UC Davis
UC Davis Electronic Theses and Dissertations

Title
An Energy-Efficient SqueezeNet Implementation on the KiloCore Platform

Permalink
https://escholarship.org/uc/item/13t253c6

Author
Dong, Ziyuan

Publication Date
2022

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

An Energy-Efficient SqueezeNet Implementation on the KiloCore Platform

By

ZIYUAN DONG
THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

____
Bevan M. Baas, Chair

____
Soheil Ghiasi

____
Houman Homayoun

Committee in Charge

2022

© Copyright by Ziyuan Dong 2022
All Rights Reserved

Abstract

Many Convolutional Neural Networks (CNNs) have been developed for object detection, image classification, and facial recognition applications. Although many deep convolutional neural networks have focused on improving accuracy, few have focused on reducing the number of required hardware resources. While reducing hardware requirements is expected to reduce throughput performance, these simpler architectures are expected to provide advantages such as lower latency, lower power, and smaller memory requirements. In addition, simpler CNNs can be implemented on more devices, and are in general easier to train because they contain fewer parameters which required to be trained. This thesis proposes a KiloCore implementation of SqueezeNet, a lightweight CNN that offers low energy and high throughput, and contains 1,248,424 parameters inside 22 layers composed of 18 convolutional layers and 4 pooling layers.

This thesis presents an implementation of SqueezeNet running on a fine-grain many-core processor array called KiloCore. The metrics to be compared include energy per frame, power, throughput, throughput per area, energy-delay product (EDP), and memory. We compare with: SqueezeNet implementations running on an Intel Xeon E3-1275 v5 @ 3.6 GHz, an Intel i5-5250U @ 2.7 GHz, an Intel Knights Landing @ 1.7 GHz, a Qualcomm Snapdragon 810 @ 1.5 GHz, an NVIDIA Pascal @ 3.0 GHz, and an ARMv71 @ 0.9 GHz.

The KiloCore many-core implementation achieves a $1.0\times - 17.0\times$ lower energy per frame and $3.1\times - 35.3\times$ lower power dissipation. Regarding throughput performance, the KiloCore implementation is $4.8\times$ higher than ARMv71 processor. The EDP value for KiloCore implementation is in the middle range among other hardware platform implementations, and the EDP is $95.2\times$ lower compared to an ARMv71 processor. SqueezeNet implementation on KiloCore has significantly fewer memory requirements than other programmable processors.

Acknowledgments

I would like to thank those who have provided invaluable assistance throughout my academic year at UC Davis.

Professor Bevan Baas advised me through all the stages of my graduate studies. His timely advice informed advice and scientific approach have helped me to a very great extent to accomplish me to complete my graduate studies. I worked as a teaching assistant for Professor Baas in many classes. During my work as a teaching assistant with Professor Baas, the most valuable thing I learned is how to take care of students and how to think thoroughly.

Also, I want to thank you Professor Soheil Ghiasi and Professor Houman Homayoun for your time and efforts in reviewing my thesis.

I am glad to have the chance to join the VCL lab. Thank you, Shifu and Brent, for helping me with the Kilocore hardware. Shifu helped me with peer review this thesis which is very helpful. Satya helped me with my interview preparation and gave me much valuable advice. Thank you Professor Aaron Stillmaker for helping me with the scaling calculation. Thank you Ryan for working as a research assistant with me on this thesis.

It is my pleasure to meet with Haotian, Yikai, and Weitai. They supported me a lot in Davis.

I would like to thank my family for supporting my graduate studies. They gave me mental and emotional support.

Contents

Abstract **ii**

Acknowledgments **iii**

List of Figures **vi**

List of Tables **viii**

1 Introduction **1**
1.1 Motivation 1
1.2 Thesis Organization 2

2 Background of SqueezeNet **4**
2.1 Overview 4
2.2 SqueezeNet 4
2.2.1 Relevant SqueezeNet Definitions 4
2.2.2 SqueezeNet Overview 5
2.2.3 Architectural Design Strategies 5
2.2.4 Padding 7
2.2.5 Stride 8
2.2.6 Rectified Linear Units (ReLU) 8
2.2.7 Convolutional Layer 8
2.2.8 Fire Module 9
2.2.9 Pooling Layer 11
2.2.10 Softmax Layer 13
2.3 Related Work 13
2.3.1 Model Deep Compression 13

3 Background of KiloCore Platform **14**
3.1 Overview 15
3.2 Processors 15
3.3 On-Chip SRAM 15
3.4 Software for Writing Many-Core Applications 16

4 Implementations of Components of SqueezeNet on KiloCore **17**
4.1 Overview 17
4.1.1 Overview of Design Strategies 17
4.1.2 Overview of Image Direction 18

4.2 Maxpool Layers . . . 18
4.2.1 Chip Level of Maxpool Layer . . . 19
4.2.2 Core Level of Input Channel Distributor Stage . . . 19
4.2.3 Core Level of Maxpool Computation Stage . . . 22
4.2.4 Core Level of Output Buffer Stage . . . 22
4.2.5 Maxpool Layer Mapping Result . . . 24
4.3 Average Pooling Layers . . . 24
4.4 Convolutional Layers . . . 25
4.4.1 Squeeze: 1×1 convolutional layers . . . 25
4.4.2 Expand: 1×1 and 3×3 convolutional layers . . . 38
4.4.3 Layer 1: 7×7 convolutional layer . . . 42
4.4.4 Summary . . . 44

5 Inference Evaluation 48
5.1 Inference Evaluation . . . 48
5.1.1 Overview . . . 48
5.1.2 Inference Result . . . 48
5.1.3 Simulation Measurements . . . 49

6 SqueezeNet Performance Comparison Between Different Hardware Platforms 57
6.1 Overview . . . 57
6.2 Comparison of SqueezeNet KiloCore Implementation with Other Platforms . . . 57
6.2.1 Throughput Performance . . . 60
6.2.2 Energy Dissipation . . . 60
6.2.3 Power Dissipation . . . 61
6.2.4 Energy $\times$ Delay . . . 61
6.2.5 Memory Requirements . . . 63
6.3 Summary . . . 63

7 Thesis Summary and Future Work 65
7.1 Thesis Summary . . . 65
7.2 Future Work . . . 65
7.2.1 Increase the Number of SRAMs . . . 65
7.2.2 More Input Ports per Core . . . 66
7.2.3 Saturating Addition . . . 66
7.2.4 Reduce the Number of Layers in SqueezeNet . . . 66

Bibliography 69

List of Figures

2.1 Zero-padding with padding size equal to one 7
2.2 Example of the convolutional layer. The image size is 5×5 and the filter size is 3×3 with the stride size equal to one. The output data size is 3×3. 9
2.3 Organization of convolution filters of fire module. The fire module is built up by two convolutional layers which are the squeeze layer and the expand layer. The squeeze layer is on the top of the figure and the expand layer is at the bottom of the figure. Each layer is followed by an ReLU layer. 10
2.4 The squeeze layer of the fire module. The input image is multiplied with weights first, then the bias is added to the result. In the end, the ReLU function is applied to generate the final value. 10
2.5 The Expand layer of the fire Module [1]. The input data size is $55 \times 55 \times 16$ and the output data size is $55 \times 55 \times 128$. 11
2.6 Example of the 3×3 maxpool Layer with stride of two 12
2.7 Example of the Max 3×3 Average Pooling Layer with stride of two 13

3.1 Die photo of the KiloCore array, and annotated layout plots of a single processor tile and a single independent memory tile [2,3]. 14
3.2 Project Manager GUI [2]. 16

4.1 Image direction in X-Y-Z direction. X corresponding to width; Y corresponding to Height; Z corresponding to the depth of image 18
4.2 The block diagram of the maxpool1 layer. The data flow is from left to right. 19
4.3 The maxpool distributor stage. One SRAM and six cores are shown in figure. . . . 20
4.4 The 3 stages output buffer of the maxpool layer. The final output is in order from A to H. 23
4.5 The mapping diagram of the maxpool1 layer. The black line represents neighbor connection, and blue line represents hop-distance-of-2 connection. The green line represents long-distance connections 24
4.6 The block diagram example of the KiloCore when processing the fire squeeze layer 7 or processing the fire squeeze layer 8 27
4.7 The block diagram of squeeze layer stage 1 — Example of the fire squeeze layer 7 or layer 8. The input_0 and input_1 are the input data from the previous SqueezeNet layer. 28
4.8 Kernel distributor block diagram — An example of the fire squeeze layer 7 or fire squeeze layer 8. Each of the cores in sub-stage 1 is connected with an SRAM. The same structure is repeated 3 more times for 384 channels. 30
4.9 The block diagram of the weight and bias distributor stage of the fire squeeze layer . 33

4.10 The core diagram of the weight and bias combination stage 33
4.11 The block diagram of the parallel computing distributor 35
4.12 The core diagram of the convolutional layer 35
4.13 The block diagram of the adder stage, which adds eight numbers together. 37
4.14 The squeeze layer architecture mapping to the KiloCore processor array with eight SRAMs. 39
4.15 The squeeze layer architecture mapping to the KiloCore processor array with 12 SRAMs. 39
4.16 The squeeze layer architecture mapping to the KiloCore processor array with 13 SRAMs. 40
4.17 The chip-level diagram of the expand Layer which is built up by applying both 1×1 convolutional layer (upper) and 3×3 convolutional layer (lower) to the same input data. 41
4.18 The expand layer architecture mapping to the KiloCore processor array with 12 SRAMs. 43
4.19 The chip-level diagram of the 7×7 convolutional layer. This is the layer 1 of the SqueezeNet. 44
4.20 The 7×7 convolutional layer architecture mapping to the KiloCore processor array with six SRAMs. 45
4.21 The top 9 layers of the SqueezeNet. Each set of blue and red boxes represents one KiloCore chip. 46
4.22 The bottom 13 layers of the SqueezeNet. 12 KiloCore chips in total because last two layers are combined. 47

5.1 The traffic light image being used to simulate the SqueezeNet 49
5.2 The mean and max error plot of each layer in SqueezeNet. The blue line is the max value and the orange line shows the mean value. 51
5.3 Maximum operating frequency of processors, memories, and routers [4]. The frequency of 1.24 GHz at 1.1 V condition is used for measurement. 53
5.4 Energy per typical operation for processors, memories, and routers [4]. 54
5.5 Power of a processor, memory, and router when 100% active and operating at the maximum clock frequency at the indicated supply voltage [4]. 56

