64]	([1, 1, 1, 1], 2)	[[114, 2, 0, 0], [8, 36, 6, 2], 64]	—
CONV_2	[[114, 2, 0, 0], [8, 36, 6, 2], 64]	[[3, 3, 64, [4, 16, 0, 0]], 1]	[[112, 2, 0, 0], [6, 36, 4, 2], [4, 16, 0, 0]]	1216
ACTI_2	[[112, 2, 0, 0], [6, 36, 4, 2], [4, 16, 0, 0]]	[0]	[[112, 2, 0, 0], [6, 36, 4, 2], [4, 16, 0, 0]]	1216
QUAN_2	[[112, 2, 0, 0], [6, 36, 4, 2], [4, 16, 0, 0]]	[0]	[[112, 2, 0, 0], [6, 36, 4, 2], [4, 16, 0, 0]]	1216
POOL_3	[[112, 2, 0, 0], [6, 36, 4, 2], [4, 16, 0, 0]]	[[2, 2], 2]	[[56, 2, 0, 0], [3, 36, 2, 2], [4, 16, 0, 0]]	1216
PADD_4	[[112, 1], [8, 1, 9, 14, 8, 1], 64]	([1, 1, 1, 1], 2)	[[114, 1, 0, 0], [9, 16, 0, 0], 64]	—
CONV_4	[[114, 1, 0, 0], [9, 16, 0, 0], 64]	[[3, 3, 64, [4, 32, 0, 0]], 1]	[[112, 1, 0, 0], [7, 16, 0, 0], [4, 32, 0, 0]]	512
ACTI_4	[[112, 1, 0, 0], [7, 16, 0, 0], [4, 32, 0, 0]]	[0]	[[112, 1, 0, 0], [7, 16, 0, 0], [4, 32, 0, 0]]	512
QUAN_4	[[112, 1, 0, 0], [7, 16, 0, 0], [4, 32, 0, 0]]	[0]	[[112, 1, 0, 0], [7, 16, 0, 0], [4, 32, 0, 0]]	512
PADD_5	[[17, 1, 18, 5, 17, 1], [17, 1, 18, 5, 17, 1], 128]	([1, 1, 1, 1], 2)	[[18, 7, 0, 0], [18, 7, 0, 0], 128]	—
CONV_5	[[18, 7, 0, 0], [18, 7, 0, 0], 128]	[[3, 3, 128, [4, 32, 0, 0]], 1]	[[16, 7, 0, 0], [16, 7, 0, 0], [4, 32, 0, 0]]	1568
ACTI_5	[[16, 7, 0, 0], [16, 7, 0, 0], [4, 32, 0, 0]]	[0]	[[16, 7, 0, 0], [16, 7, 0, 0], [4, 32, 0, 0]]	1568
QUAN_5	[[16, 7, 0, 0], [16, 7, 0, 0], [4, 32, 0, 0]]	[0]	[[16, 7, 0, 0], [16, 7, 0, 0], [4, 32, 0, 0]]	1568
POOL_6	[[16, 7, 0, 0], [16, 7, 0, 0], [4, 32, 0, 0]]	[[2, 2], 2]	[[8, 7, 0, 0], [8, 7, 0, 0], [4, 32, 0, 0]]	1568
PADD_7	[[29, 1, 29, 1], [15, 1, 16, 2, 15, 1], 128]	([1, 1, 1, 1], 2)	[[30, 2, 0, 0], [16, 4, 0, 0], 128]	—
CONV_7	[[30, 2, 0, 0], [16, 4, 0, 0], 128]	[[3, 3, 128, [4, 64, 0, 0]], 1]	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	512
ACTI_7	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	[0]	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	512
QUAN_7	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	[0]	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	512
PADD_8	[[29, 1, 29, 1], [9, 1, 10, 5, 9, 1], 256]	([1, 1, 1, 1], 2)	[[30, 2, 0, 0], [10, 7, 0, 0], 256]	—
CONV_8	[[30, 2, 0, 0], [10, 7, 0, 0], 256]	[[3, 3, 256, [4, 64, 0, 0]], 1]	[[28, 2, 0, 0], [8, 7, 0, 0], [4, 64, 0, 0]]	896
ACTI_8	[[28, 2, 0, 0], [8, 7, 0, 0], [4, 64, 0, 0]]	[0]	[[28, 2, 0, 0], [8, 7, 0, 0], [4, 64, 0, 0]]	896
QUAN_8	[[28, 2, 0, 0], [8, 7, 0, 0], [4, 64, 0, 0]]	[0]	[[28, 2, 0, 0], [8, 7, 0, 0], [4, 64, 0, 0]]	896
PADD_9	[[29, 1, 29, 1], [9, 1, 10, 5, 9, 1], 256]	([1, 1, 1, 1], 2)	[[30, 2, 0, 0], [10, 7, 0, 0], 256]	—
CONV_9	[[30, 2, 0, 0], [10, 7, 0, 0], 256]	[[3, 3, 256, [4, 64, 0, 0]], 1]	[[28, 2, 0, 0], [8, 7, 0, 0], [4, 64, 0, 0]]	896
ACTI_9	[[28, 2, 0, 0], [8, 7, 0, 0], [4, 64, 0, 0]]	[0]	[[28, 2, 0, 0], [8, 7, 0, 0], [4, 64, 0, 0]]	896
QUAN_9	[[28, 2, 0, 0], [8, 7, 0, 0], [4, 64, 0, 0]]	[0]	[[28, 2, 0, 0], [8, 7, 0, 0], [4, 64, 0, 0]]	896
POOL_10	[[28, 2, 0, 0], [8, 7, 0, 0], [4, 64, 0, 0]]	[[2, 2], 2]	[[14, 2, 0, 0], [4, 7, 0, 0], [4, 64, 0, 0]]	896
PADD_11	[[28, 1], [8, 1, 9, 2, 8, 1], 256]	([1, 1, 1, 1], 2)	[[30, 1, 0, 0], [9, 4, 0, 0], 256]	—
CONV_11	[[30, 1, 0, 0], [9, 4, 0, 0], 256]	[[3, 3, 256, [4, 128, 0, 0]], 1]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 128, 0, 0]]	512
ACTI_11	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 128, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 128, 0, 0]]	512
QUAN_11	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 128, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 128, 0, 0]]	512
PADD_12	[[15, 1, 15, 1], [8, 1, 9, 2, 8, 1], 512]	([1, 1, 1, 1], 2)	[[16, 2, 0, 0], [9, 4, 0, 0], 512]	—
CONV_12	[[16, 2, 0, 0], [9, 4, 0, 0], 512]	[[3, 3, 512, [4, 128, 0, 0]], 1]	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 128, 0, 0]]	1024
ACTI_12	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 128, 0, 0]]	1024
QUAN_12	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 128, 0, 0]]	1024
PADD_13	[[15, 1, 15, 1], [7, 1, 8, 3, 5, 1], 512]	([1, 1, 1, 1], 2)	[[16, 2, 0, 0], [8, 4, 6, 1], 512]	—
CONV_13	[[16, 2, 0, 0], [8, 4, 6, 1], 512]	[[3, 3, 512, [4, 128, 0, 0]], 1]	[[14, 2, 0, 0], [6, 4, 4, 1], [4, 128, 0, 0]]	1280
ACTI_13	[[14, 2, 0, 0], [6, 4, 4, 1], [4, 128, 0, 0]]	[0]	[[14, 2, 0, 0], [6, 4, 4, 1], [4, 128, 0, 0]]	1280
QUAN_13	[[14, 2, 0, 0], [6, 4, 4, 1], [4, 128, 0, 0]]	[0]	[[14, 2, 0, 0], [6, 4, 4, 1], [4, 128, 0, 0]]	1280
POOL_14	[[14, 2, 0, 0], [6, 4, 4, 1], [4, 128, 0, 0]]	[[2, 2], 2]	[[7, 2, 0, 0], [3, 4, 2, 1], [4, 128, 0, 0]]	1280
PADD_15	[[14, 1], [8, 1, 8, 1], 512]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [9, 2, 0, 0], 512]	—
CONV_15	[[16, 1, 0, 0], [9, 2, 0, 0], 512]	[[3, 3, 512, [4, 128, 0, 0]], 1]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 128, 0, 0]]	256
ACTI_15	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 128, 0, 0]]	256
QUAN_15	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 128, 0, 0]]	256
PADD_16	[[14, 1], [8, 1, 8, 1], 512]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [9, 2, 0, 0], 512]	—
CONV_16	[[16, 1, 0, 0], [9, 2, 0, 0], 512]	[[3, 3, 512, [4, 128, 0, 0]], 1]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 128, 0, 0]]	256
ACTI_16	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 128, 0, 0]]	256
QUAN_16	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 128, 0, 0]]	256
PADD_17	[[14, 1], [3, 1, 4, 5, 3, 1], 512]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [4, 7, 0, 0], 512]	—
CONV_17	[[16, 1, 0, 0], [4, 7, 0, 0], 512]	[[3, 3, 512, [4, 128, 0, 0]], 1]	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	896
ACTI_17	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	896
QUAN_17	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	896
POOL_18	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	[[2, 2], 2]	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 128, 0, 0]]	896
FC_19	[[256, 98, 0, 0], 1]	[[256, 16, 0, 0], [256, 98, 0, 0]]	[[256, 16, 0, 0], 1]	1568
FCE_19	[[256, 16, 0, 0], 98]	[0]	[[256, 16, 0, 0], 1]	1552
ACTI_19	[[256, 16, 0, 0], 1]	[0]	[[256, 16, 0, 0], 1]	16
QUAN_19	[[256, 16, 0, 0], 1]	[0]	[[256, 16, 0, 0], 1]	16
FC_20	[[256, 16, 0, 0], 1]	[[256, 16, 0, 0], [256, 16, 0, 0]]	[[256, 16, 0, 0], 1]	256
FCE_20	[[256, 16, 0, 0], 16]	[0]	[[256, 16, 0, 0], 1]	240
ACTI_20	[[256, 16, 0, 0], 1]	[0]	[[256, 16, 0, 0], 1]	16
QUAN_20	[[256, 16, 0, 0], 1]	[0]	[[256, 16, 0, 0], 1]	16
FC_21	[[256, 16, 0, 0], 1]	[[256, 4, 0, 0], [256, 16, 0, 0]]	[[256, 4, 0, 0], 1]	64
FCE_21	[[256, 4, 0, 0], 16]	[0]	[[256, 4, 0, 0], 1]	60
ACTI_21	[[256, 4, 0, 0], 1]	[0]	[[256, 4, 0, 0], 1]	4
QUAN_21	[[256, 4, 0, 0], 1]	[0]	[[256, 4, 0, 0], 1]	4