6.1 Energy comparison between different hardware platforms based on Equation 6.5 and Table 6.5. 61
6.2 Power comparison between different hardware platforms by Table 6.5 and Equation 5.7. 62
6.3 Memory comparison between different hardware platforms based on Table 6.5 63

List of Tables

2.1 SqueezeNet architectural dimensions of each layer [1]. 6

5.1 The mean and max error with the relative error of each layer in SqueezeNet calculated by Equation 5.1, 5.2, 5.3, 5.4. 50
5.2 KiloCore performance data when processing each frame 1.24 GHz @ 900 mV by using Equation 5.5, 5.6, 5.7, 5.8, 5.9. 53
5.3 Detailed active and total energy and output time measurements of each layer of 1.24 GHz at 900 mV when processing one frame. 55
5.4 KiloCore performance data when processing each frame 1.74 GHz @ 1100 mV by using Equation 5.8, 5.9, and Figure 5.4, 5.5. 55

6.1 The polynomial coefficient values and delay factors calculated with Equation 6.1 [5]. 58
6.2 The polynomial coefficient values and energy factors calculated with Equation 6.2 [5]. 58
6.3 Factors used for area scaling [5]. 59
6.4 Unscaled raw data for different programmable processors. 59
6.5 A comparison of key parameters for a variety of programmable hardware platforms all scaled from original data shown in Table 6.4 to 32 nm CMOS technology. Numbers in this table are calculated using Equations 6.3, 6.4, 6.5, and 6.6. 59
6.6 Normalized energy per image. The smallest number is normalized to one based on Table 6.5. 60

Chapter 1

Introduction

1.1 Motivation

As computer hardware evolves, deep learning has become more popular and has more use cases such as natural language processing, automatic driving, and computer vision. Convolutional neural network (CNN) is one of the most popular deep learning algorithms for image-related tasks.

The AlexNet is one of the most influential neural networks in the computer vision area, which has 60 million parameters and consists of five convolutional layers [6]. It won the ILSVRC-2012 competition and achieved a winning top-5 error rate of 15.3%. The new idea proposed in the paper includes max pooling, dropout, and rectified linear activation unit (ReLU) functions. The details of each function are introduced in chapter 2. The ResNet is another convolutional neural network that won the ILSVR-2015 competition. ResNet addresses the degradation problem by introducing a deep residual learning framework. Instead of expecting each stacked layer to fit a desired underlying mapping directly, ResNet explicitly lets these layers fit a residual mapping [7].

The previous two CNNs have shown the benefits of efficiently detecting object features; however, all of them are large CNN architecture and require large hardware resources.

At the same level of accuracy, there are many advantages of a smaller CNN architecture with fewer parameters:

- More efficient distributed training
- Less overhead when exporting new models to users
- Embedded deployment and on-chip storage

For the first point, the training speed of the small model is faster due to requiring less communication. The training speed is directly proportional to the number of parameters in the model [8]. The model has some benefits by reducing the long training time, such as fitting more devices and lowering the training cost in both time and hardware.

For the second point, in some circumstances, such as autonomous driving, companies such as Tesla periodically copy their new trained models from their cloud servers to customers' cars. This updating process is often referred to as an over-the-air update. However, over-the-air updates of the typical CNN/DNN models can require large data transfers. For example, with AlexNet, this would require 240 MB of communication from the server to the car. Smaller models require less communication, making frequent updates easier [1].

For the third point, the smaller size model is easier to fit on more devices and requires fewer hardware resources. For some of the ASIC chips, the CNNs can be stored directly on the chip and the smaller model allows more chips to run the model.

This thesis implemented the SqueezeNet on the KiloCore many-core platform that can offer low complexity with the same accuracy as AlexNet.

1.2 Thesis Organization

The remainder of this thesis is organized as follows.

- Chapter 2 outlines the evolutionary history of the SqueezeNet architecture. The basic CNN concept and the component utilized to construct SqueezeNet are briefly discussed.
- Chapter 3 introduces the KiloCore many-core platform architecture, including the chip implementation, processor architecture, on-chip memory hierarchy, and programming software.
- Chapter 4 discusses the implementations of SqueezeNet with details of the algorithms, including convolution, pooling, and other layers.
- Chapter 5 illustrates the simulation results of the SqueezeNet implementation on the KiloCore platform.
- Chapter 6 compares the power and efficiency comparisons among KiloCore implementation, general-purpose processors, GPUs, and FPGAs.

- Chapter 7 summarizes the thesis and the ideas for future work.

Chapter 2

Background of SqueezeNet

2.1 Overview

Convolutional neural network (CNN) is one of the deep learning algorithms that can extract usable information from multi-dimensional input matrices. CNN has been widely used in many scientific fields such as speech recognition [9], search engines [10], and biometric authentication [11], especially in the field of image classification, because CNN avoids the complex pre-processing of the image and can directly input the original image.

Researchers from DeepScale, the University of California, Berkeley, and Stanford University developed SqueezeNet in 2016 [1]. SqueezeNet is designed to have a minimal architecture with the same degree of precision as AlexNet [6]. One of the common misunderstandings regarding SqueezeNet is that it has the same architecture as AlexNet, despite the fact that their architectures are fundamentally different. This chapter examines the SqueezeNet architectural design strategies, including the Fire module, convolutional layers, and maxpool layers.

2.2 SqueezeNet

2.2.1 Relevant SqueezeNet Definitions

To aid in the introduction of the SqueezeNet special terms, we define several new terms:

Definition: Squeeze The *squeeze* layer is a single convolutional layer that contains 1×1 filters.

Definition: Expand The *expand* layer is a single convolutional layer that contains a mix of 1×1 and 3×3 filters.

Definition: Fire The *fire* module is a building block for SqueezeNet. The *fire* module is comprised of a squeeze convolutional layer, which has only 1×1 filters, feeding into an expand layer that has a mix of 1×1 and 3×3 convolution filters.

2.2.2 SqueezeNet Overview

SqueezeNet is a 22-layers CNN, which has 18 convolutional layers and 4 pooling layers. The network is designed to target the embedded device. The small memory problem is approached by taking an existing CNN model and compressing it in a lossy fashion. The key characteristics of about SqueezeNet are in Table 2.1. The network has an image input size of 227×227. The *conv1*, *conv10*, and convolutional layers within the nine fire modules total eighteen convolutional layers. The sixth column $s1 \times 1$ contains the Squeeze layer output depth. The seventh and eighth columns stand for the Expand layer output image depth. The details of each layer and the design strategy is explained in the later section of this chapter.

2.2.3 Architectural Design Strategies

Strategy 1. Replace 3×3 filters with 1×1 filters

Given a budgeted number of convolution filters, the majority of these filters were chosen to be 1×1, since a 1×1 filter has $9\times$ fewer parameters than a 3×3 filter. When running on the chip, the 1×1 filter requires fewer accumulators to do the computation. The computation time is reduced by 9 times when compared with the 3×3 filter.

Strategy 2. Decrease the number of input channels to 3×3 filters

Consider a convolutional layer that is comprised of only 3×3 filters. The total quantity of parameters in this layer is (number of input channels) $\times$ (number of filters) $\times$ (3×3). Therefore, to maintain a small total number of parameters in a CNN, it is important to not only decrease the number of 3×3 filters, but also to decrease the number of input channels.

Strategy 3. Downsample late in the network so that convolutional layers have large activation maps

In the common cases, downsampling is used by setting the stride larger than one in the convolution or pooling layers. If the early layers in the network have large strides, some of the layers will have small activation maps. In converse, if more stride of one are used, the CNN will have large

Layer	**Output size**	**Filter size/ stride**	**Conv depth**	**Maxpool depth**	**s1×1**	**e1×1**	**e3×3**	**Parameter**
input	224×224×3							
conv1	111×111×96	7×7/2 (×96)	1					14,208
maxpool1	55×55×96	3×3/2		1				
fire2	55×55×128		2		16	64	64	11,920
fire3	55×55×128		2		16	64	64	12,432
fire4	55×55×256		2		32	128	128	45,344
maxpool4	27×27×256	3×3/2		1				
fire5	27×27×256		2		32	128	128	49,440
fire6	27×27×384		2		48	192	192	104,880
fire7	27×27×384		2		48	192	192	111,024
fire8	27×27×512		2		64	256	256	188,992
maxpool8	13×13×512	3×3/2		1				
fire9	13×13×512		2		64	256	256	197,184
conv10	13×13×1000	1×1/1 (×1000)	1					513,000
avgpool10	1×1×1000	13×13/1		1				
sum			18	4				1,248,424

Table 2.1: SqueezeNet architectural dimensions of each layer [1].

activation maps. Therefore the large activation maps due to later downsampling is used can cause the higher classification accuracy.

In summary, strategies 1 and 2 are about decreasing the parameter in SqueezeNet while preserving the accuracy. Strategy 3 maximizes the accuracy with the limited amount of parameter. All three strategies are combined to make the SqueezeNet efficient and accurate.

2.2.4 Padding

One of the drawbacks of the convolution step is the loss of information that on the border of the image. Padding can expand the dimension of the output feature map so that it will not shrink with the stacking of convolutional layers. In SqueezeNet, the padding is only applied inside the fire module and it is a process of adding layers of zeros to the input image. Figure 2.1 shows the zero padding to a channel with padding size equal to one.

$$Out.X = In.X + PaddingSize.X \times 2 \tag{2.1}$$

$$Out.Y = In.Y + PaddingSize.Y \times 2 \tag{2.2}$$

$$Out.Z = In.Z \tag{2.3}$$

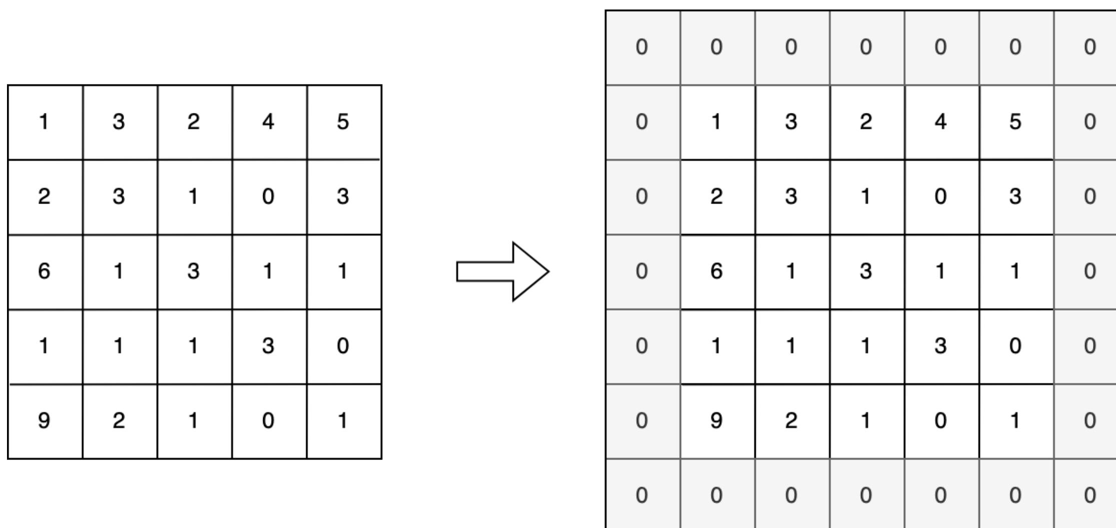


Figure 2.1: Zero-padding with padding size equal to one

2.2.5 Stride

The stride is a parameter of the filter movement over the image. For example, if the stride is set to two the filter moves two pixel at a time. The default stride value is one. The large stride will reduce the amount of overlapping and produce an output of lower spatial dimensions. In SqueezeNet, all of the stripes are either one or two.