Table G.1: VGG-16 split scheme for QPE-DRAM Simulator

Operation	Partial Operation	Pieces	Input Feature Map	Weight	Output Feature Map	MAC Utilization	SRAM Utilization
CONV_1	CONVSUTL_1	64	16560 [15594(94.17%)]	112 [108(96.43%)]	75264 [75264(100.00%)]	1.00	0.93
	CONVSUBL_1	112	15840 [14916(94.17%)]	112 [108(96.43%)]	71680 [71680(100.00%)]	1.00	0.88
CONV_2	CONVSUTL_2	1152	65536 [58368(89.06%)]	2304 [2304(100.00%)]	10752 [10752(100.00%)]	1.00	0.73
	CONVSUBL_2	64	49152 [43776(89.06%)]	2304 [2304(100.00%)]	7168 [7168(100.00%)]	1.00	0.54
CONV_4	CONVSUTL_4	512	73728 [65664(89.06%)]	2304 [2304(100.00%)]	12544 [12544(100.00%)]	1.00	0.82
CONV_5	CONVSUTL_5	1568	73728 [41472(56.25%)]	4608 [4608(100.00%)]	4096 [4096(100.00%)]	1.00	0.51
CONV_7	CONVSUTL_7	512	65536 [61440(93.75%)]	4608 [4608(100.00%)]	6272 [6272(100.00%)]	0.88	0.74
CONV_8	CONVSUTL_8	896	81920 [76800(93.75%)]	9216 [9216(100.00%)]	3584 [3584(100.00%)]	0.88	0.91
CONV_9	CONVSUTL_9	896	81920 [76800(93.75%)]	9216 [9216(100.00%)]	3584 [3584(100.00%)]	0.88	0.91
CONV_11	CONVSUTL_11	512	73728 [69120(93.75%)]	9216 [9216(100.00%)]	3136 [3136(100.00%)]	0.88	0.83
CONV_12	CONVSUTL_12	1024	73728 [73728(100.00%)]	18432 [18432(100.00%)]	1792 [1568(87.50%)]	0.88	0.95
CONV_13	CONVSUTL_13	1024	65536 [65536(100.00%)]	18432 [18432(100.00%)]	1536 [1344(87.50%)]	0.88	0.87
	CONVSUBL_13	256	49152 [49152(100.00%)]	18432 [18432(100.00%)]	1024 [896(87.50%)]	0.88	0.70
CONV_15	CONVSUTL_15	256	73728 [73728(100.00%)]	18432 [18432(100.00%)]	1792 [1568(87.50%)]	0.88	0.95
CONV_16	CONVSUTL_16	256	73728 [73728(100.00%)]	18432 [18432(100.00%)]	1792 [1568(87.50%)]	0.88	0.95
CONV_17	CONVSUTL_17	896	32768 [32768(100.00%)]	18432 [18432(100.00%)]	512 [448(87.50%)]	0.88	0.53
FC_19	FCTL_19	1568	1024 [256(25.00%)]	65536 [65536(100.00%)]	4096 [1024(25.00%)]	0.25	0.68
FC_20	FCTL_20	256	1024 [256(25.00%)]	65536 [65536(100.00%)]	4096 [1024(25.00%)]	0.25	0.68
FC_21	FCTL_21	64	1024 [256(25.00%)]	65536 [65536(100.00%)]	4096 [1024(25.00%)]	0.25	0.68

Table G.2: VGG-16: SRAM and MAC utilization in QPE-DRAM Simulator

H ResNet-50 split report for QPE-DRAM Simulator

Table H.1 is the split result of ResNet-50 for QPE. The interpretations of the split results for the shortcut operation [3] are showed below.

1. Convolution operation for shortcut: *CONV_SC_x* is a convolution operation, therefore, its interpretations of the split result is same as that of *CONV_x*. However, because it is used for projection shortcuts[3], there is no non-linearity operation applied after the convolution operation.

2. Matrix element-wise operation for shortcut: *MAT_ELE_x* represents for a matrix element-wise addition operation, which is used for identity shortcuts[3] or used after projection shortcuts *CONV_SC_x*. For the matrix element-wise operation, column *Weight* has no meaning and the other columns has the same interpretations with convolution operation.

Operation	Input Feature Map	Weight	Output Feature Map	Pieces
PADD_1	[[224, 1], [22, 1, 25, 7, 21, 3, 19, 1], 3]	([3, 2, 3, 2], 5)	[[229, 1, 0, 0], [25, 8, 21, 4], 3]	—
CONV_1	[[229, 1, 0, 0], [25, 8, 21, 4], 3]	[[7, 7, 3, [4, 16, 0, 0]], 2]	[[112, 1, 0, 0], [10, 8, 8, 4], [4, 16, 0, 0]]	192
ACTI_1	[[112, 1, 0, 0], [10, 8, 8, 4], [4, 16, 0, 0]]	[0]	[[112, 1, 0, 0], [10, 8, 8, 4], [4, 16, 0, 0]]	192
QUAN_1	[[112, 1, 0, 0], [10, 8, 8, 4], [4, 16, 0, 0]]	[0]	[[112, 1, 0, 0], [10, 8, 8, 4], [4, 16, 0, 0]]	192
PADD_2	[[112, 1], [112, 1], [6, 9, 5, 2]]	([1, 0, 1, 0], 1)	[[113, 1, 0, 0], [113, 1, 0, 0], [6, 9, 5, 2]]	—
POOL_2	[[113, 1, 0, 0], [113, 1, 0, 0], [6, 9, 5, 2]]	[[3, 3], 2]	[[56, 1, 0, 0], [56, 1, 0, 0], [6, 9, 5, 2]]	11
CONV_3	[[56, 1, 0, 0], [14, 4, 0, 0], 64]	[[1, 1, 64, [4, 16, 0, 0]], 1]	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	64
ACTI_3	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	64
QUAN_3	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	64
PADD_4	[[29, 1, 29, 1], [29, 1, 29, 1], 64]	([1, 1, 1, 1], 2)	[[30, 2, 0, 0], [30, 2, 0, 0], 64]	—
CONV_4	[[30, 2, 0, 0], [30, 2, 0, 0], 64]	[[3, 3, 64, [4, 16, 0, 0]], 1]	[[28, 2, 0, 0], [28, 2, 0, 0], [4, 16, 0, 0]]	64
ACTI_4	[[28, 2, 0, 0], [28, 2, 0, 0], [4, 16, 0, 0]]	[0]	[[28, 2, 0, 0], [28, 2, 0, 0], [4, 16, 0, 0]]	64
QUAN_4	[[28, 2, 0, 0], [28, 2, 0, 0], [4, 16, 0, 0]]	[0]	[[28, 2, 0, 0], [28, 2, 0, 0], [4, 16, 0, 0]]	64
CONV_5	[[56, 1, 0, 0], [14, 4, 0, 0], 64]	[[1, 1, 64, [4, 64, 0, 0]], 1]	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	256
CONV_SC_5	[[56, 1, 0, 0], [14, 4, 0, 0], 64]	[[1, 1, 64, [4, 64, 0, 0]], 1]	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	256
MAT_ELE_5	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	256
ACTI_5	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	256
QUAN_5	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	256
CONV_6	[[56, 1, 0, 0], [5, 8, 4, 4], 256]	[[1, 1, 256, [4, 16, 0, 0]], 1]	[[56, 1, 0, 0], [5, 8, 4, 4], [4, 16, 0, 0]]	192
ACTI_6	[[56, 1, 0, 0], [5, 8, 4, 4], [4, 16, 0, 0]]	[0]	[[56, 1, 0, 0], [5, 8, 4, 4], [4, 16, 0, 0]]	192
QUAN_6	[[56, 1, 0, 0], [5, 8, 4, 4], [4, 16, 0, 0]]	[0]	[[56, 1, 0, 0], [5, 8, 4, 4], [4, 16, 0, 0]]	192
PADD_7	[[29, 1, 29, 1], [29, 1, 29, 1], 64]	([1, 1, 1, 1], 2)	[[30, 2, 0, 0], [30, 2, 0, 0], 64]	—
CONV_7	[[30, 2, 0, 0], [30, 2, 0, 0], 64]	[[3, 3, 64, [4, 16, 0, 0]], 1]	[[28, 2, 0, 0], [28, 2, 0, 0], [4, 16, 0, 0]]	64
ACTI_7	[[28, 2, 0, 0], [28, 2, 0, 0], [4, 16, 0, 0]]	[0]	[[28, 2, 0, 0], [28, 2, 0, 0], [4, 16, 0, 0]]	64
QUAN_7	[[28, 2, 0, 0], [28, 2, 0, 0], [4, 16, 0, 0]]	[0]	[[28, 2, 0, 0], [28, 2, 0, 0], [4, 16, 0, 0]]	64
CONV_8	[[56, 1, 0, 0], [14, 4, 0, 0], 64]	[[1, 1, 64, [4, 64, 0, 0]], 1]	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	256
MAT_ELE_8	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	256
ACTI_8	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	256
QUAN_8	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	256
CONV_9	[[56, 1, 0, 0], [5, 8, 4, 4], 256]	[[1, 1, 256, [4, 16, 0, 0]], 1]	[[56, 1, 0, 0], [5, 8, 4, 4], [4, 16, 0, 0]]	192
ACTI_9	[[56, 1, 0, 0], [5, 8, 4, 4], [4, 16, 0, 0]]	[0]	[[56, 1, 0, 0], [5, 8, 4, 4], [4, 16, 0, 0]]	192
QUAN_9	[[56, 1, 0, 0], [5, 8, 4, 4], [4, 16, 0, 0]]	[0]	[[56, 1, 0, 0], [5, 8, 4, 4], [4, 16, 0, 0]]	192
PADD_10	[[29, 1, 29, 1], [29, 1, 29, 1], 64]	([1, 1, 1, 1], 2)	[[30, 2, 0, 0], [30, 2, 0, 0], 64]	—
CONV_10	[[30, 2, 0, 0], [30, 2, 0, 0], 64]	[[3, 3, 64, [4, 16, 0, 0]], 1]	[[28, 2, 0, 0], [28, 2, 0, 0], [4, 16, 0, 0]]	64