2.2.6 Rectified Linear Units (ReLU)

ReLU is the rectified linear activation function which is a common operation applied per input activation value. It changes the negative values in the feature map to zero. The positive number will be outputted directly. It follows the calculation in Equation 2.4.

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

2.2.7 Convolutional Layer

The convolutional layer plays a significant role in how CNNs operate. The layers parameters focus around the use of learnable kernels. The kernels are usually small in spatial dimensinality but spread across the entirety of the depth of the input. When the data hit a convolutional layer, the layer convolves each filter across the spatial dimensional of the input to compute a 2D activation map [12]. Figure 2.2 shows and example of the convolution process. The first element 16 in the output activation map is calculated by dot product between every element of the filter. The activation map is calculated by repeating the process for every element of the input image.

The output size of the convolutional layer is calculated as follows. The convolutional layer has an input feature map with a dimension of $In.X \times In.Y \times In.Z$ and filter window dimension of $Filter.X \times Filter.Y \times Filter.Z$. The number of the filter kernel is K. $FilterStride.X$, $FilterStride.Y$ represent the pace stride of the convolution window. The dimension of the output activation map is:

$$Out.X = \frac{In.X - Filter.X}{FilterStride.X} + 1 \tag{2.5}$$

$$Out.Y = \frac{In.Y - Filter.Y}{FilterStride.Y} + 1 \tag{2.6}$$

$$Out.Z = K \tag{2.7}$$

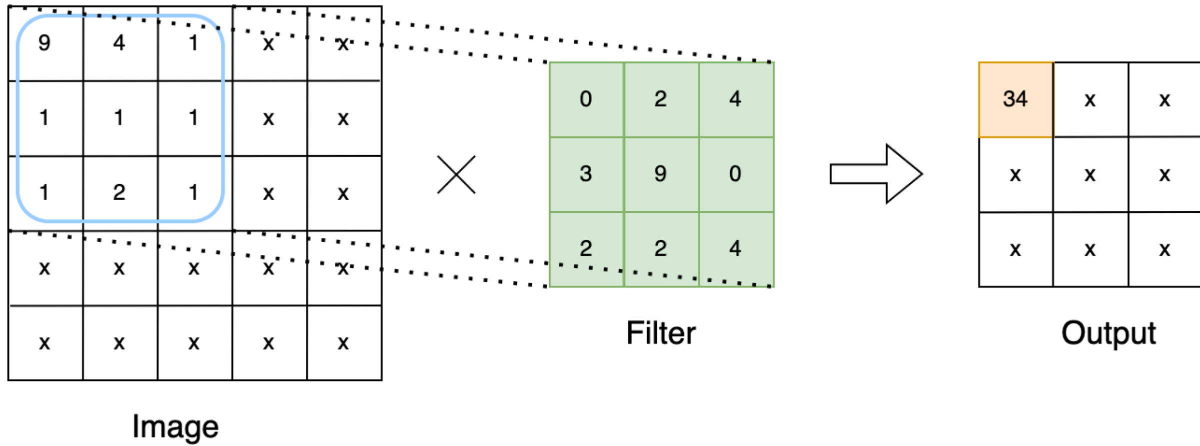


Figure 2.2: Example of the convolutional layer. The image size is 5×5 and the filter size is 3×3 with the stride size equal to one. The output data size is 3×3.

2.2.8 Fire Module

A fire module is built up by two layers of the convolutional layer with both followed by one ReLU layer as shown in Figure 2.3. The name of the first layer is the squeeze layer and the second layer is expand layer. Figure 2.4 shows the algorithm of the squeeze convolutional layer of fire 2 as an example. The input image dimension is $55 \times 55 \times 96$ and the output image dimension is $55 \times 55 \times 16$. The filter size is 1×1 and the stride value is one in squeeze layer, so the width and height of the output image keep at the same value as the input image.

The output of the squeeze convolutional layer feeds into an expand layer. The expand layer is a convolutional layer that has a mix of 1×1 and 3×3 filters. which is shown in Figure 2.5 which used the fire 2 squeeze as an example in dimension. The 64 output channels from 1×1 convolutional filter is combined with 3×3 convolution filter to form a 128 channel output result. The 1×1 convolution output result is places at channel 1-64 and the 3×3 convolution output result is places at channel 65-128 in the output image.

Three tunable dimensions in a fire module are $s_{1\times1}$, $e_{1\times1}$ and $e_{3\times3}$. In the fire module, $s_{1\times1}$ is the number of 1×1 filters in squeeze layer. It corresponds to the strategy 1. The small number of parameters used here makes the SqueezeNet more efficient. The $e_{1\times1}$ and $e_{3\times3}$ are the number of 1×1 and 3×3 in the expand layers as shown in Figure 2.5. The value of $s_{1\times1}$ is less than $e_{1\times1} + e_{3\times3}$, so the squeeze layer helps to limit the number of input channels to the 3×3 filters as shown in strategy 2 [1].

The relationship between $s_{1\times1}$, $e_{1\times1}$ and $e_{3\times3}$ is shown in Equation 2.8. In Figure 2.3,

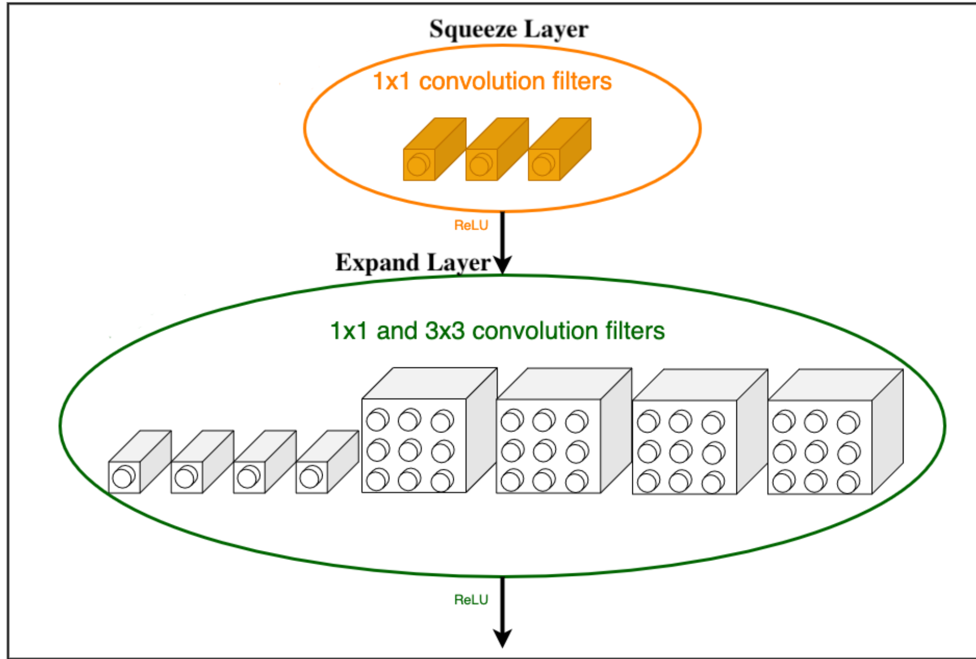


Figure 2.3: Organization of convolution filters of fire module. The fire module is built up by two convolutional layers which are the squeeze layer and the expand layer. The squeeze layer is on the top of the figure and the expand layer is at the bottom of the figure. Each layer is followed by an ReLU layer.

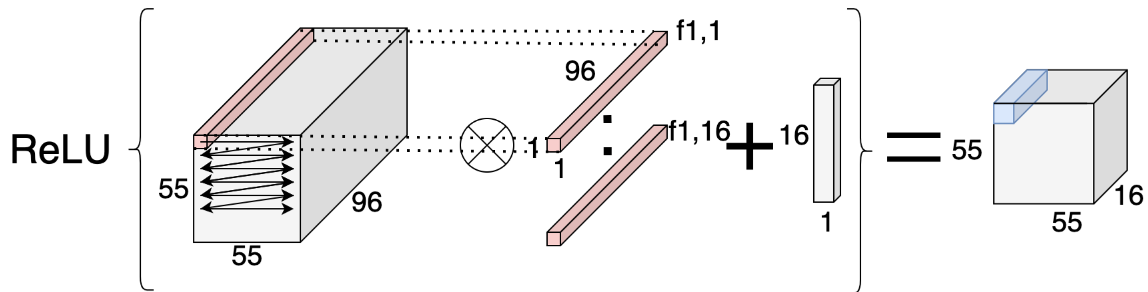


Figure 2.4: The squeeze layer of the fire module. The input image is multiplied with weights first, then the bias is added to the result. In the end, the ReLU function is applied to generate the final value.