Operation	Input Feature Map	Weight	Output Feature Map	Pieces
ACTI_10	[[28, 2, 0, 0], [28, 2, 0, 0], [4, 16, 0, 0]]	[0]	[[28, 2, 0, 0], [28, 2, 0, 0], [4, 16, 0, 0]]	64
QUAN_10	[[28, 2, 0, 0], [28, 2, 0, 0], [4, 16, 0, 0]]	[0]	[[28, 2, 0, 0], [28, 2, 0, 0], [4, 16, 0, 0]]	64
CONV_11	[[56, 1, 0, 0], [14, 4, 0, 0], 64]	[[1, 1, 64, [4, 64, 0, 0]], 1]	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	256
MAT_ELE_11	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	256
ACTI_11	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	256
QUAN_11	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	256
CONV_12	[[56, 1, 0, 0], [2, 28, 0, 0], 256]	[[1, 1, 256, [4, 32, 0, 0]], 2]	[[28, 1, 0, 0], [1, 28, 0, 0], [4, 32, 0, 0]]	896
ACTI_12	[[28, 1, 0, 0], [1, 28, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [1, 28, 0, 0], [4, 32, 0, 0]]	896
QUAN_12	[[28, 1, 0, 0], [1, 28, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [1, 28, 0, 0], [4, 32, 0, 0]]	896
PADD_13	[[28, 1], [15, 1, 15, 1], 128]	([1, 1, 1, 1], 2)	[[30, 1, 0, 0], [16, 2, 0, 0], 128]	—
CONV_13	[[30, 1, 0, 0], [16, 2, 0, 0], 128]	[[3, 3, 128, [4, 32, 0, 0]], 1]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	64
ACTI_13	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	64
QUAN_13	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	64
CONV_14	[[28, 1, 0, 0], [14, 2, 0, 0], 128]	[[1, 1, 128, [4, 128, 0, 0]], 1]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	256
CONV_SC_14	[[56, 1, 0, 0], [2, 28, 0, 0], 256]	[[1, 1, 256, [4, 128, 0, 0]], 2]	[[28, 1, 0, 0], [1, 28, 0, 0], [4, 128, 0, 0]]	3584
MAT_ELE_14	[[28, 1, 0, 0], [1, 28, 0, 0], [4, 128, 0, 0]]	[0]	[[28, 1, 0, 0], [1, 28, 0, 0], [4, 128, 0, 0]]	3584
ACTI_14	[[28, 1, 0, 0], [1, 28, 0, 0], [4, 128, 0, 0]]	[0]	[[28, 1, 0, 0], [1, 28, 0, 0], [4, 128, 0, 0]]	3584
QUAN_14	[[28, 1, 0, 0], [1, 28, 0, 0], [4, 128, 0, 0]]	[0]	[[28, 1, 0, 0], [1, 28, 0, 0], [4, 128, 0, 0]]	3584
CONV_15	[[28, 1, 0, 0], [5, 4, 4, 2], 512]	[[1, 1, 512, [4, 32, 0, 0]], 1]	[[28, 1, 0, 0], [5, 4, 4, 2], [4, 32, 0, 0]]	192
ACTI_15	[[28, 1, 0, 0], [5, 4, 4, 2], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [5, 4, 4, 2], [4, 32, 0, 0]]	192
QUAN_15	[[28, 1, 0, 0], [5, 4, 4, 2], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [5, 4, 4, 2], [4, 32, 0, 0]]	192
PADD_16	[[28, 1], [15, 1, 15, 1], 128]	([1, 1, 1, 1], 2)	[[30, 1, 0, 0], [16, 2, 0, 0], 128]	—
CONV_16	[[30, 1, 0, 0], [16, 2, 0, 0], 128]	[[3, 3, 128, [4, 32, 0, 0]], 1]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	64
ACTI_16	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	64
QUAN_16	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	64
CONV_17	[[28, 1, 0, 0], [14, 2, 0, 0], 128]	[[1, 1, 128, [4, 128, 0, 0]], 1]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	256
MAT_ELE_17	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	[0]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	256
ACTI_17	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	[0]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	256
QUAN_17	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	[0]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	256
CONV_18	[[28, 1, 0, 0], [5, 4, 4, 2], 512]	[[1, 1, 512, [4, 32, 0, 0]], 1]	[[28, 1, 0, 0], [5, 4, 4, 2], [4, 32, 0, 0]]	192
ACTI_18	[[28, 1, 0, 0], [5, 4, 4, 2], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [5, 4, 4, 2], [4, 32, 0, 0]]	192
QUAN_18	[[28, 1, 0, 0], [5, 4, 4, 2], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [5, 4, 4, 2], [4, 32, 0, 0]]	192
PADD_19	[[28, 1], [15, 1, 15, 1], 128]	([1, 1, 1, 1], 2)	[[30, 1, 0, 0], [16, 2, 0, 0], 128]	—
CONV_19	[[30, 1, 0, 0], [16, 2, 0, 0], 128]	[[3, 3, 128, [4, 32, 0, 0]], 1]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	64
ACTI_19	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	64
QUAN_19	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	64
CONV_20	[[28, 1, 0, 0], [14, 2, 0, 0], 128]	[[1, 1, 128, [4, 128, 0, 0]], 1]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	256
MAT_ELE_20	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	[0]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	256
ACTI_20	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	[0]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	256
QUAN_20	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	[0]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	256
CONV_21	[[28, 1, 0, 0], [5, 4, 4, 2], 512]	[[1, 1, 512, [4, 32, 0, 0]], 1]	[[28, 1, 0, 0], [5, 4, 4, 2], [4, 32, 0, 0]]	192
ACTI_21	[[28, 1, 0, 0], [5, 4, 4, 2], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [5, 4, 4, 2], [4, 32, 0, 0]]	192
QUAN_21	[[28, 1, 0, 0], [5, 4, 4, 2], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [5, 4, 4, 2], [4, 32, 0, 0]]	192
PADD_22	[[28, 1], [15, 1, 15, 1], 128]	([1, 1, 1, 1], 2)	[[30, 1, 0, 0], [16, 2, 0, 0], 128]	—
CONV_22	[[30, 1, 0, 0], [16, 2, 0, 0], 128]	[[3, 3, 128, [4, 32, 0, 0]], 1]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	64
ACTI_22	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	64
QUAN_22	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 32, 0, 0]]	64
CONV_23	[[28, 1, 0, 0], [14, 2, 0, 0], 128]	[[1, 1, 128, [4, 128, 0, 0]], 1]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	256
MAT_ELE_23	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	[0]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	256
ACTI_23	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	[0]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	256
QUAN_23	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	[0]	[[28, 1, 0, 0], [14, 2, 0, 0], [4, 128, 0, 0]]	256
CONV_24	[[28, 1, 0, 0], [2, 14, 0, 0], 512]	[[1, 1, 512, [4, 64, 0, 0]], 2]	[[14, 1, 0, 0], [1, 14, 0, 0], [4, 64, 0, 0]]	896
ACTI_24	[[14, 1, 0, 0], [1, 14, 0, 0], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [4, 64, 0, 0]]	896
QUAN_24	[[14, 1, 0, 0], [1, 14, 0, 0], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [4, 64, 0, 0]]	896
PADD_25	[[14, 1], [14, 1], 256]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [16, 1, 0, 0], 256]	—
CONV_25	[[16, 1, 0, 0], [16, 1, 0, 0], 256]	[[3, 3, 256, [8, 32, 0, 0]], 1]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
ACTI_25	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
QUAN_25	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
CONV_26	[[14, 1, 0, 0], [14, 1, 0, 0], 256]	[[1, 1, 256, [32, 32, 0, 0]], 1]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
CONV_SC_26	[[28, 1, 0, 0], [2, 14, 0, 0], 512]	[[1, 1, 512, [4, 256, 0, 0]], 2]	[[14, 1, 0, 0], [1, 14, 0, 0], [4, 256, 0, 0]]	3584
MAT_ELE_26	[[14, 1, 0, 0], [1, 14, 0, 0], [4, 256, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [4, 256, 0, 0]]	3584
ACTI_26	[[14, 1, 0, 0], [1, 14, 0, 0], [4, 256, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [4, 256, 0, 0]]	3584
QUAN_26	[[14, 1, 0, 0], [1, 14, 0, 0], [4, 256, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [4, 256, 0, 0]]	3584
CONV_27	[[14, 1, 0, 0], [3, 4, 2, 1], 1024]	[[1, 1, 1024, [4, 64, 0, 0]], 1]	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	320
ACTI_27	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	320
QUAN_27	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	320
PADD_28	[[14, 1], [14, 1], 256]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [16, 1, 0, 0], 256]	—
CONV_28	[[16, 1, 0, 0], [16, 1, 0, 0], 256]	[[3, 3, 256, [8, 32, 0, 0]], 1]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
ACTI_28	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
QUAN_28	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
CONV_29	[[14, 1, 0, 0], [14, 1, 0, 0], 256]	[[1, 1, 256, [32, 32, 0, 0]], 1]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
MAT_ELE_29	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
ACTI_29	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
QUAN_29	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
CONV_30	[[14, 1, 0, 0], [3, 4, 2, 1], 1024]	[[1, 1, 1024, [4, 64, 0, 0]], 1]	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	320
ACTI_30	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	320
QUAN_30	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	320

Operation	Input Feature Map	Weight	Output Feature Map	Pieces
PADD_31	[[14, 1], [14, 1], 256]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [16, 1, 0, 0], 256]	——
CONV_31	[[16, 1, 0, 0], [16, 1, 0, 0], 256]	[[3, 3, 256, [8, 32, 0, 0]], 1]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
ACTI_31	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
QUAN_31	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
CONV_32	[[14, 1, 0, 0], [14, 1, 0, 0], 256]	[[1, 1, 256, [32, 32, 0, 0]], 1]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
MAT_ELE_32	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
ACTI_32	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
QUAN_32	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
CONV_33	[[14, 1, 0, 0], [3, 4, 2, 1], 1024]	[[1, 1, 1024, [4, 64, 0, 0]], 1]	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	320
ACTI_33	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	320
QUAN_33	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	320
PADD_34	[[14, 1], [14, 1], 256]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [16, 1, 0, 0], 256]	——
CONV_34	[[16, 1, 0, 0], [16, 1, 0, 0], 256]	[[3, 3, 256, [8, 32, 0, 0]], 1]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
ACTI_34	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
QUAN_34	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
CONV_35	[[14, 1, 0, 0], [14, 1, 0, 0], 256]	[[1, 1, 256, [32, 32, 0, 0]], 1]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
MAT_ELE_35	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
ACTI_35	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
QUAN_35	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
CONV_36	[[14, 1, 0, 0], [3, 4, 2, 1], 1024]	[[1, 1, 1024, [4, 64, 0, 0]], 1]	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	320
ACTI_36	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	320
QUAN_36	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	320
PADD_37	[[14, 1], [14, 1], 256]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [16, 1, 0, 0], 256]	——
CONV_37	[[16, 1, 0, 0], [16, 1, 0, 0], 256]	[[3, 3, 256, [8, 32, 0, 0]], 1]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
ACTI_37	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
QUAN_37	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
CONV_38	[[14, 1, 0, 0], [14, 1, 0, 0], 256]	[[1, 1, 256, [32, 32, 0, 0]], 1]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
MAT_ELE_38	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
ACTI_38	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
QUAN_38	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
CONV_39	[[14, 1, 0, 0], [3, 4, 2, 1], 1024]	[[1, 1, 1024, [4, 64, 0, 0]], 1]	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	320
ACTI_39	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	320
QUAN_39	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [3, 4, 2, 1], [4, 64, 0, 0]]	320
PADD_40	[[14, 1], [14, 1], 256]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [16, 1, 0, 0], 256]	——
CONV_40	[[16, 1, 0, 0], [16, 1, 0, 0], 256]	[[3, 3, 256, [8, 32, 0, 0]], 1]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
ACTI_40	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
QUAN_40	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [8, 32, 0, 0]]	32
CONV_41	[[14, 1, 0, 0], [14, 1, 0, 0], 256]	[[1, 1, 256, [32, 32, 0, 0]], 1]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
MAT_ELE_41	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
ACTI_41	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
QUAN_41	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [14, 1, 0, 0], [32, 32, 0, 0]]	32
CONV_42	[[14, 1, 0, 0], [2, 7, 0, 0], 1024]	[[1, 1, 1024, [4, 128, 0, 0]], 2]	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 128, 0, 0]]	896
ACTI_42	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 128, 0, 0]]	896
QUAN_42	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 128, 0, 0]]	896
PADD_43	[[7, 1], [7, 1], 512]	([1, 1, 1, 1], 2)	[[9, 1, 0, 0], [9, 1, 0, 0], 512]	——
CONV_43	[[9, 1, 0, 0], [9, 1, 0, 0], 512]	[[3, 3, 512, [4, 128, 0, 0]], 1]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
ACTI_43	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
QUAN_43	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
CONV_44	[[7, 1, 0, 0], [7, 1, 0, 0], 512]	[[1, 1, 512, [32, 64, 0, 0]], 1]	[[7, 1, 0, 0], [7, 1, 0, 0], [32, 64, 0, 0]]	64
CONV_SC_44	[[14, 1, 0, 0], [2, 7, 0, 0], 1024]	[[1, 1, 1024, [4, 512, 0, 0]], 2]	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 512, 0, 0]]	3584
MAT_ELE_44	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 512, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 512, 0, 0]]	3584
ACTI_44	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 512, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 512, 0, 0]]	3584
QUAN_44	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 512, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 512, 0, 0]]	3584
CONV_45	[[7, 1, 0, 0], [1, 7, 0, 0], 2048]	[[1, 1, 2048, [4, 128, 0, 0]], 1]	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 128, 0, 0]]	896
ACTI_45	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 128, 0, 0]]	896
QUAN_45	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 128, 0, 0]]	896
PADD_46	[[7, 1], [7, 1], 512]	([1, 1, 1, 1], 2)	[[9, 1, 0, 0], [9, 1, 0, 0], 512]	——
CONV_46	[[9, 1, 0, 0], [9, 1, 0, 0], 512]	[[3, 3, 512, [4, 128, 0, 0]], 1]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
ACTI_46	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
QUAN_46	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
CONV_47	[[7, 1, 0, 0], [7, 1, 0, 0], 512]	[[1, 1, 512, [32, 64, 0, 0]], 1]	[[7, 1, 0, 0], [7, 1, 0, 0], [32, 64, 0, 0]]	64
MAT_ELE_47	[[7, 1, 0, 0], [7, 1, 0, 0], [32, 64, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [32, 64, 0, 0]]	64
ACTI_47	[[7, 1, 0, 0], [7, 1, 0, 0], [32, 64, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [32, 64, 0, 0]]	64
QUAN_47	[[7, 1, 0, 0], [7, 1, 0, 0], [32, 64, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [32, 64, 0, 0]]	64
CONV_48	[[7, 1, 0, 0], [1, 7, 0, 0], 2048]	[[1, 1, 2048, [4, 128, 0, 0]], 1]	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 128, 0, 0]]	896
ACTI_48	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 128, 0, 0]]	896
QUAN_48	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 128, 0, 0]]	896
PADD_49	[[7, 1], [7, 1], 512]	([1, 1, 1, 1], 2)	[[9, 1, 0, 0], [9, 1, 0, 0], 512]	——
CONV_49	[[9, 1, 0, 0], [9, 1, 0, 0], 512]	[[3, 3, 512, [4, 128, 0, 0]], 1]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
ACTI_49	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
QUAN_49	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
CONV_50	[[7, 1, 0, 0], [7, 1, 0, 0], 512]	[[1, 1, 512, [32, 64, 0, 0]], 1]	[[7, 1, 0, 0], [7, 1, 0, 0], [32, 64, 0, 0]]	64
MAT_ELE_50	[[7, 1, 0, 0], [7, 1, 0, 0], [32, 64, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [32, 64, 0, 0]]	64
ACTI_50	[[7, 1, 0, 0], [7, 1, 0, 0], [32, 64, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [32, 64, 0, 0]]	64
QUAN_50	[[7, 1, 0, 0], [7, 1, 0, 0], [32, 64, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [32, 64, 0, 0]]	64
POOL_51	[[7, 1, 0, 0], [7, 1, 0, 0], [32, 64, 0, 0]]	[[7, 7], 7]	[[1, 1, 0, 0], [1, 1, 0, 0], [32, 64, 0, 0]]	64
FC_52	[[256, 8, 0, 0], 1]	[[256, 4, 0, 0], [256, 8, 0, 0]]	[[256, 4, 0, 0], 1]	32

Operation	Input Feature Map	Weight	Output Feature Map	Pieces
FCE_52	[[256, 4, 0, 0], 8]	[0]	[[256, 4, 0, 0], 1]	28
ACTI_52	[[256, 4, 0, 0], 1]	[0]	[[256, 4, 0, 0], 1]	4
QUAN_52	[[256, 4, 0, 0], 1]	[0]	[[256, 4, 0, 0], 1]	4

Table H.1: ResNet-50: Splits scheme for QPE-DRAM Simulator

Operation	Partial Operation	Pieces	Input Feature Map	Filter	Output Feature Map	MAC Utilization	SRAM Utilization
CONV_1	CONVSUTL_1	128	18000 [17175(95.42%)]	592 [588(99.32%)]	68096 [17920(26.32%)]	0.25	0.36
	CONVSUBL_1	64	15120 [14427(95.42%)]	592 [588(99.32%)]	14336 [14336(100.00%)]	0.25	0.30
POOL_2	POOLTLF_2	9	76614 [76614(100.00%)]	0 [0(0.00%)]	18816 [18816(100.00%)]	—	0.97
	POOLTLB_2	2	63845 [63845(100.00%)]	0 [0(0.00%)]	15680 [15680(100.00%)]	—	0.81
CONV_3	CONVSUTL_3	64	57344 [50176(87.50%)]	256 [256(100.00%)]	12544 [12544(100.00%)]	0.88	0.64
CONV_4	CONVSUTL_4	64	61440 [57600(93.75%)]	2304 [2304(100.00%)]	12544 [12544(100.00%)]	0.88	0.74
CONV_5	CONVSUTL_5	256	57344 [50176(87.50%)]	256 [256(100.00%)]	12544 [12544(100.00%)]	0.88	0.64
CONV_SC_5	CONVSUTL_SC_5	256	57344 [50176(87.50%)]	256 [256(100.00%)]	12544 [12544(100.00%)]	0.88	0.64
CONV_6	CONVSUTL_6	128	81920 [71680(87.50%)]	1024 [1024(100.00%)]	4480 [4480(100.00%)]	0.88	0.79
	CONVSUBL_6	64	65536 [57344(87.50%)]	1024 [1024(100.00%)]	3584 [3584(100.00%)]	0.88	0.63
CONV_7	CONVSUTL_7	64	61440 [57600(93.75%)]	2304 [2304(100.00%)]	12544 [12544(100.00%)]	0.88	0.74
CONV_8	CONVSUTL_8	256	57344 [50176(87.50%)]	256 [256(100.00%)]	12544 [12544(100.00%)]	0.88	0.64
CONV_9	CONVSUTL_9	128	81920 [71680(87.50%)]	1024 [1024(100.00%)]	4480 [4480(100.00%)]	0.88	0.79
	CONVSUBL_9	64	65536 [57344(87.50%)]	1024 [1024(100.00%)]	3584 [3584(100.00%)]	0.88	0.63
CONV_10	CONVSUTL_10	64	61440 [57600(93.75%)]	2304 [2304(100.00%)]	12544 [12544(100.00%)]	0.88	0.74
CONV_11	CONVSUTL_11	256	57344 [50176(87.50%)]	256 [256(100.00%)]	12544 [12544(100.00%)]	0.88	0.64
CONV_12	CONVSUTL_12	896	32768 [28672(87.50%)]	1024 [1024(100.00%)]	896 [448(50.00%)]	0.22	0.31
CONV_13	CONVSUTL_13	64	65536 [61440(93.75%)]	4608 [4608(100.00%)]	6272 [6272(100.00%)]	0.88	0.74
CONV_14	CONVSUTL_14	256	57344 [50176(87.50%)]	512 [512(100.00%)]	6272 [6272(100.00%)]	0.88	0.58
CONV_SC_14	CONVSUTL_SC_14	3584	32768 [28672(87.50%)]	1024 [1024(100.00%)]	896 [448(50.00%)]	0.22	0.31
CONV_15	CONVSUTL_15	128	81920 [71680(87.50%)]	2048 [2048(100.00%)]	2240 [2240(100.00%)]	0.88	0.77
	CONVSUBL_15	64	65536 [57344(87.50%)]	2048 [2048(100.00%)]	1792 [1792(100.00%)]	0.88	0.62
CONV_16	CONVSUTL_16	64	65536 [61440(93.75%)]	4608 [4608(100.00%)]	6272 [6272(100.00%)]	0.88	0.74
CONV_17	CONVSUTL_17	256	57344 [50176(87.50%)]	512 [512(100.00%)]	6272 [6272(100.00%)]	0.88	0.58
CONV_18	CONVSUTL_18	128	81920 [71680(87.50%)]	2048 [2048(100.00%)]	2240 [2240(100.00%)]	0.88	0.77
	CONVSUBL_18	64	65536 [57344(87.50%)]	2048 [2048(100.00%)]	1792 [1792(100.00%)]	0.88	0.62
CONV_19	CONVSUTL_19	64	65536 [61440(93.75%)]	4608 [4608(100.00%)]	6272 [6272(100.00%)]	0.88	0.74
CONV_20	CONVSUTL_20	256	57344 [50176(87.50%)]	512 [512(100.00%)]	6272 [6272(100.00%)]	0.88	0.58
CONV_21	CONVSUTL_21	128	81920 [71680(87.50%)]	2048 [2048(100.00%)]	2240 [2240(100.00%)]	0.88	0.77
	CONVSUBL_21	64	65536 [57344(87.50%)]	2048 [2048(100.00%)]	1792 [1792(100.00%)]	0.88	0.62
CONV_22	CONVSUTL_22	64	65536 [61440(93.75%)]	4608 [4608(100.00%)]	6272 [6272(100.00%)]	0.88	0.74
CONV_23	CONVSUTL_23	256	57344 [50176(87.50%)]	512 [512(100.00%)]	6272 [6272(100.00%)]	0.88	0.58
CONV_24	CONVSUTL_24	896	32768 [28672(87.50%)]	2048 [2048(100.00%)]	448 [224(50.00%)]	0.22	0.31
CONV_25	CONVSU_25	32	65536 [65536(100.00%)]	18432 [18432(100.00%)]	7168 [6272(87.50%)]	0.88	0.92
CONV_26	CONVSU_26	32	57344 [50176(87.50%)]	8192 [8192(100.00%)]	28672 [25088(87.50%)]	0.88	0.85
CONV_SC_26	CONVSUTL_SC_26	3584	32768 [28672(87.50%)]	2048 [2048(100.00%)]	448 [224(50.00%)]	0.22	0.31
CONV_27	CONVSUTL_27	256	49152 [43008(87.50%)]	4096 [4096(100.00%)]	768 [672(87.50%)]	0.88	0.49
	CONVSUBL_27	64	32768 [28672(87.50%)]	4096 [4096(100.00%)]	512 [448(87.50%)]	0.88	0.34
CONV_28	CONVSU_28	32	65536 [65536(100.00%)]	18432 [18432(100.00%)]	7168 [6272(87.50%)]	0.88	0.92
CONV_29	CONVSU_29	32	57344 [50176(87.50%)]	8192 [8192(100.00%)]	28672 [25088(87.50%)]	0.88	0.85
CONV_30	CONVSUTL_30	256	49152 [43008(87.50%)]	4096 [4096(100.00%)]	768 [672(87.50%)]	0.88	0.49
	CONVSUBL_30	64	32768 [28672(87.50%)]	4096 [4096(100.00%)]	512 [448(87.50%)]	0.88	0.34
CONV_31	CONVSU_31	32	65536 [65536(100.00%)]	18432 [18432(100.00%)]	7168 [6272(87.50%)]	0.88	0.92
CONV_32	CONVSU_32	32	57344 [50176(87.50%)]	8192 [8192(100.00%)]	28672 [25088(87.50%)]	0.88	0.85
CONV_33	CONVSUTL_33	256	49152 [43008(87.50%)]	4096 [4096(100.00%)]	768 [672(87.50%)]	0.88	0.49
	CONVSUBL_33	64	32768 [28672(87.50%)]	4096 [4096(100.00%)]	512 [448(87.50%)]	0.88	0.34
CONV_34	CONVSU_34	32	65536 [65536(100.00%)]	18432 [18432(100.00%)]	7168 [6272(87.50%)]	0.88	0.92
CONV_35	CONVSU_35	32	57344 [50176(87.50%)]	8192 [8192(100.00%)]	28672 [25088(87.50%)]	0.88	0.85
CONV_36	CONVSUTL_36	256	49152 [43008(87.50%)]	4096 [4096(100.00%)]	768 [672(87.50%)]	0.88	0.49
	CONVSUBL_36	64	32768 [28672(87.50%)]	4096 [4096(100.00%)]	512 [448(87.50%)]	0.88	0.34
CONV_37	CONVSU_37	32	65536 [65536(100.00%)]	18432 [18432(100.00%)]	7168 [6272(87.50%)]	0.88	0.92
CONV_38	CONVSU_38	32	57344 [50176(87.50%)]	8192 [8192(100.00%)]	28672 [25088(87.50%)]	0.88	0.85
CONV_39	CONVSUTL_39	256	49152 [43008(87.50%)]	4096 [4096(100.00%)]	768 [672(87.50%)]	0.88	0.49
	CONVSUBL_39	64	32768 [28672(87.50%)]	4096 [4096(100.00%)]	512 [448(87.50%)]	0.88	0.34
CONV_40	CONVSU_40	32	65536 [65536(100.00%)]	18432 [18432(100.00%)]	7168 [6272(87.50%)]	0.88	0.92
CONV_41	CONVSU_41	32	57344 [50176(87.50%)]	8192 [8192(100.00%)]	28672 [25088(87.50%)]	0.88	0.85
CONV_42	CONVSUTL_42	896	32768 [28672(87.50%)]	4096 [4096(100.00%)]	256 [112(43.75%)]	0.11	0.33
CONV_43	CONVSU_43	128	73728 [41472(56.25%)]	18432 [18432(100.00%)]	896 [784(87.50%)]	0.44	0.62
CONV_44	CONVSU_44	64	57344 [25088(43.75%)]	16384 [16384(100.00%)]	7168 [6272(87.50%)]	0.44	0.49
CONV_SC_44	CONVSUTL_SC_44	3584	32768 [28672(87.50%)]	4096 [4096(100.00%)]	256 [112(43.75%)]	0.11	0.33
CONV_45	CONVSUTL_45	896	32768 [14336(43.75%)]	8192 [8192(100.00%)]	128 [112(87.50%)]	0.44	0.23
CONV_46	CONVSU_46	128	73728 [41472(56.25%)]	18432 [18432(100.00%)]	896 [784(87.50%)]	0.44	0.62
CONV_47	CONVSU_47	64	57344 [25088(43.75%)]	16384 [16384(100.00%)]	7168 [6272(87.50%)]	0.44	0.49
CONV_48	CONVSUTL_48	896	32768 [14336(43.75%)]	8192 [8192(100.00%)]	128 [112(87.50%)]	0.44	0.23
CONV_49	CONVSU_49	128	73728 [41472(56.25%)]	18432 [18432(100.00%)]	896 [784(87.50%)]	0.44	0.62
CONV_50	CONVSU_50	64	57344 [25088(43.75%)]	16384 [16384(100.00%)]	7168 [6272(87.50%)]	0.44	0.49
FC_51	FCTL_51	1568	1024 [256(25.00%)]	65536 [65536(100.00%)]	4096 [1024(25.00%)]	0.25	0.68

Table H.2: ResNet-50: SRAM and MAC utilization in QPE-DRAM Simulator. MLA could not accelerate the pooling operation, therefore the MAC unit utilization is not available.