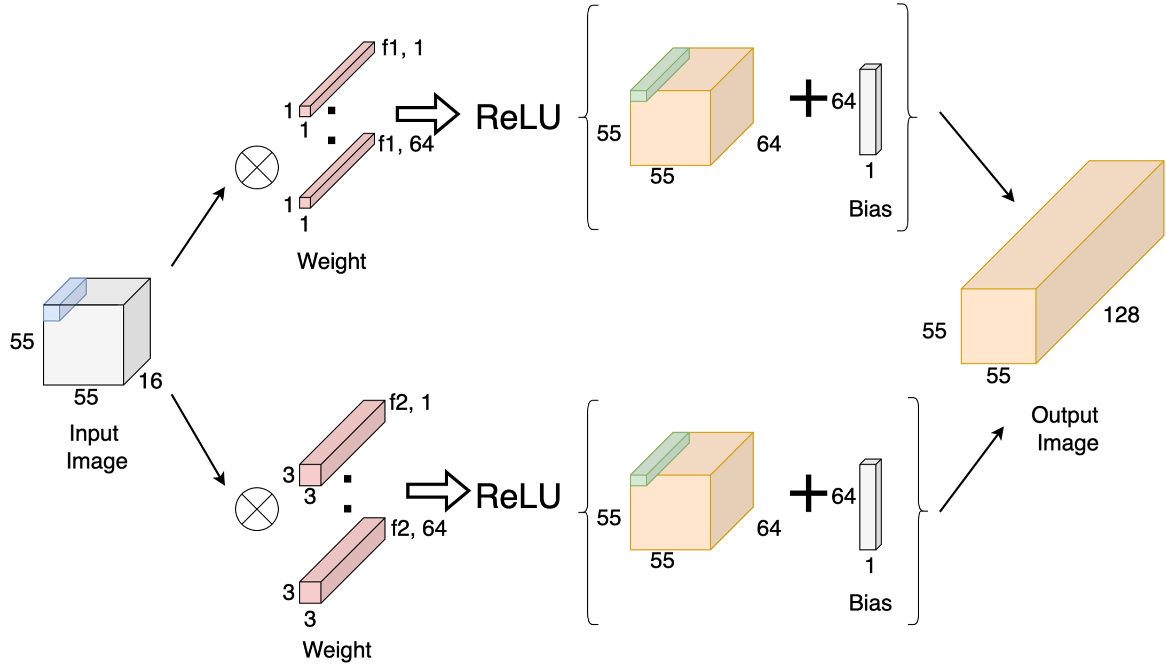


Figure 2.5: The Expand layer of the fire Module [1]. The input data size is $55 \times 55 \times 16$ and the output data size is $55 \times 55 \times 128$.

$s_{1\times1} = 3$, $e_{1\times1} = e_{3\times3} = 4$.

$$s_{1\times1} = \frac{e_{1\times1}}{4} = \frac{e_{3\times3}}{4} \tag{2.8}$$

2.2.9 Pooling Layer

In SqueezeNet, there are two kinds of pooling layers which are maxpool and average pooling layers. The main idea of pooling layer is downsampling to reduce the complexity of subsequent layers. In the field of image processing, it can be considered similar to reducing the resolution without affecting the number of channels [13]. The benefit of the pooling layer is that it reduces the memory requirement for the next layer. Meanwhile, it also results in the disadvantages of reduction in precision and accuracy, because some details are lost after the pooling layer.

The pooling layer has an input feature map with a dimension of $In.X \times In.Y \times In.Z$ and pooling window dimension of $Pooling.X \times Pooling.Y \times Pooling.Z$. $PoolingStride.X, PoolingStride.Y$ represents the pace stride of the pooling window. The dimension of the output feature map is

calculated by:

$$Out.X = \frac{In.X - Pooling.X}{PoolingStride.X} + 1 \tag{2.9}$$

$$Out.Y = \frac{In.Y - Pooling.Y}{PoolingStride.Y} + 1 \tag{2.10}$$

$$Out.Z = In.Z \tag{2.11}$$

MaxPool Layer

MaxPool layer outputs the maximum value from the area of the image covered by the pooling window. There are three layers of the maxpool in SqueezeNet and all of the filter sizes are 3 ×3 with stride equal to two as shown in Figure 2.6. The X and Y dimensions are reduced by about half after each maxpool layer when pooling stride value is equal to two. The accurate output dimension is calculated by Equation 2.9, 2.10, and 2.11.

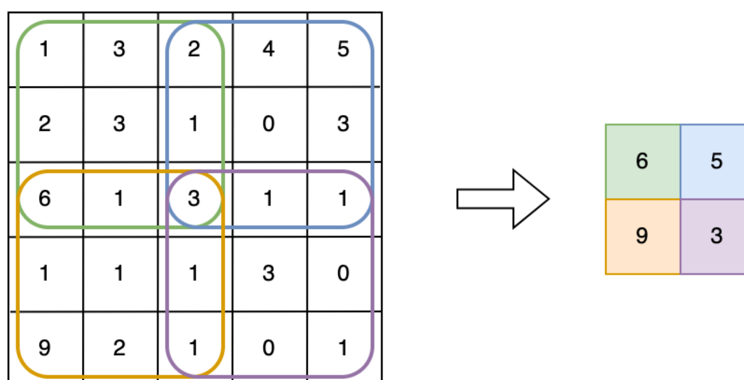


Figure 2.6: Example of the 3 × 3 maxpool Layer with stride of two

Average Pooling Layer

Average pooling is a pooling operation that calculates the average value in each patch of each feature map. Average Pooling output the average value from the area of the image covered by the average pooling window. The second last of the layer of the SqueezeNet is the average pooling which takes all of the number from one 13 × 13 channel to compute the average value. Figure 2.7 below shows how does the average pooling works, when the filter is 3 × 3 and the stride is two.

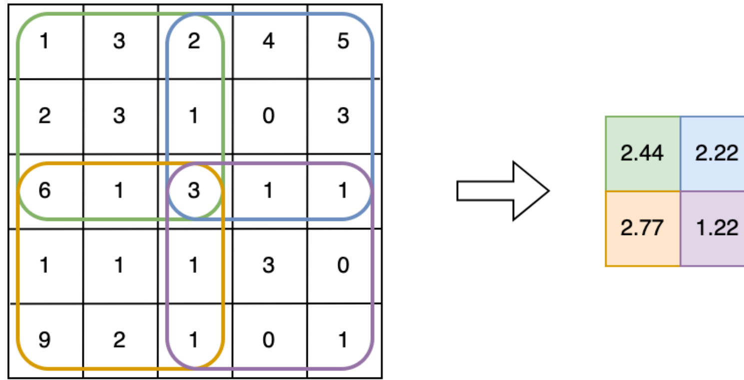


Figure 2.7: Example of the Max 3 × 3 Average Pooling Layer with stride of two

2.2.10 Softmax Layer

The softmax layer is a significant part of neural network that specified for classification scenario. The softmax layer transforms the previous layer's output into the vector of probabilities. It is used in the last layer to convert the result of the neural networks to a probability distribution over K of predicted output categories. In SqueezeNet, the $K = 1,000$. The softmax regression can be formulated as Equation 2.12 [14].

$$\sigma(\mathbf{z}) = \frac{e^{z_i}}{\sum_{i=1}^{K} e^{z_i}} \tag{2.12}$$

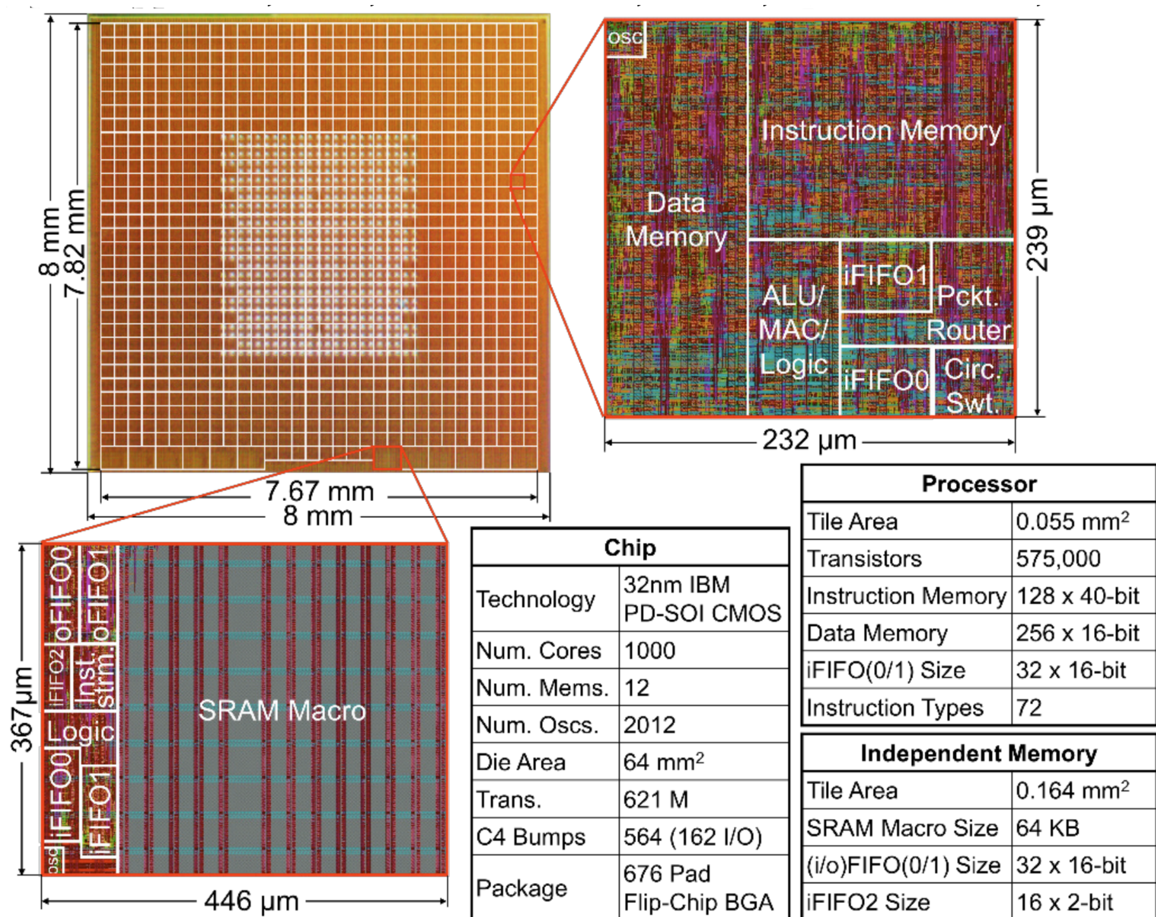
2.3 Related Work

2.3.1 Model Deep Compression

The primary objective of SqueezeNet is to identify a model with fewer parameters while preserving accuracy. The previous research led by Song Han [15] suggested the possibility of lowering the size of SqueezeNet to 0.47Mb, which could be achieved by applying the three stage pipeline: pruning, trained quantization and Huffman coding, which work together to reduce the storage requirement of neural networks by 35× to 49× without compromising accuracy. This method first prunes the network by learning only the important connections. Next, the weights are quantized to enforce weight sharing. Finally, Huffman coding is applied [15].