I VGG-16 split report for SpiNNaker2 Simulator

Operation	Input Feature Map	Weight	Output Feature Map	Pieces
PADD_1	[[224, 1], [15, 1, 16, 14, 15, 1], 3]	([1, 1, 1, 1], 2)	[[226, 1, 0, 0], [16, 16, 0, 0], 3]	—
CONV_1	[[226, 1, 0, 0], [16, 16, 0, 0], 3]	[[3, 3, 3, [4, 16, 0, 0]], 1]	[[224, 1, 0, 0], [14, 16, 0, 0], [4, 16, 0, 0]]	256
ACTI_1	[[224, 1, 0, 0], [14, 16, 0, 0], [4, 16, 0, 0]]	[0]	[[224, 1, 0, 0], [14, 16, 0, 0], [4, 16, 0, 0]]	256
QUAN_1	[[224, 1, 0, 0], [14, 16, 0, 0], [4, 16, 0, 0]]	[0]	[[224, 1, 0, 0], [14, 16, 0, 0], [4, 16, 0, 0]]	256
PADD_2	[[33, 1, 34, 5, 33, 1], [17, 1, 18, 12, 17, 1], 64]	([1, 1, 1, 1], 2)	[[34, 7, 0, 0], [18, 14, 0, 0], 64]	—
CONV_2	[[34, 7, 0, 0], [18, 14, 0, 0], 64]	[[3, 3, 64, [4, 16, 0, 0]], 1]	[[32, 7, 0, 0], [16, 14, 0, 0], [4, 16, 0, 0]]	1568
ACTI_2	[[32, 7, 0, 0], [16, 14, 0, 0], [4, 16, 0, 0]]	[0]	[[32, 7, 0, 0], [16, 14, 0, 0], [4, 16, 0, 0]]	1568
QUAN_2	[[32, 7, 0, 0], [16, 14, 0, 0], [4, 16, 0, 0]]	[0]	[[32, 7, 0, 0], [16, 14, 0, 0], [4, 16, 0, 0]]	1568
POOL_3	[[32, 7, 0, 0], [16, 14, 0, 0], [4, 16, 0, 0]]	[[2, 2], 2]	[[16, 7, 0, 0], [8, 14, 0, 0], [4, 16, 0, 0]]	1568
PADD_4	[[112, 1], [8, 1, 9, 14, 8, 1], 64]	([1, 1, 1, 1], 2)	[[114, 1, 0, 0], [9, 16, 0, 0], 64]	—
CONV_4	[[114, 1, 0, 0], [9, 16, 0, 0], 64]	[[3, 3, 64, [4, 32, 0, 0]], 1]	[[112, 1, 0, 0], [7, 16, 0, 0], [4, 32, 0, 0]]	512
ACTI_4	[[112, 1, 0, 0], [7, 16, 0, 0], [4, 32, 0, 0]]	[0]	[[112, 1, 0, 0], [7, 16, 0, 0], [4, 32, 0, 0]]	512
QUAN_4	[[112, 1, 0, 0], [7, 16, 0, 0], [4, 32, 0, 0]]	[0]	[[112, 1, 0, 0], [7, 16, 0, 0], [4, 32, 0, 0]]	512
PADD_5	[[112, 1], [3, 1, 4, 54, 3, 1], 128]	([1, 1, 1, 1], 2)	[[114, 1, 0, 0], [4, 56, 0, 0], 128]	—
CONV_5	[[114, 1, 0, 0], [4, 56, 0, 0], 128]	[[3, 3, 128, [4, 32, 0, 0]], 1]	[[112, 1, 0, 0], [2, 56, 0, 0], [4, 32, 0, 0]]	1792
ACTI_5	[[112, 1, 0, 0], [2, 56, 0, 0], [4, 32, 0, 0]]	[0]	[[112, 1, 0, 0], [2, 56, 0, 0], [4, 32, 0, 0]]	1792
QUAN_5	[[112, 1, 0, 0], [2, 56, 0, 0], [4, 32, 0, 0]]	[0]	[[112, 1, 0, 0], [2, 56, 0, 0], [4, 32, 0, 0]]	1792
POOL_6	[[112, 1, 0, 0], [2, 56, 0, 0], [4, 32, 0, 0]]	[[2, 2], 2]	[[56, 1, 0, 0], [1, 56, 0, 0], [4, 32, 0, 0]]	1792
PADD_7	[[29, 1, 29, 1], [15, 1, 16, 2, 15, 1], 128]	([1, 1, 1, 1], 2)	[[30, 2, 0, 0], [16, 4, 0, 0], 128]	—
CONV_7	[[30, 2, 0, 0], [16, 4, 0, 0], 128]	[[3, 3, 128, [8, 32, 0, 0]], 1]	[[28, 2, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	256
ACTI_7	[[28, 2, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	[0]	[[28, 2, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	256
QUAN_7	[[28, 2, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	[0]	[[28, 2, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	256
PADD_8	[[29, 1, 29, 1], [8, 1, 9, 6, 8, 1], 256]	([1, 1, 1, 1], 2)	[[30, 2, 0, 0], [9, 8, 0, 0], 256]	—
CONV_8	[[30, 2, 0, 0], [9, 8, 0, 0], 256]	[[3, 3, 256, [4, 64, 0, 0]], 1]	[[28, 2, 0, 0], [7, 8, 0, 0], [4, 64, 0, 0]]	1024
ACTI_8	[[28, 2, 0, 0], [7, 8, 0, 0], [4, 64, 0, 0]]	[0]	[[28, 2, 0, 0], [7, 8, 0, 0], [4, 64, 0, 0]]	1024
QUAN_8	[[28, 2, 0, 0], [7, 8, 0, 0], [4, 64, 0, 0]]	[0]	[[28, 2, 0, 0], [7, 8, 0, 0], [4, 64, 0, 0]]	1024
PADD_9	[[15, 1, 16, 2, 15, 1], [15, 1, 16, 2, 15, 1], 256]	([1, 1, 1, 1], 2)	[[16, 4, 0, 0], [16, 4, 0, 0], 256]	—
CONV_9	[[16, 4, 0, 0], [16, 4, 0, 0], 256]	[[3, 3, 256, [4, 64, 0, 0]], 1]	[[14, 4, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	1024
ACTI_9	[[14, 4, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	[0]	[[14, 4, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	1024
QUAN_9	[[14, 4, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	[0]	[[14, 4, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	1024
POOL_10	[[14, 4, 0, 0], [14, 4, 0, 0], [4, 64, 0, 0]]	[[2, 2], 2]	[[7, 4, 0, 0], [7, 4, 0, 0], [4, 64, 0, 0]]	1024
PADD_11	[[28, 1], [8, 1, 9, 2, 8, 1], 256]	([1, 1, 1, 1], 2)	[[30, 1, 0, 0], [9, 4, 0, 0], 256]	—
CONV_11	[[30, 1, 0, 0], [9, 4, 0, 0], 256]	[[3, 3, 256, [4, 128, 0, 0]], 1]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 128, 0, 0]]	512
ACTI_11	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 128, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 128, 0, 0]]	512
QUAN_11	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 128, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 128, 0, 0]]	512
PADD_12	[[15, 1, 15, 1], [5, 1, 6, 5, 5, 1], 512]	([1, 1, 1, 1], 2)	[[16, 2, 0, 0], [6, 7, 0, 0], 512]	—
CONV_12	[[16, 2, 0, 0], [6, 7, 0, 0], 512]	[[3, 3, 512, [4, 128, 0, 0]], 1]	[[14, 2, 0, 0], [4, 7, 0, 0], [4, 128, 0, 0]]	1792
ACTI_12	[[14, 2, 0, 0], [4, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 2, 0, 0], [4, 7, 0, 0], [4, 128, 0, 0]]	1792
QUAN_12	[[14, 2, 0, 0], [4, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 2, 0, 0], [4, 7, 0, 0], [4, 128, 0, 0]]	1792
PADD_13	[[15, 1, 15, 1], [5, 1, 6, 5, 5, 1], 512]	([1, 1, 1, 1], 2)	[[16, 2, 0, 0], [6, 7, 0, 0], 512]	—
CONV_13	[[16, 2, 0, 0], [6, 7, 0, 0], 512]	[[3, 3, 512, [4, 128, 0, 0]], 1]	[[14, 2, 0, 0], [4, 7, 0, 0], [4, 128, 0, 0]]	1792
ACTI_13	[[14, 2, 0, 0], [4, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 2, 0, 0], [4, 7, 0, 0], [4, 128, 0, 0]]	1792
QUAN_13	[[14, 2, 0, 0], [4, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 2, 0, 0], [4, 7, 0, 0], [4, 128, 0, 0]]	1792
POOL_14	[[14, 2, 0, 0], [4, 7, 0, 0], [4, 128, 0, 0]]	[[2, 2], 2]	[[7, 2, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	1792
PADD_15	[[14, 1], [3, 1, 4, 5, 3, 1], 512]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [4, 7, 0, 0], 512]	—
CONV_15	[[16, 1, 0, 0], [4, 7, 0, 0], 512]	[[3, 3, 512, [4, 128, 0, 0]], 1]	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	896
ACTI_15	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	896
QUAN_15	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	896
PADD_16	[[14, 1], [3, 1, 4, 5, 3, 1], 512]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [4, 7, 0, 0], 512]	—
CONV_16	[[16, 1, 0, 0], [4, 7, 0, 0], 512]	[[3, 3, 512, [4, 128, 0, 0]], 1]	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	896
ACTI_16	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	896
QUAN_16	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	896
PADD_17	[[14, 1], [3, 1, 4, 5, 3, 1], 512]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [4, 7, 0, 0], 512]	—
CONV_17	[[16, 1, 0, 0], [4, 7, 0, 0], 512]	[[3, 3, 512, [4, 128, 0, 0]], 1]	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	896
ACTI_17	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	896
QUAN_17	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	[0]	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	896
POOL_18	[[14, 1, 0, 0], [2, 7, 0, 0], [4, 128, 0, 0]]	[[2, 2], 2]	[[7, 1, 0, 0], [1, 7, 0, 0], [4, 128, 0, 0]]	896
FC_19	[[256, 98, 0, 0], 1]	[[128, 32, 0, 0], [256, 98, 0, 0]]	[[128, 32, 0, 0], 1]	3136
FCE_19	[[128, 32, 0, 0], 98]	[0]	[[128, 32, 0, 0], 1]	3104
ACTI_19	[[128, 32, 0, 0], 1]	[0]	[[128, 32, 0, 0], 1]	32
QUAN_19	[[128, 32, 0, 0], 1]	[0]	[[128, 32, 0, 0], 1]	32
FC_20	[[256, 16, 0, 0], 1]	[[128, 32, 0, 0], [256, 16, 0, 0]]	[[128, 32, 0, 0], 1]	512
FCE_20	[[128, 32, 0, 0], 16]	[0]	[[128, 32, 0, 0], 1]	480
ACTI_20	[[128, 32, 0, 0], 1]	[0]	[[128, 32, 0, 0], 1]	32
QUAN_20	[[128, 32, 0, 0], 1]	[0]	[[128, 32, 0, 0], 1]	32

Operation	Input Feature Map	Weight	Output Feature Map	Pieces
FC_21	[[256, 16, 0, 0], 1]	[[32, 32, 0, 0], [256, 16, 0, 0]]	[[32, 32, 0, 0], 1]	512
FCE_21	[[32, 32, 0, 0], 16]	[0]	[[32, 32, 0, 0], 1]	480
ACTI_21	[[32, 32, 0, 0], 1]	[0]	[[32, 32, 0, 0], 1]	32
QUAN_21	[[32, 32, 0, 0], 1]	[0]	[[32, 32, 0, 0], 1]	32

Table I.1: VGG-16: split scheme for SpiNNaker2 Simulator

Operation	Partial Operation	Pieces	Input Feature Map	Filter	Output Feature Map	MAC Utilization	SRAM Utilization
CONV_1	CONVSUTL_1	256	11520 [10848(94.17%)]	112 [108(96.43%)]	50176 [50176(100.00%)]	1.00	0.62
CONV_2	CONVSUTL_2	1568	55296 [39168(70.83%)]	2304 [2304(100.00%)]	8192 [8192(100.00%)]	1.00	0.51
CONV_4	CONVSUTL_4	512	73728 [65664(89.06%)]	2304 [2304(100.00%)]	12544 [12544(100.00%)]	1.00	0.82
CONV_5	CONVSUTL_5	1792	65536 [58368(89.06%)]	4608 [4608(100.00%)]	3584 [3584(100.00%)]	1.00	0.68
CONV_7	CONVSUTL_7	256	65536 [61440(93.75%)]	9216 [9216(100.00%)]	12544 [12544(100.00%)]	0.88	0.85
CONV_8	CONVSUTL_8	1024	73728 [69120(93.75%)]	9216 [9216(100.00%)]	3136 [3136(100.00%)]	0.88	0.83
CONV_9	CONVSUTL_9	1024	65536 [65536(100.00%)]	9216 [9216(100.00%)]	3584 [3136(87.50%)]	0.88	0.79
CONV_11	CONVSUTL_11	512	73728 [69120(93.75%)]	9216 [9216(100.00%)]	3136 [3136(100.00%)]	0.88	0.83
CONV_12	CONVSUTL_12	1792	49152 [49152(100.00%)]	18432 [18432(100.00%)]	1024 [896(87.50%)]	0.88	0.70
CONV_13	CONVSUTL_13	1792	49152 [49152(100.00%)]	18432 [18432(100.00%)]	1024 [896(87.50%)]	0.88	0.70
CONV_15	CONVSUTL_15	896	32768 [32768(100.00%)]	18432 [18432(100.00%)]	512 [448(87.50%)]	0.88	0.53
CONV_16	CONVSUTL_16	896	32768 [32768(100.00%)]	18432 [18432(100.00%)]	512 [448(87.50%)]	0.88	0.53
CONV_17	CONVSUTL_17	896	32768 [32768(100.00%)]	18432 [18432(100.00%)]	512 [448(87.50%)]	0.88	0.53
FC_19	FCTL_19	3136	1024 [256(25.00%)]	32768 [32768(100.00%)]	2048 [512(25.00%)]	0.25	0.34
FC_20	FCTL_20	512	1024 [256(25.00%)]	32768 [32768(100.00%)]	2048 [512(25.00%)]	0.25	0.34
FC_21	FCTL_21	512	1024 [256(25.00%)]	8192 [8192(100.00%)]	512 [128(25.00%)]	0.25	0.09

Table I.2: VGG-16: SRAM and MAC utilization in SpiNNaker2 Simulator

J ResNet-50 split report for SpiNNaker2 Simulator

Operation	Input Feature Map	Weight	Output Feature Map	Pieces
PADD_1	[[224, 1], [18, 1, 21, 12, 19, 1], 3]	([3, 2, 3, 2], 5)	[[229, 1, 0, 0], [21, 14, 0, 0], 3]	—
CONV_1	[[229, 1, 0, 0], [21, 14, 0, 0], 3]	[[7, 7, 3, [4, 16, 0, 0]], 2]	[[112, 1, 0, 0], [8, 14, 0, 0], [4, 16, 0, 0]]	224
ACTI_1	[[112, 1, 0, 0], [8, 14, 0, 0], [4, 16, 0, 0]]	[0]	[[112, 1, 0, 0], [8, 14, 0, 0], [4, 16, 0, 0]]	224
QUAN_1	[[112, 1, 0, 0], [8, 14, 0, 0], [4, 16, 0, 0]]	[0]	[[112, 1, 0, 0], [8, 14, 0, 0], [4, 16, 0, 0]]	224
PADD_2	[[56, 1, 57, 1], [112, 1], [1, 64, 0, 0]]	([1, 0, 1, 0], 1)	[[57, 2, 0, 0], [113, 1, 0, 0], [1, 64, 0, 0]]	—
POOL_2	[[57, 2, 0, 0], [113, 1, 0, 0], [1, 64, 0, 0]]	[[3, 3], 2]	[[28, 2, 0, 0], [56, 1, 0, 0], [1, 64, 0, 0]]	128
CONV_3	[[56, 1, 0, 0], [7, 8, 0, 0], 64]	[[1, 1, 64, [4, 16, 0, 0]], 1]	[[56, 1, 0, 0], [7, 8, 0, 0], [4, 16, 0, 0]]	128
ACTI_3	[[56, 1, 0, 0], [7, 8, 0, 0], [4, 16, 0, 0]]	[0]	[[56, 1, 0, 0], [7, 8, 0, 0], [4, 16, 0, 0]]	128
QUAN_3	[[56, 1, 0, 0], [7, 8, 0, 0], [4, 16, 0, 0]]	[0]	[[56, 1, 0, 0], [7, 8, 0, 0], [4, 16, 0, 0]]	128
PADD_4	[[29, 1, 29, 1], [15, 1, 16, 2, 15, 1], 64]	([1, 1, 1, 1], 2)	[[30, 2, 0, 0], [16, 4, 0, 0], 64]	—
CONV_4	[[30, 2, 0, 0], [16, 4, 0, 0], 64]	[[3, 3, 64, [4, 16, 0, 0]], 1]	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	128
ACTI_4	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	[0]	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	128
QUAN_4	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	[0]	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	128
CONV_5	[[56, 1, 0, 0], [14, 4, 0, 0], 64]	[[1, 1, 64, [8, 32, 0, 0]], 1]	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	128
CONV_SC_5	[[56, 1, 0, 0], [14, 4, 0, 0], 64]	[[1, 1, 64, [8, 32, 0, 0]], 1]	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	128
MAT_ELE_5	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	128
ACTI_5	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	128
QUAN_5	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	128
CONV_6	[[28, 2, 0, 0], [7, 8, 0, 0], 256]	[[1, 1, 256, [4, 16, 0, 0]], 1]	[[28, 2, 0, 0], [7, 8, 0, 0], [4, 16, 0, 0]]	256
ACTI_6	[[28, 2, 0, 0], [7, 8, 0, 0], [4, 16, 0, 0]]	[0]	[[28, 2, 0, 0], [7, 8, 0, 0], [4, 16, 0, 0]]	256
QUAN_6	[[28, 2, 0, 0], [7, 8, 0, 0], [4, 16, 0, 0]]	[0]	[[28, 2, 0, 0], [7, 8, 0, 0], [4, 16, 0, 0]]	256
PADD_7	[[29, 1, 29, 1], [15, 1, 16, 2, 15, 1], 64]	([1, 1, 1, 1], 2)	[[30, 2, 0, 0], [16, 4, 0, 0], 64]	—
CONV_7	[[30, 2, 0, 0], [16, 4, 0, 0], 64]	[[3, 3, 64, [4, 16, 0, 0]], 1]	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	128
ACTI_7	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	[0]	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	128
QUAN_7	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	[0]	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	128
CONV_8	[[56, 1, 0, 0], [14, 4, 0, 0], 64]	[[1, 1, 64, [8, 32, 0, 0]], 1]	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	128
MAT_ELE_8	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	128
ACTI_8	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	128
QUAN_8	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	128
CONV_9	[[28, 2, 0, 0], [7, 8, 0, 0], 256]	[[1, 1, 256, [4, 16, 0, 0]], 1]	[[28, 2, 0, 0], [7, 8, 0, 0], [4, 16, 0, 0]]	256
ACTI_9	[[28, 2, 0, 0], [7, 8, 0, 0], [4, 16, 0, 0]]	[0]	[[28, 2, 0, 0], [7, 8, 0, 0], [4, 16, 0, 0]]	256
QUAN_9	[[28, 2, 0, 0], [7, 8, 0, 0], [4, 16, 0, 0]]	[0]	[[28, 2, 0, 0], [7, 8, 0, 0], [4, 16, 0, 0]]	256
PADD_10	[[29, 1, 29, 1], [15, 1, 16, 2, 15, 1], 64]	([1, 1, 1, 1], 2)	[[30, 2, 0, 0], [16, 4, 0, 0], 64]	—
CONV_10	[[30, 2, 0, 0], [16, 4, 0, 0], 64]	[[3, 3, 64, [4, 16, 0, 0]], 1]	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	128
ACTI_10	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	[0]	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	128
QUAN_10	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	[0]	[[28, 2, 0, 0], [14, 4, 0, 0], [4, 16, 0, 0]]	128
CONV_11	[[56, 1, 0, 0], [14, 4, 0, 0], 64]	[[1, 1, 64, [8, 32, 0, 0]], 1]	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	128
MAT_ELE_11	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	128
ACTI_11	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	128
QUAN_11	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	[0]	[[56, 1, 0, 0], [14, 4, 0, 0], [8, 32, 0, 0]]	128
CONV_12	[[56, 1, 0, 0], [2, 28, 0, 0], 256]	[[1, 1, 256, [4, 32, 0, 0]], 2]	[[28, 1, 0, 0], [1, 28, 0, 0], [4, 32, 0, 0]]	896
ACTI_12	[[28, 1, 0, 0], [1, 28, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [1, 28, 0, 0], [4, 32, 0, 0]]	896
QUAN_12	[[28, 1, 0, 0], [1, 28, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [1, 28, 0, 0], [4, 32, 0, 0]]	896
PADD_13	[[28, 1], [8, 1, 9, 2, 8, 1], 128]	([1, 1, 1, 1], 2)	[[30, 1, 0, 0], [9, 4, 0, 0], 128]	—
CONV_13	[[30, 1, 0, 0], [9, 4, 0, 0], 128]	[[3, 3, 128, [4, 32, 0, 0]], 1]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	128
ACTI_13	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	128
QUAN_13	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	128
CONV_14	[[28, 1, 0, 0], [7, 4, 0, 0], 128]	[[1, 1, 128, [16, 32, 0, 0]], 1]	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	128
CONV_SC_14	[[56, 1, 0, 0], [2, 28, 0, 0], 256]	[[1, 1, 256, [16, 32, 0, 0]], 2]	[[28, 1, 0, 0], [1, 28, 0, 0], [16, 32, 0, 0]]	896
MAT_ELE_14	[[28, 1, 0, 0], [1, 28, 0, 0], [16, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [1, 28, 0, 0], [16, 32, 0, 0]]	896
ACTI_14	[[28, 1, 0, 0], [1, 28, 0, 0], [16, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [1, 28, 0, 0], [16, 32, 0, 0]]	896
QUAN_14	[[28, 1, 0, 0], [1, 28, 0, 0], [16, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [1, 28, 0, 0], [16, 32, 0, 0]]	896
CONV_15	[[14, 2, 0, 0], [7, 4, 0, 0], 512]	[[1, 1, 512, [4, 32, 0, 0]], 1]	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	256
ACTI_15	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	[0]	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	256
QUAN_15	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	[0]	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	256
PADD_16	[[28, 1], [8, 1, 9, 2, 8, 1], 128]	([1, 1, 1, 1], 2)	[[30, 1, 0, 0], [9, 4, 0, 0], 128]	—
CONV_16	[[30, 1, 0, 0], [9, 4, 0, 0], 128]	[[3, 3, 128, [4, 32, 0, 0]], 1]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	128
ACTI_16	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	128
QUAN_16	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	128
CONV_17	[[28, 1, 0, 0], [7, 4, 0, 0], 128]	[[1, 1, 128, [16, 32, 0, 0]], 1]	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	128
MAT_ELE_17	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	128
ACTI_17	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	128
QUAN_17	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	128