Chapter 3

Background of KiloCore Platform



Chip	
Technology	32nm IBM PD-SOI CMOS
Num. Cores	1000
Num. Mems.	12
Num. Oscs.	2012
Die Area	64 mm^2
Trans.	621 M
C4 Bumps	564 (162 I/O)
Package	676 Pad Flip-Chip BGA

Processor	
Tile Area	0.055 mm^2
Transistors	575,000
Instruction Memory	128 x 40-bit
Data Memory	256 x 16-bit
iFIFO(0/1) Size	32 x 16-bit
Instruction Types	72

Independent Memory	
Tile Area	0.164 mm^2
SRAM Macro Size	64 KB
(i/o)FIFO(0/1) Size	32 x 16-bit
iFIFO2 Size	16 x 2-bit

Figure 3.1: Die photo of the KiloCore array, and annotated layout plots of a single processor tile and a single independent memory tile [2, 3].

3.1 Overview

KiloCore is the 3rd generation of the Asynchronous Array of simple Processors many-core platform [16–18]. Compared with the previous generations [19–21], KiloCore comes with various enhancements such as a circuit switch network, packet switch routers for better long-distance communication, a new oscillator design, and various core architecture/ISA improvements. The fabricated chip consists of 697 efficient, programmable processors to run software programs, 697 packet routers paired with a processor, and 14 memory modules containing 64 KB of memory each that may be used for data or instructions. Each element has an independent oscillator for local clock generation and communicates with asynchronous neighbors using dual-clock FIFOs [2, 4, 22–25].

3.2 Processors

The bulk of KiloCore is made up of small, programmable processors. These processors execute user-supplied assembly instructions and are interconnected using a mixture of statically configured circuit links, dynamic packet links, and dynamic circuit links. A processor is primarily made up of a central datapath that executes user code, instruction+data memories, and a series of control modules that regulate processor operation and supply special support functionality.

Each processor, as shown in the right upper corner of Figure 3.1 contains 128 × 40-bit instruction memory, 512 Bytes of data memory, three programmable data address generators, two 32 × 16-bit input buffers, and a 16-bit fixed-point datapath with a 32-bit multiplier output and an accumulator. The 72 instruction types include signed and unsigned operations to enable efficient scaling to 32-bit or larger word widths, with no instructions being algorithm-specific [2, 26].

3.3 On-Chip SRAM

14 of 64 KB SRAM memories are placed at the bottom of the KiloCore chip, which offers 896 KB total off-core memory on-die, as shown in left down corner Figure 3.1. The memories can also be used to run larger programs on the neighboring core, acting as an extended instruction memory for that core. With the help of the very-small-area packet router, data can be supplied to all 697 programmable cores, which helps reduce spaces inside the processors and leave more area for the computing components [2].

3.4 Software for Writing Many-Core Applications

The programming environment developed for KiloCore is called Project Manager, which provides the software packages for writing task-based applications, mapping the proposed architecture on the chip, launching simulations to verify its results, and gathering measurements. Programs are written in either Python or C++, with the open-sourced Clang compiler front end for optimizing the assembly code [2].

KiloCore supports RISC-type assembly instructions formatted as `"Opcode`, `Destination`, `Source1`, `Source2`, and `Options"`. `STALL` and `NOP` instructions are inserted by the compiler to avoid pipeline hazards, including Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW). Based on application profiling done by the compiler, branch prediction paths are determined at compile-time and included in the branch Opcode [2] [27].

The Project Manager also has a GUI, which provides an accessible way for users to run scripts, view task layouts and mappings, run the integrated tools, and view simulation results. Figure 3.2 shows the Project Manager GUI after a simulation of a 650-processor version of the FFT application on KiloCore [28, 29]. Simulation metrics are recorded for each individual processor, along with a global summary.

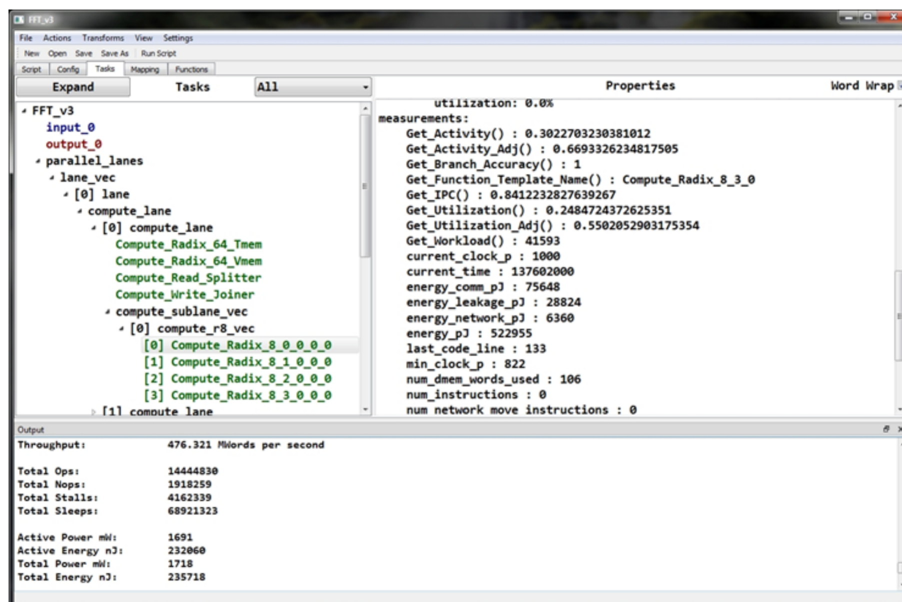


Figure 3.2: Project Manager GUI [2].

Chapter 4

Implementations of Components of SqueezeNet on KiloCore

4.1 Overview

Chapter 4 presents the details of the algorithms, including convolution and pooling layers with different filter sizes. This chapter forces on the chip-level and core-level design of the KiloCore platform.

4.1.1 Overview of Design Strategies

In each SqueezeNet layer, parallel computing addresses the bottleneck of computational latency. There are 697 user-programmable cores and 14 64KB shared SRAMs on the KiloCore chip. Each core performs a similar amount of computation tasks to achieve high performance. If the computing task requires more time to complete, the task would be distributed to other cores and arranged the result after each computation is completed. Tasks are divided into small chunks because it saves the waiting time and lowers the idle power. Other cores do not need to remain idle to complete the tasks.

The other design strategy is to avoid using too many cores in each layer. If all cores are used, the close stages of the data flow may be separated by a long distance during the routing phase. To achieve high efficiency, the performance is higher when using less than 80% of the 697 cores. From the perspective of input and output port numbers, it is ideal to assign most of the cores less

or equal to five input and output ports; In this way, the core can be placed closer, and there is no long time delays between two cores. In this work, all of the core has either one input port with multiple output ports or one output port with multiple input ports to avoid long-distance routing.

4.1.2 Overview of Image Direction

The direction of the 3D image is saved as width-height-depth order (X-Y-Z) format as shown in Figure 4.1. For example, the size of the original input image is 227 pixels high and 227 pixels wide. The depth or channel of the input image is three, which represents red, green, and blue colors.

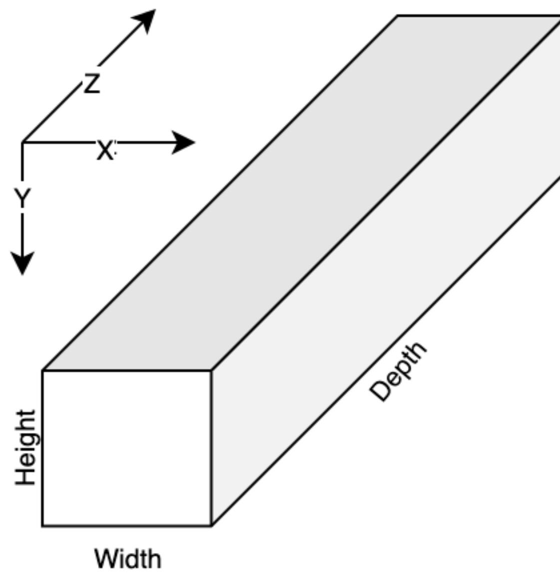


Figure 4.1: Image direction in X-Y-Z direction. X corresponding to width; Y corresponding to Height; Z corresponding to the depth of image

4.2 Maxpool Layers

There are three maxpool layers in the SqueezeNet. All three layers have the same architecture with some minor modifications to fit different sizes of the input image. The chip-level and core-level details are presented in this section. The maxpool1 is used as an example in this section because it has the largest Feature map, which uses the largest portion of the SRAMs.

4.2.1 Chip Level of Maxpool Layer

All three maxpool layers in SqueezeNet are implemented on the KiloCore in the same way as shown in Figure 4.2. The chip-level architecture consists of three groups of identical structures corresponding to three rows in Figure 4.2. Each group of structures has three main stages which are the input distributor stage (blue), compute maxpool stage (green) and output buffer stage (purple). For maxpool1, the input data size is $111 \times 111 \times 96$ and input data is from the previous layer's output. The whole input data is divided into three pieces for reading into KiloCore as shown in the left side of the figure to achieve the high bandwidth. The first piece includes channels 1–32 and the second piece includes the channels 33–64. The third piece contains the channels 65–96. The data distribution stage is responsible for reading the data from the previous layer and the data is distributed and rearranged for the next stage. Compute stage reads the data to calculate the max value. The output buffer stage changes the data from parallel data format to the sequential format. In the end, the output data is outputted via three output ports on the right of Figure 4.2 in the X-Y-Z direction.

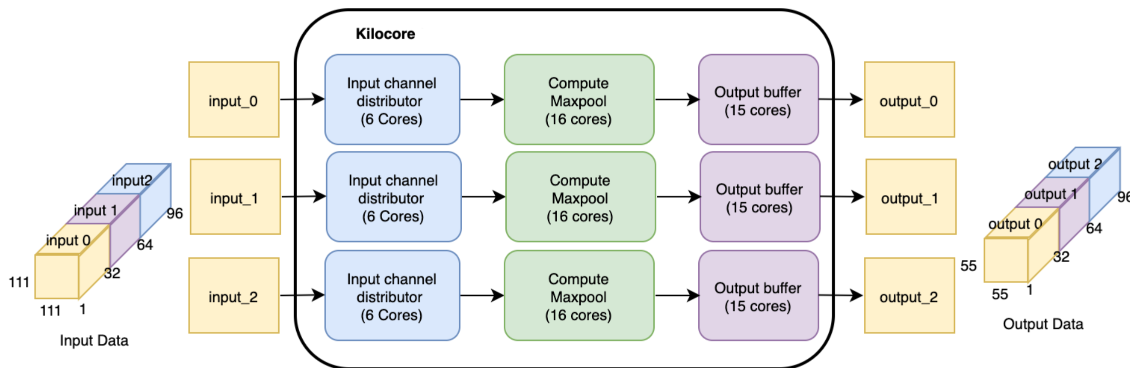


Figure 4.2: The block diagram of the maxpool1 layer. The data flow is from left to right.