Operation	Input Feature Map	Weight	Output Feature Map	Pieces
CONV_18	[[14, 2, 0, 0], [7, 4, 0, 0], 512]	[[1, 1, 512, [4, 32, 0, 0]], 1]	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	256
ACTI_18	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	[0]	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	256
QUAN_18	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	[0]	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	256
PADD_19	[[28, 1], [8, 1, 9, 2, 8, 1], 128]	([1, 1, 1, 1], 2)	[[30, 1, 0, 0], [9, 4, 0, 0], 128]	—
CONV_19	[[30, 1, 0, 0], [9, 4, 0, 0], 128]	[[3, 3, 128, [4, 32, 0, 0]], 1]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	128
ACTI_19	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	128
QUAN_19	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	128
CONV_20	[[28, 1, 0, 0], [7, 4, 0, 0], 128]	[[1, 1, 128, [16, 32, 0, 0]], 1]	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	128
MAT_ELE_20	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	128
ACTI_20	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	128
QUAN_20	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	128
CONV_21	[[14, 2, 0, 0], [7, 4, 0, 0], 512]	[[1, 1, 512, [4, 32, 0, 0]], 1]	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	256
ACTI_21	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	[0]	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	256
QUAN_21	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	[0]	[[14, 2, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	256
PADD_22	[[28, 1], [8, 1, 9, 2, 8, 1], 128]	([1, 1, 1, 1], 2)	[[30, 1, 0, 0], [9, 4, 0, 0], 128]	—
CONV_22	[[30, 1, 0, 0], [9, 4, 0, 0], 128]	[[3, 3, 128, [4, 32, 0, 0]], 1]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	128
ACTI_22	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	128
QUAN_22	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [4, 32, 0, 0]]	128
CONV_23	[[28, 1, 0, 0], [7, 4, 0, 0], 128]	[[1, 1, 128, [16, 32, 0, 0]], 1]	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	128
MAT_ELE_23	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	128
ACTI_23	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	128
QUAN_23	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	[0]	[[28, 1, 0, 0], [7, 4, 0, 0], [16, 32, 0, 0]]	128
CONV_24	[[28, 1, 0, 0], [2, 14, 0, 0], 512]	[[1, 1, 512, [8, 32, 0, 0]], 2]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
ACTI_24	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
QUAN_24	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
PADD_25	[[14, 1], [8, 1, 8, 1], 256]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [9, 2, 0, 0], 256]	—
CONV_25	[[16, 1, 0, 0], [9, 2, 0, 0], 256]	[[3, 3, 256, [4, 64, 0, 0]], 1]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
ACTI_25	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
QUAN_25	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
CONV_26	[[14, 1, 0, 0], [7, 2, 0, 0], 256]	[[1, 1, 256, [16, 64, 0, 0]], 1]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
CONV_SC_26	[[28, 1, 0, 0], [2, 14, 0, 0], 512]	[[1, 1, 512, [32, 32, 0, 0]], 2]	[[14, 1, 0, 0], [1, 14, 0, 0], [32, 32, 0, 0]]	448
MAT_ELE_26	[[14, 1, 0, 0], [1, 14, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [32, 32, 0, 0]]	448
ACTI_26	[[14, 1, 0, 0], [1, 14, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [32, 32, 0, 0]]	448
QUAN_26	[[14, 1, 0, 0], [1, 14, 0, 0], [32, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [32, 32, 0, 0]]	448
CONV_27	[[14, 1, 0, 0], [1, 14, 0, 0], 1024]	[[1, 1, 1024, [8, 32, 0, 0]], 1]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
ACTI_27	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
QUAN_27	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
PADD_28	[[14, 1], [8, 1, 8, 1], 256]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [9, 2, 0, 0], 256]	—
CONV_28	[[16, 1, 0, 0], [9, 2, 0, 0], 256]	[[3, 3, 256, [4, 64, 0, 0]], 1]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
ACTI_28	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
QUAN_28	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
CONV_29	[[14, 1, 0, 0], [7, 2, 0, 0], 256]	[[1, 1, 256, [16, 64, 0, 0]], 1]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
MAT_ELE_29	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
ACTI_29	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
QUAN_29	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
CONV_30	[[14, 1, 0, 0], [1, 14, 0, 0], 1024]	[[1, 1, 1024, [8, 32, 0, 0]], 1]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
ACTI_30	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
QUAN_30	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
PADD_31	[[14, 1], [8, 1, 8, 1], 256]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [9, 2, 0, 0], 256]	—
CONV_31	[[16, 1, 0, 0], [9, 2, 0, 0], 256]	[[3, 3, 256, [4, 64, 0, 0]], 1]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
ACTI_31	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
QUAN_31	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
CONV_32	[[14, 1, 0, 0], [7, 2, 0, 0], 256]	[[1, 1, 256, [16, 64, 0, 0]], 1]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
MAT_ELE_32	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
ACTI_32	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
QUAN_32	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
CONV_33	[[14, 1, 0, 0], [1, 14, 0, 0], 1024]	[[1, 1, 1024, [8, 32, 0, 0]], 1]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
ACTI_33	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
QUAN_33	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
PADD_34	[[14, 1], [8, 1, 8, 1], 256]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [9, 2, 0, 0], 256]	—
CONV_34	[[16, 1, 0, 0], [9, 2, 0, 0], 256]	[[3, 3, 256, [4, 64, 0, 0]], 1]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
ACTI_34	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
QUAN_34	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
CONV_35	[[14, 1, 0, 0], [7, 2, 0, 0], 256]	[[1, 1, 256, [16, 64, 0, 0]], 1]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
MAT_ELE_35	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
ACTI_35	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
QUAN_35	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
CONV_36	[[14, 1, 0, 0], [1, 14, 0, 0], 1024]	[[1, 1, 1024, [8, 32, 0, 0]], 1]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
ACTI_36	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
QUAN_36	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
PADD_37	[[14, 1], [8, 1, 8, 1], 256]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [9, 2, 0, 0], 256]	—
CONV_37	[[16, 1, 0, 0], [9, 2, 0, 0], 256]	[[3, 3, 256, [4, 64, 0, 0]], 1]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
ACTI_37	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
QUAN_37	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
CONV_38	[[14, 1, 0, 0], [7, 2, 0, 0], 256]	[[1, 1, 256, [16, 64, 0, 0]], 1]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
MAT_ELE_38	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
ACTI_38	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128

Operation	Input Feature Map	Weight	Output Feature Map	Pieces
QUAN_38	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
CONV_39	[[14, 1, 0, 0], [1, 14, 0, 0], 1024]	[[1, 1, 1024, [8, 32, 0, 0]], 1]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
ACTI_39	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
QUAN_39	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	[0]	[[14, 1, 0, 0], [1, 14, 0, 0], [8, 32, 0, 0]]	448
PADD_40	[[14, 1], [8, 1, 8, 1], 256]	([1, 1, 1, 1], 2)	[[16, 1, 0, 0], [9, 2, 0, 0], 256]	—
CONV_40	[[16, 1, 0, 0], [9, 2, 0, 0], 256]	[[3, 3, 256, [4, 64, 0, 0]], 1]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
ACTI_40	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
QUAN_40	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [4, 64, 0, 0]]	128
CONV_41	[[14, 1, 0, 0], [7, 2, 0, 0], 256]	[[1, 1, 256, [16, 64, 0, 0]], 1]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
MAT_ELE_41	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
ACTI_41	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
QUAN_41	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	[0]	[[14, 1, 0, 0], [7, 2, 0, 0], [16, 64, 0, 0]]	128
CONV_42	[[14, 1, 0, 0], [2, 7, 0, 0], 1024]	[[1, 1, 1024, [16, 32, 0, 0]], 2]	[[7, 1, 0, 0], [1, 7, 0, 0], [16, 32, 0, 0]]	224
ACTI_42	[[7, 1, 0, 0], [1, 7, 0, 0], [16, 32, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [16, 32, 0, 0]]	224
QUAN_42	[[7, 1, 0, 0], [1, 7, 0, 0], [16, 32, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [16, 32, 0, 0]]	224
PADD_43	[[7, 1], [7, 1], 512]	([1, 1, 1, 1], 2)	[[9, 1, 0, 0], [9, 1, 0, 0], 512]	—
CONV_43	[[9, 1, 0, 0], [9, 1, 0, 0], 512]	[[3, 3, 512, [4, 128, 0, 0]], 1]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
ACTI_43	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
QUAN_43	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
CONV_44	[[7, 1, 0, 0], [7, 1, 0, 0], 512]	[[1, 1, 512, [16, 128, 0, 0]], 1]	[[7, 1, 0, 0], [7, 1, 0, 0], [16, 128, 0, 0]]	128
CONV_SC_44	[[14, 1, 0, 0], [2, 7, 0, 0], 1024]	[[1, 1, 1024, [16, 128, 0, 0]], 2]	[[7, 1, 0, 0], [1, 7, 0, 0], [16, 128, 0, 0]]	896
MAT_ELE_44	[[7, 1, 0, 0], [1, 7, 0, 0], [16, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [16, 128, 0, 0]]	896
ACTI_44	[[7, 1, 0, 0], [1, 7, 0, 0], [16, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [16, 128, 0, 0]]	896
QUAN_44	[[7, 1, 0, 0], [1, 7, 0, 0], [16, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [16, 128, 0, 0]]	896
CONV_45	[[7, 1, 0, 0], [1, 7, 0, 0], 2048]	[[1, 1, 2048, [8, 64, 0, 0]], 1]	[[7, 1, 0, 0], [1, 7, 0, 0], [8, 64, 0, 0]]	448
ACTI_45	[[7, 1, 0, 0], [1, 7, 0, 0], [8, 64, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [8, 64, 0, 0]]	448
QUAN_45	[[7, 1, 0, 0], [1, 7, 0, 0], [8, 64, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [8, 64, 0, 0]]	448
PADD_46	[[7, 1], [7, 1], 512]	([1, 1, 1, 1], 2)	[[9, 1, 0, 0], [9, 1, 0, 0], 512]	—
CONV_46	[[9, 1, 0, 0], [9, 1, 0, 0], 512]	[[3, 3, 512, [4, 128, 0, 0]], 1]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
ACTI_46	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
QUAN_46	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
CONV_47	[[7, 1, 0, 0], [7, 1, 0, 0], 512]	[[1, 1, 512, [16, 128, 0, 0]], 1]	[[7, 1, 0, 0], [7, 1, 0, 0], [16, 128, 0, 0]]	128
MAT_ELE_47	[[7, 1, 0, 0], [7, 1, 0, 0], [16, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [16, 128, 0, 0]]	128
ACTI_47	[[7, 1, 0, 0], [7, 1, 0, 0], [16, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [16, 128, 0, 0]]	128
QUAN_47	[[7, 1, 0, 0], [7, 1, 0, 0], [16, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [16, 128, 0, 0]]	128
CONV_48	[[7, 1, 0, 0], [1, 7, 0, 0], 2048]	[[1, 1, 2048, [8, 64, 0, 0]], 1]	[[7, 1, 0, 0], [1, 7, 0, 0], [8, 64, 0, 0]]	448
ACTI_48	[[7, 1, 0, 0], [1, 7, 0, 0], [8, 64, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [8, 64, 0, 0]]	448
QUAN_48	[[7, 1, 0, 0], [1, 7, 0, 0], [8, 64, 0, 0]]	[0]	[[7, 1, 0, 0], [1, 7, 0, 0], [8, 64, 0, 0]]	448
PADD_49	[[7, 1], [7, 1], 512]	([1, 1, 1, 1], 2)	[[9, 1, 0, 0], [9, 1, 0, 0], 512]	—
CONV_49	[[9, 1, 0, 0], [9, 1, 0, 0], 512]	[[3, 3, 512, [4, 128, 0, 0]], 1]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
ACTI_49	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
QUAN_49	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [4, 128, 0, 0]]	128
CONV_50	[[7, 1, 0, 0], [7, 1, 0, 0], 512]	[[1, 1, 512, [16, 128, 0, 0]], 1]	[[7, 1, 0, 0], [7, 1, 0, 0], [16, 128, 0, 0]]	128
MAT_ELE_50	[[7, 1, 0, 0], [7, 1, 0, 0], [16, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [16, 128, 0, 0]]	128
ACTI_50	[[7, 1, 0, 0], [7, 1, 0, 0], [16, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [16, 128, 0, 0]]	128
QUAN_50	[[7, 1, 0, 0], [7, 1, 0, 0], [16, 128, 0, 0]]	[0]	[[7, 1, 0, 0], [7, 1, 0, 0], [16, 128, 0, 0]]	128
POOL_51	[[7, 1, 0, 0], [7, 1, 0, 0], [16, 128, 0, 0]]	[[7, 7], 7]	[[1, 1, 0, 0], [1, 1, 0, 0], [16, 128, 0, 0]]	128
FC_52	[[256, 8, 0, 0], 1]	[[32, 32, 0, 0], [256, 8, 0, 0]]	[[32, 32, 0, 0], 1]	256
FCE_52	[[32, 32, 0, 0], 8]	[0]	[[32, 32, 0, 0], 1]	224
ACTI_52	[[32, 32, 0, 0], 1]	[0]	[[32, 32, 0, 0], 1]	32
QUAN_52	[[32, 32, 0, 0], 1]	[0]	[[32, 32, 0, 0], 1]	32