4.2.2 Core Level of Input Channel Distributor Stage

The input channel distributor is the first stage in the maxpool layer which used $3 \times 6 = 18$ cores and three SRAMs in total. Figure 4.3 shows one third of the input channel distributor stage, and this figure supposes to be repeated three times in the KiloCore chip. The data flow is from left to right in figure. The Distribute[0] core first gathers one channel at a time from the input port0, then each channel is saved to the on-chip SRAM to be reorganized which is a red rectangular

in Figure 4.3. The data is outputted in the kernel order from the output port0 of Distribute[0] core. Four of the loops are used to calculated the index position as shown in Algorithm 1. The index value is calculated based on stride size, channel size and the previous kernel position.

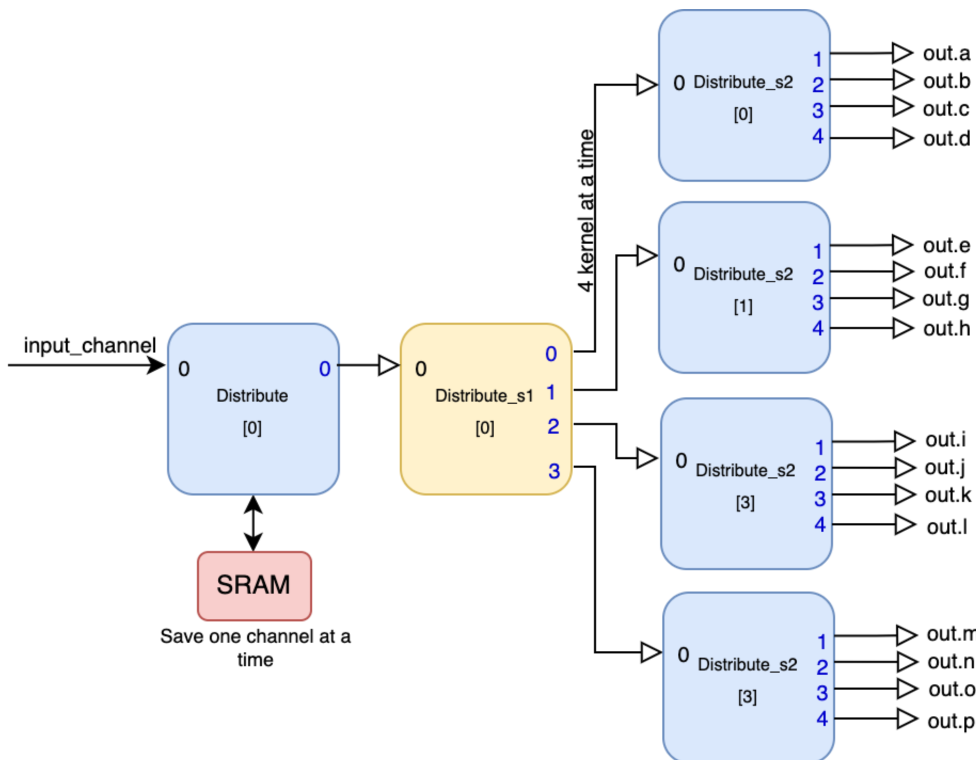


Figure 4.3: The maxpool distributor stage. One SRAM and six cores are shown in figure.

The Distribute_s1 and Distribute_s2 cores are designed to distribute the kernel to enable the parallel computing. Each group of processing cores distribute each kernel to 16 of maxpool calculation cores. This makes $16\ cores/group \times 3\ group = 48\ cores$ parallel kernel processed at the same time be possible. The Distribute_s1 core receives four of 3×3 kernels at one time and repeats four times for four different output port because there are four output ports in the Distribute_s2 core. The Algorithm 2 presents the details of the code.

The distribute_s2 core inputs one of 3×3 kernel at a time and repeats the process four times for four output ports. The code is similar to the Algorithm 2, and the only difference is the loop repeat four times less than Algorithm 2. The distribute_s2 distributes the data directly into the compute stage via output port0.

Algorithm 1 MaxPool Distributer with Three SRAMs

```
1: loop(i < channel_width)
2:     loop(i < channel_height)
3:         SRAM ← Input_0                              ▷ Read one channel from port 0
4:     end loop
5: end loop
6: loop(y < (channel_width − 1)/stride)                ▷ Shift kernel towards bottom side
7:     loop(x < (channel_height − 1)/stride)           ▷ Shift kernel towards right side
8:         loop(j < Kernel_width)                      ▷ Change the row in the kernel
9:             loop(i < Kernel_height)                 ▷ Shift one pixel to the right
10:                SRAM[i + j × channel_height + x × stride + y × channel_width × stride] →
   Output_0
11:                        ▷ The kernel position from left to right and top to bottom
12:            end loop
13:        end loop
14:    end loop
15: end loop
```

Algorithm 2 MaxPool Distributor Stage

```
1: loop(i < kernel_size × kernel_size × port_number)
2:     Output_0 ← Input_0            ▷ Pass data from input port 0 to output port 0
3: end loop
4: loop(i < kernel_size × kernel_size × port_number)
5:     Output_1 ← Input_0            ▷ Pass data from input port 0 to output port 1
6: end loop
7: loop(i < kernel_size × kernel_size × port_number)
8:     Output_2 ← Input_0            ▷ Pass data from input port 0 to output port 2
9: end loop
10: loop(i < kernel_size × kernel_size × port_number)
11:    Output_3 ← Input_0            ▷ Pass data from input port 0 to output port 3
12: end loop
```

4.2.3 Core Level of Maxpool Computation Stage

The computation stage of the maxpool layer includes $16 cores/group \times 3 group = 48 cores$ in total. Each computing core receives nine numbers from port 0 and outputs the maximum of the nine numbers to output port 0. Output port 0 is connected to the next stage, which is the output buffer stage. The output max variable is first initialized to zero, which is the smallest possible value due to the ReLU layer in the previous layer. Every input value is compared with the max value by an if statement. If the next input value is larger than the max variable, the max value is assigned to the new input value. Otherwise, the max variable remains unchanged as shown in Algorithm 3.

Algorithm 3 MaxPool Computation Stage

1: **loop**($i < kernel_size$)
2:     **loop**($i < kernel_size$)
3:         $Input_0 \rightarrow temp$ ▷ Assign the new input value to temp variable
4:         **if** $temp > max$ **then**
5:             $max \leftarrow temp$
6:         **end if**
7:     **end loop**
8: **end loop**
9: $Output_0 \leftarrow max$

4.2.4 Core Level of Output Buffer Stage

The output buffer stage of the maxpool layer uses a total of $15 \times 3 = 45$ cores. The tree structure is designed to achieve the maximum parallel output feature, as shown in Figure 4.4. Each core has two input ports, and the number of required cores for the output buffer tree structure is shown in Equation 4.1. The reason is that each core has two input ports, so each output stage would cut the total number of cores in half. The final stage would have one core to arrange the output order. The code below uses stage 2 as an example to present how the buffer stage works. Firstly, the buffer_2 core reads two computed values from buffer_1 port 0 and sends data to output port 0. Secondly, the core transfer two data from input port 1 to output port 0 as shown in Algorithm 4.

$$Number_cores = \sum_{n=1}^{n-1} n \tag{4.1}$$

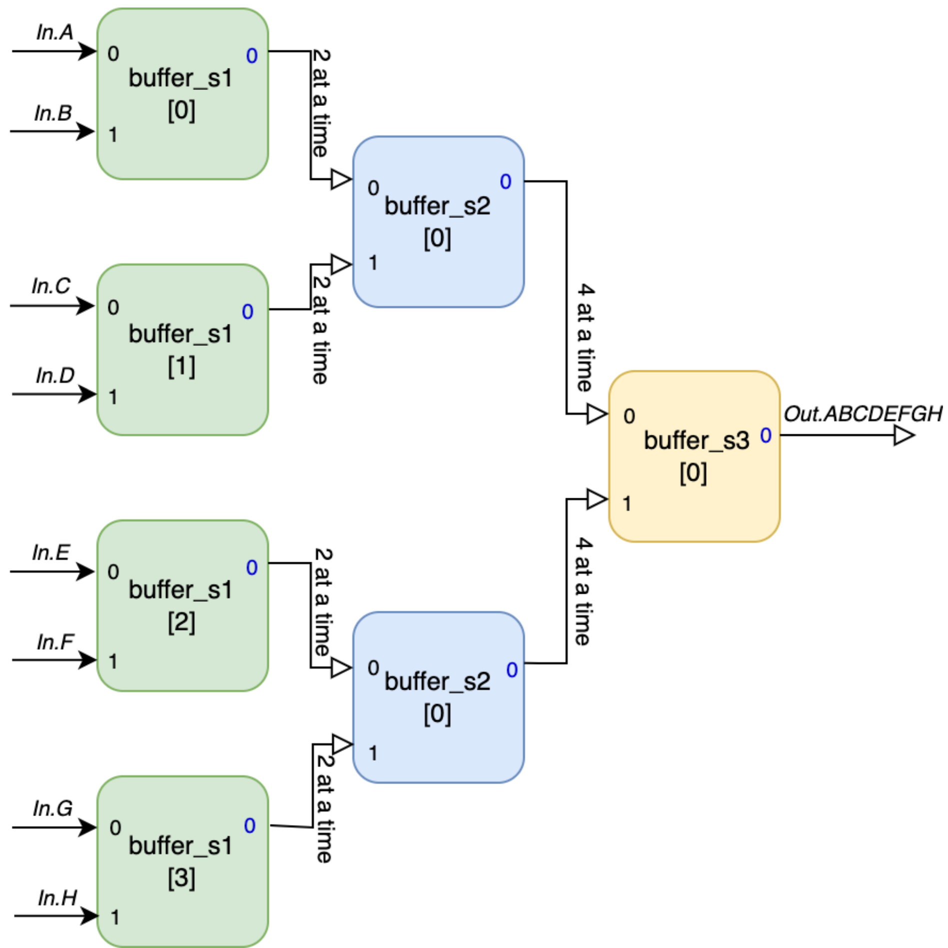


Figure 4.4: The 3 stages output buffer of the maxpool layer. The final output is in order from A to H.

Algorithm 4 MaxPool Output Buffer Stage

1: **loop**$(i < 2^(stage_value - 1))$

2: $\quad Output_0 \leftarrow Input_0$ ▷ Pass data from input port 0 to output port 0

3: **end loop**

4: **loop**$(i < 2^(stage_value - 1))$

5: $\quad Output_0 \leftarrow Input_1$ ▷ Pass data from input port 1 to output port 0

6: **end loop**

4.2.5 Maxpool Layer Mapping Result

The mapping diagram of all three maxpool layers has the same routing structure on the KiloCore because the core number is the same and the block diagram is the same. The only difference is that the input data has different dimensions, so the input channel distributor stage distributes the data has slightly different internal logic according to the data dimension. Figure 4.5 shows the mapping diagram of the maxpool1. The mapping diagram is produced by the mapper of the Project Manager. The black line represents neighbor connection, and blue line represents hop-distance-of-2 connection. The green line represents long-distance connections and the red line represents routing congested links. The congested links do not appear in the maxpool inference. The cores of the three input groups are mapped to different portions of the KiloCore as shown in the lower left, lower middle, and lower right of Figure 4.5. There are no congested links to limit the data processing speed.

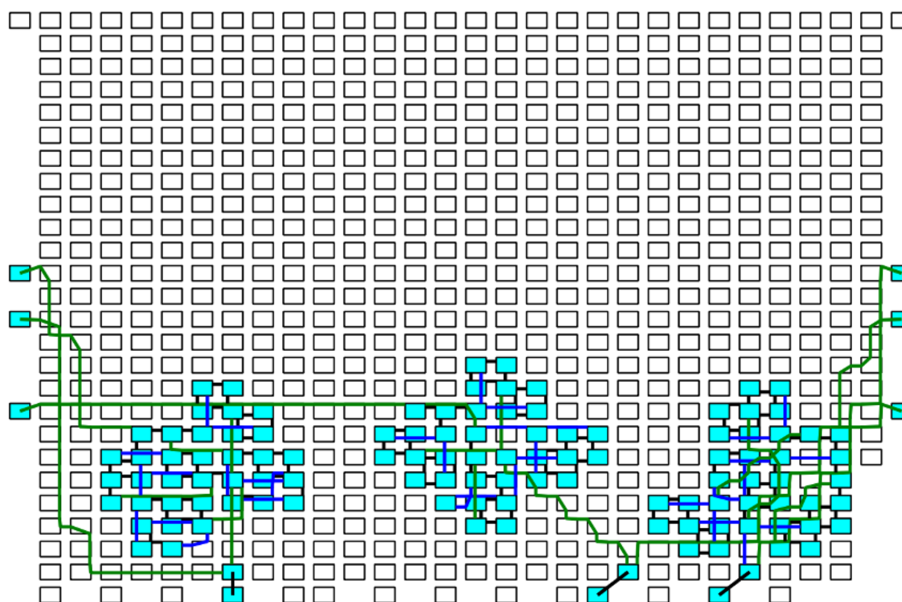


Figure 4.5: The mapping diagram of the maxpool1 layer. The black line represents neighbor connection, and blue line represents hop-distance-of-2 connection. The green line represents long-distance connections

4.3 Average Pooling Layers

There is only one average pooling layer in SqueezeNet which is the last layer. The average pooling layer is combined with the conv10 layer to save one KiloCore chip area. The average pooling

core is responsible for producing the average value of each channel. The channel input size is 13×13 and the output size is 1×1 number for the core to the process. The accumulator is set to 32-bits width to prevent overflow and a while loop is used to summarize all of the inputs. The average value is calculated after the loop. The output value is outputted to port 0.

Algorithm 5 Global Average Pooling

1: **loop**$(i < PoolWidth \times PoolHeight)$ ▷ Read 13×13 input in and sum the value

2: $temp \leftarrow Input[DataCache_GAP]$ ▷ Read input and store in temp variable

3: $sum += temp$ ▷ Accumulate input to sum

4: **end loop**

▷ Calculate the average value of a channel then output the value to port 0

5: $ave \leftarrow sum/PoolWidth/PoolHeight$

6: $Output_0 \leftarrow ave$

4.4 Convolutional Layers

There are three cases of the convolutional layers in the SqueezeNet. Each type of the convolutional layer is customized based on the input data size and filter size so the KiloCore provides the best performance and fits on KiloCore to meet the design strategy.

- 1×1 convolutional layers: Fire Squeeze Layer
- 1×1 with 3×3 convolutional layers: Fire Expand Layer
- 7×7 convolutional layer: Layer 1

The three cases of the convolutional layers is discussed separately to present the details of each case. The 1×1 convolutional layer is the fire squeeze layer. The 1×1 with 3×3 convolutional layer is the fire expand layer. The 7×7 convolutional layer is the first convolutional layer in the SqueezeNet and it only occurs one time.

4.4.1 Squeeze: 1×1 convolutional layers

The high-level block diagram of the 1×1 convolutional layers of the squeeze layer 7 or layer 8 of the fire module is shown in Figure 4.6. Other layers are slightly different in stage 1 and

stage 2, and the difference is explained in the subsection when discussing each stage. The difference is due to the width, height, and channel number of the input data being different for different layers. Layer 7 or layer 8 are used as an example here because they are the most representative layer. Those two layers have the same input data size so these two layers are the same. The input_0 and input_1 cores are both input image from the previous layer output. The data is either from the previous fire expand layer or the maxpool layer based on the position of the squeeze layer.

Stage 1 is the input channel distributor, which distributes the data into 12 SRAMs. Stage 2 used 12 SRAMs and 12 cores to store the entire data in order to distribute each kernel. The SRAMs also allows the image to be used multiple times for different weight and bias calculation. Stage 2 used some extra cores to distribute the output to 16 ports which make each 1×1 convolutional layer has a consistent structure from stage 3. Stage 3 used 16 cores to combine the kernel and the weight. Each core receives data from the weight distributor and the kernel distributor. The output of stage 3 output kernel and weight one at a time for faster processing in the convolution computing stage. Stage 4 has two internal stages, which are combined by 16 cores and 32 cores to enable parallel computing. Stage 5 does the convolution calculation and outputs the partial convolution result to stage 6. Stage 6 is combined by both adder and the output buffer to summarize the convolution result and buffer the parallel output. The last stage adds the bias to the output and do the ReLU calculation, which is done by one core and the output is saved to output_0.

Squeeze Layer Stage 1: Input Channel Distributor

The input channel distributor stage is designed to divide the raw image data into even pieces based on the channel number. The divided data can be processed easily for the next stage. The KiloCore needs 8, 12, or 13 SRAMs based on different input data sizes. For the large input data such as fire 3 squeeze layer and fire 4 squeeze layer, 13 on-chip SRAMs are required to save the entire data on the KiloCore Chip. This way of implementation offers higher bandwidth for larger input data, so the processing speed is not decreased when the data is larger. For the squeeze layer 7/8, both layers have 384 channels in total. The channel number is the ratio of 12, so it is best for these two layers to use 12 over 13 SRAMs. Each 32 (384/12) channel dataset is divided into groups and sent to the next stage. The block diagram of the 12 SRAMs implementation is shown in Figure 4.7. For the 8 or 13 SRAMs implementation, the structure is similar.

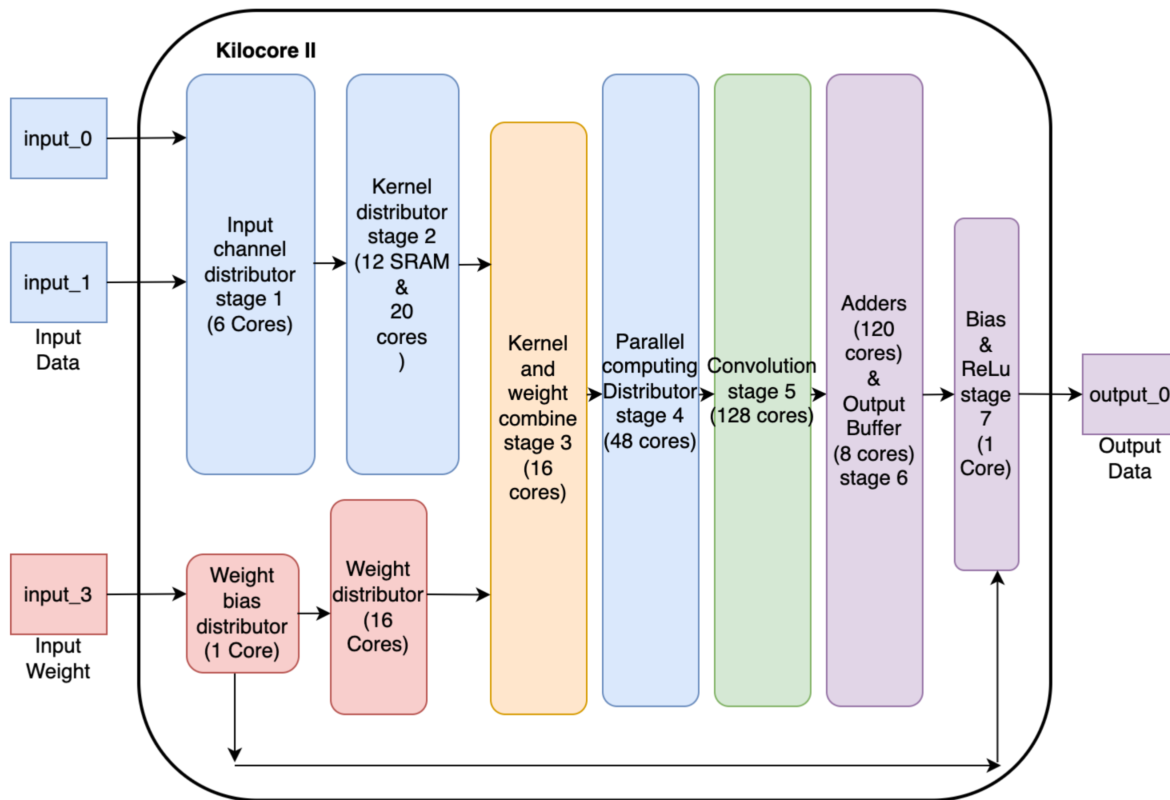


Figure 4.6: The block diagram example of the KiloCore when processing the fire squeeze layer 7 or processing the fire squeeze layer 8