Table J.1: ResNet-50: Splits scheme for SpiNNaker2 Simulator

Operation	Partial Operation	Pieces	Input Feature Map	Filter	Output Feature Map	MAC Utilization	SRAM Utilization
CONV_1	CONVSUTL_1	224	15120 [14427(95.42%)]	592 [588(99.32%)]	53760 [14336(26.67%)]	0.25	0.30
POOL_2	POOLTLF_2	128	6441 [6441(100.00%)]	0 [0(0.00%)]	1568 [1568(100.00%)]	—	0.08
CONV_3	CONVSUTL_3	128	28672 [25088(87.50%)]	256 [256(100.00%)]	6272 [6272(100.00%)]	0.88	0.32
CONV_4	CONVSUTL_4	128	32768 [30720(93.75%)]	2304 [2304(100.00%)]	6272 [6272(100.00%)]	0.88	0.40
CONV_5	CONVSUTL_5	256	57344 [50176(87.50%)]	256 [256(100.00%)]	12544 [12544(100.00%)]	0.88	0.64
CONV_SC_5	CONVSUTL_SC_5	256	57344 [50176(87.50%)]	256 [256(100.00%)]	12544 [12544(100.00%)]	0.88	0.64
CONV_6	CONVSUTL_6	256	57344 [50176(87.50%)]	1024 [1024(100.00%)]	3136 [3136(100.00%)]	0.88	0.55
CONV_7	CONVSUTL_7	128	32768 [30720(93.75%)]	2304 [2304(100.00%)]	6272 [6272(100.00%)]	0.88	0.40
CONV_8	CONVSUTL_8	256	57344 [50176(87.50%)]	256 [256(100.00%)]	12544 [12544(100.00%)]	0.88	0.64
CONV_9	CONVSUTL_9	256	57344 [50176(87.50%)]	1024 [1024(100.00%)]	3136 [3136(100.00%)]	0.88	0.55
CONV_10	CONVSUTL_10	128	32768 [30720(93.75%)]	2304 [2304(100.00%)]	6272 [6272(100.00%)]	0.88	0.40
CONV_11	CONVSUTL_11	256	57344 [50176(87.50%)]	256 [256(100.00%)]	12544 [12544(100.00%)]	0.88	0.64
CONV_12	CONVSUTL_12	896	32768 [28672(87.50%)]	1024 [1024(100.00%)]	896 [448(50.00%)]	0.22	0.31
CONV_13	CONVSUTL_13	128	36864 [34560(93.75%)]	4608 [4608(100.00%)]	3136 [3136(100.00%)]	0.88	0.43
CONV_14	CONVSUTL_14	512	28672 [25088(87.50%)]	512 [512(100.00%)]	3136 [3136(100.00%)]	0.88	0.29
CONV_SC_14	CONVSUTL_SC_14	3584	32768 [28672(87.50%)]	1024 [1024(100.00%)]	896 [448(50.00%)]	0.22	0.31
CONV_15	CONVSUTL_15	256	57344 [50176(87.50%)]	2048 [2048(100.00%)]	1792 [1568(87.50%)]	0.88	0.55
CONV_16	CONVSUTL_16	128	36864 [34560(93.75%)]	4608 [4608(100.00%)]	3136 [3136(100.00%)]	0.88	0.43
CONV_17	CONVSUTL_17	512	28672 [25088(87.50%)]	512 [512(100.00%)]	3136 [3136(100.00%)]	0.88	0.29
CONV_18	CONVSUTL_18	256	57344 [50176(87.50%)]	2048 [2048(100.00%)]	1792 [1568(87.50%)]	0.88	0.55
CONV_19	CONVSUTL_19	128	36864 [34560(93.75%)]	4608 [4608(100.00%)]	3136 [3136(100.00%)]	0.88	0.43
CONV_20	CONVSUTL_20	512	28672 [25088(87.50%)]	512 [512(100.00%)]	3136 [3136(100.00%)]	0.88	0.29
CONV_21	CONVSUTL_21	256	57344 [50176(87.50%)]	2048 [2048(100.00%)]	1792 [1568(87.50%)]	0.88	0.55
CONV_22	CONVSUTL_22	128	36864 [34560(93.75%)]	4608 [4608(100.00%)]	3136 [3136(100.00%)]	0.88	0.43
CONV_23	CONVSUTL_23	512	28672 [25088(87.50%)]	512 [512(100.00%)]	3136 [3136(100.00%)]	0.88	0.29
CONV_24	CONVSUTL_24	896	32768 [28672(87.50%)]	2048 [2048(100.00%)]	448 [224(50.00%)]	0.22	0.31
CONV_25	CONVSU_25	32	65536 [65536(100.00%)]	18432 [18432(100.00%)]	7168 [6272(87.50%)]	0.88	0.92
CONV_26	CONVSU_26	32	57344 [50176(87.50%)]	8192 [8192(100.00%)]	28672 [25088(87.50%)]	0.88	0.85
CONV_SC_26	CONVSUTL_SC_26	3584	32768 [28672(87.50%)]	2048 [2048(100.00%)]	448 [224(50.00%)]	0.22	0.31
CONV_27	CONVSUTL_27	896	16384 [14336(87.50%)]	4096 [4096(100.00%)]	256 [224(87.50%)]	0.88	0.19
CONV_28	CONVSU_28	32	65536 [65536(100.00%)]	18432 [18432(100.00%)]	7168 [6272(87.50%)]	0.88	0.92
CONV_29	CONVSU_29	32	57344 [50176(87.50%)]	8192 [8192(100.00%)]	28672 [25088(87.50%)]	0.88	0.85
CONV_30	CONVSUTL_30	896	16384 [14336(87.50%)]	4096 [4096(100.00%)]	256 [224(87.50%)]	0.88	0.19
CONV_31	CONVSU_31	32	65536 [65536(100.00%)]	18432 [18432(100.00%)]	7168 [6272(87.50%)]	0.88	0.92
CONV_32	CONVSU_32	32	57344 [50176(87.50%)]	8192 [8192(100.00%)]	28672 [25088(87.50%)]	0.88	0.85
CONV_33	CONVSUTL_33	896	16384 [14336(87.50%)]	4096 [4096(100.00%)]	256 [224(87.50%)]	0.88	0.19
CONV_34	CONVSU_34	32	65536 [65536(100.00%)]	18432 [18432(100.00%)]	7168 [6272(87.50%)]	0.88	0.92
CONV_35	CONVSU_35	32	57344 [50176(87.50%)]	8192 [8192(100.00%)]	28672 [25088(87.50%)]	0.88	0.85
CONV_36	CONVSUTL_36	896	16384 [14336(87.50%)]	4096 [4096(100.00%)]	256 [224(87.50%)]	0.88	0.19
CONV_37	CONVSU_37	32	65536 [65536(100.00%)]	18432 [18432(100.00%)]	7168 [6272(87.50%)]	0.88	0.92
CONV_38	CONVSU_38	32	57344 [50176(87.50%)]	8192 [8192(100.00%)]	28672 [25088(87.50%)]	0.88	0.85
CONV_39	CONVSUTL_39	896	16384 [14336(87.50%)]	4096 [4096(100.00%)]	256 [224(87.50%)]	0.88	0.19
CONV_40	CONVSU_40	32	65536 [65536(100.00%)]	18432 [18432(100.00%)]	7168 [6272(87.50%)]	0.88	0.92
CONV_41	CONVSU_41	32	57344 [50176(87.50%)]	8192 [8192(100.00%)]	28672 [25088(87.50%)]	0.88	0.85
CONV_42	CONVSUTL_42	896	32768 [28672(87.50%)]	4096 [4096(100.00%)]	256 [112(43.75%)]	0.11	0.33
CONV_43	CONVSU_43	128	73728 [41472(56.25%)]	18432 [18432(100.00%)]	896 [784(87.50%)]	0.44	0.62
CONV_44	CONVSU_44	64	57344 [25088(43.75%)]	16384 [16384(100.00%)]	7168 [6272(87.50%)]	0.44	0.49
CONV_SC_44	CONVSUTL_SC_44	3584	32768 [28672(87.50%)]	4096 [4096(100.00%)]	256 [112(43.75%)]	0.11	0.33
CONV_45	CONVSUTL_45	896	32768 [14336(43.75%)]	8192 [8192(100.00%)]	128 [112(87.50%)]	0.44	0.23
CONV_46	CONVSU_46	128	73728 [41472(56.25%)]	18432 [18432(100.00%)]	896 [784(87.50%)]	0.44	0.62
CONV_47	CONVSU_47	64	57344 [25088(43.75%)]	16384 [16384(100.00%)]	7168 [6272(87.50%)]	0.44	0.49
CONV_48	CONVSUTL_48	896	32768 [14336(43.75%)]	8192 [8192(100.00%)]	128 [112(87.50%)]	0.44	0.23
CONV_49	CONVSU_49	128	73728 [41472(56.25%)]	18432 [18432(100.00%)]	896 [784(87.50%)]	0.44	0.62
CONV_50	CONVSU_50	64	57344 [25088(43.75%)]	16384 [16384(100.00%)]	7168 [6272(87.50%)]	0.44	0.49
FC_52	FCTL_52	256	1024 [256(25.00%)]	8192 [8192(100.00%)]	512 [128(25.00%)]	0.25	0.09

Table J.2: ResNet-50: SRAM and MAC utilization in SpiNNaker2 Simulator. MLA could not accelerate the pooling operation, therefore the MAC unit utilization is not available.

References

[1] Sebastian Hoeppner and Christian Mayr. *SpiNNaker2 Towards extremely efficient digital neuromorphics and multi-scale brain emulation*. 2018 (cit. on pp. iii, 1, 11).

[2] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015. URL: http://arxiv.org/abs/1409.1556 (cit. on pp. iii, 1, 3, 9).

[3] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90 (cit. on pp. iii, 1, 3, 9, 93).

[4] V. Sze, Y. Chen, T. Yang, and J. S. Emer. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329. ISSN: 0018-9219. DOI: 10.1109/JPROC.2017.2761740 (cit. on pp. 3, 4, 6–8).

[5] Dominik Marek Loroch, Franz-Josef Pfreundt, Norbert Wehn, and Janis Keuper. "TensorQuant: A Simulation Toolbox for Deep Neural Network Quantization". In: *Proceedings of the Machine Learning on HPC Environments*. MLHPC'17. Denver, CO, USA: ACM, 2017, 1:1–1:8. ISBN: 978-1-4503-5137-9. DOI: 10.1145/3146347.3146348. URL: http://doi.acm.org/10.1145/3146347.3146348 (cit. on pp. 4, 9).

[6] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks". In: *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 127–138. ISSN: 0018-9200. DOI: 10.1109/JSSC.2016.2616357 (cit. on p. 5).

[7] A. Maas, A. Hannun, and A. Ng. "Rectifier Nonlinearities improve Neural Network Acoustic Models". In: *Proc. 30th Int. Conf. on Machine Learning (ICML)*. 2013 (cit. on p. 6).

[8] TU Dresden. *SpiNNaker2 Wiki: SpiNNaker2 Universal Spiking Neural Network Architecture* (cit. on pp. 11, 14, 15).

[9] Python software foundation. *Python/C API Reference Manual* (cit. on p. 21).

[10] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L Ceze, C. Guestrin, and A. Krishnamurthy. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: (2018). URL: https://arxiv.org/abs/1802.04799 (cit. on p. 23).

[11] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *arXiv e-prints*, arXiv:1603.04467 (2016). arXiv: 1603.04467 [cs.DC] (cit. on p. 46).

[12] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: http://doi.acm.org/10.1145/1498765.1498785 (cit. on p. 52).