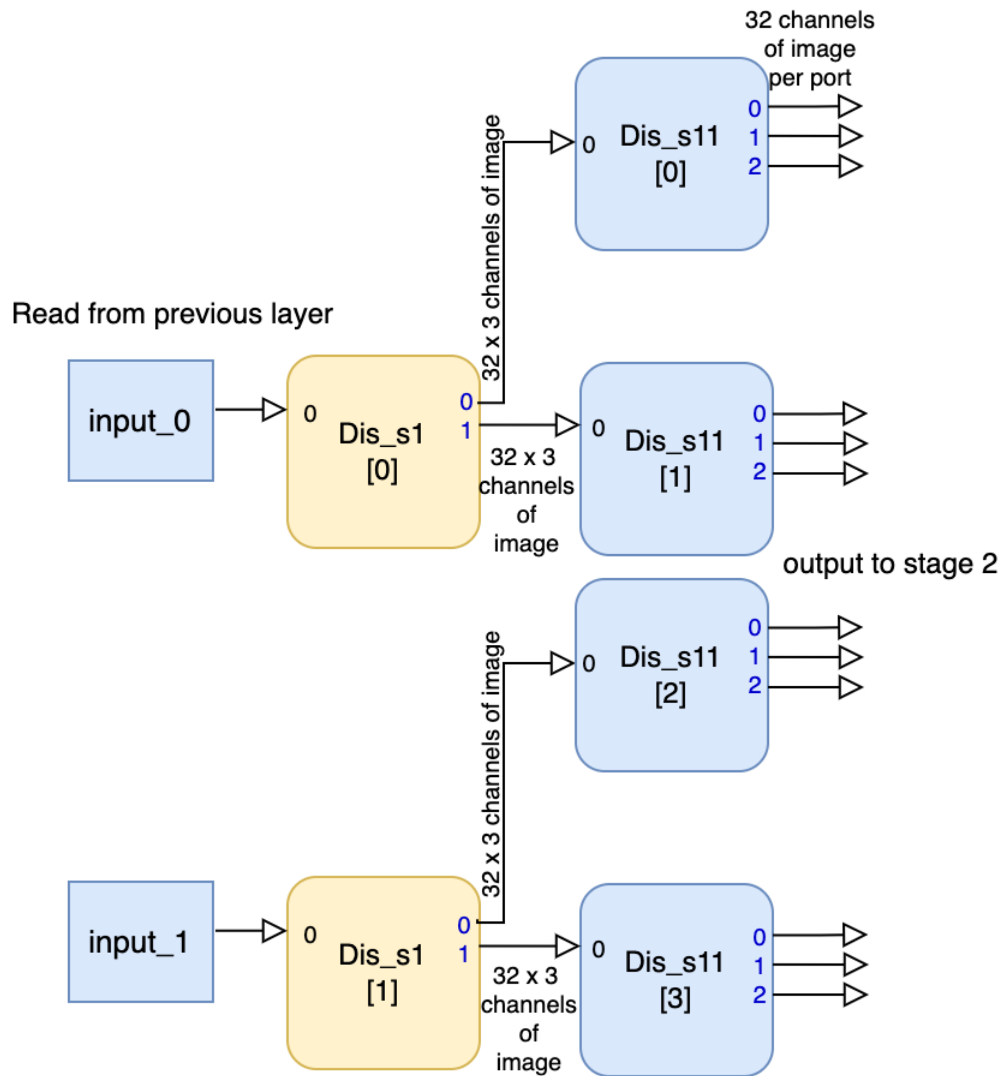


Figure 4.7: The block diagram of squeeze layer stage 1 — Example of the fire squeeze layer 7 or layer 8. The input_0 and input_1 are the input data from the previous SqueezeNet layer.

The codes of the Dis_s1 and Dis_s11 cores are the same in roles, and the difference is that the output ports quantity and the data counts are distributed to each output port. The data is sent from input directly into output ports to minimize any internal variables read and write time. The Algorithm 6 represents the code example for the Dis_s11.

Algorithm 6 Squeeze Distributor: Stage1

```
1: loop(i < image_size × image_size × depth)
2:     Output_0 ← Input_0        ▷ Pass data from input port 0 to output port 0
3: end loop
4: loop(i < image_size × image_size × depth)
5:     Output_1 ← Input_0        ▷ Pass data from input port 0 to output port 1
6: end loop
7: loop(i < image_size × image_size × depth)
8:     Output_2 ← Input_0        ▷ Pass data from input port 0 to output port 2
9: end loop
```

Squeeze Layer Stage 2: Kernel Distributor

Stage 2 of the squeeze layer used SRAMs to store the portion of the image and output the data in kernel order. After that, the output port number is unified to 16 output ports, which allows the core structure to be the same as stage 3 for different layers of the SqueezeNet. Stage 2 is built up by two sub-stages, and the one-fourth core organization is shown in Figure 4.8 which contains five cores and three SRAMs. The whole structure contains 20 cores and 12 SRAMs.

The first sub-stage on the left of the figure uses 12 SRAMs to save the whole image and generate the kernel in Z-X-Y order. Each SRAM needs to save one-twelfth of the total image which is $27 \times 27 \times 32 = 23,328$ of 16-bits data. Each core in sub-stage one has one input port and two output ports. There are three types of cores in sub-stage 1. The reason is that there are 12 SRAMs and need to send outputs to 16 output ports to stage 3. This is also why the hardware structure is slightly different based on the input data size. The upper-left core (Dis_s21[0]) in Figure 4.8 outputs 24 depth of the kernel in Z direction to output port 0 first and eight depth of the kernel in Z direction to output port 1 next. The left middle core (Dis_s22[0]) outputs 16 of the kernel in Z direction first and 16 of the kernel in Z direction depth next. The lower left core (Dis_s23[0])

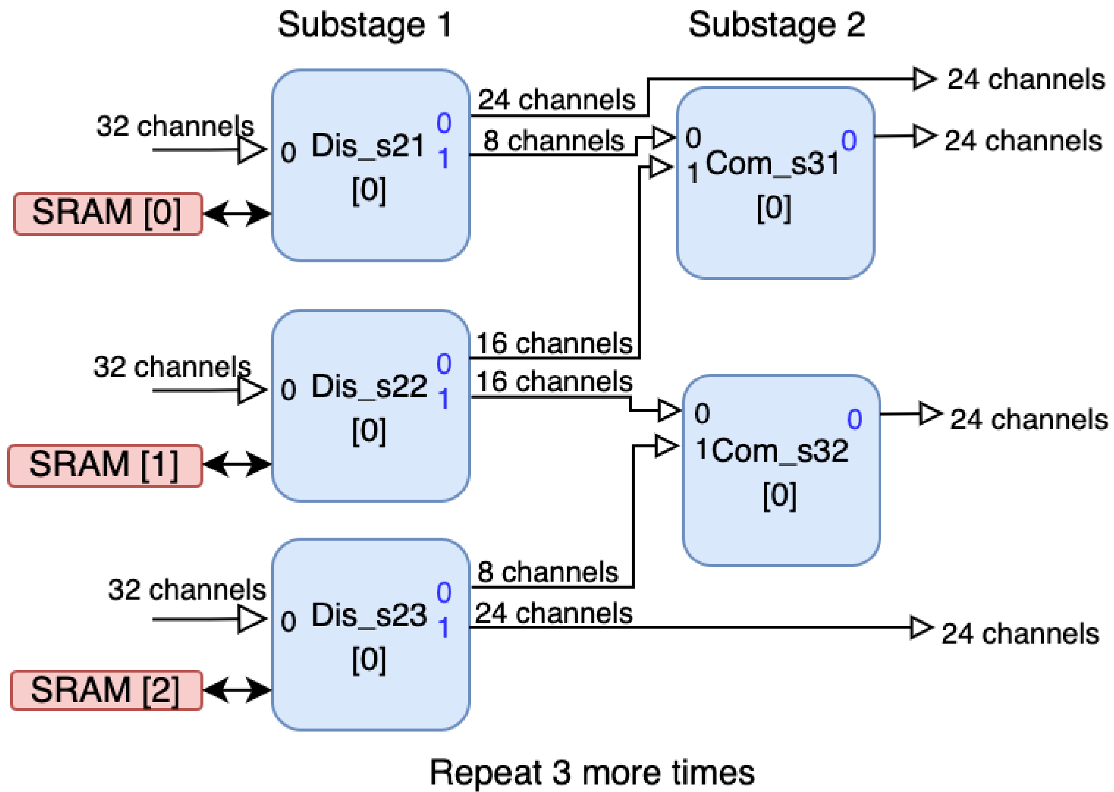


Figure 4.8: Kernel distributor block diagram — An example of the fire squeeze layer 7 or fire squeeze layer 8. Each of the cores in sub-stage 1 is connected with an SRAM. The same structure is repeated 3 more times for 384 channels.

outputs the 8 depth of the kernel in Z direction first and 24 depth of the kernel in Z direction next. All of the channels outputted to the ports in the sub-stage 1 add up to 32 depths. The kernel is outputted in Z-X-Y direction. Algorithm 7 shows the algorithm of Dis_s21. Two of the loops are used to read the image into the SRAMs. One loop is for depth and the other loop is for the image width and height. There are five loops used for the output ports. The most internal loop is used to change the depth number and the second and third internal loops are responsible for shifting the kernel right and down. The outermost loop is used to repeat the whole output process multiple times for different weights and biases. The processing speed of this sub-stage is mainly limited by the reading and writing speed of the SRAM access.

Algorithm 7 Squeeze Distributor Stage 2 Sub-stage 1: Kernel Distributor

```
1:  loop(i < image_depth/12)
2:      loop(i < image_height * image_weight)
3:          SRAM ← Input_0                                    ▷ Read 32 depth of image from port 0
4:      end loop
5:  end loop
                                                              ▷ Output the kernel to the output ports
6:  loop(k < repeat_times)                                    ▷ Repeat N times
7:      loop(y < image_height)                                ▷ Shift kernel in vertical side
8:          loop(x < image_width)                             ▷ Shift kernel in horizontal side
9:              loop(i < depth_port0)                         ▷ Change the depth output
10:                 SRAM[x + y * image_height + i * image_height * image_width] → Output_0
    ▷ The output for port 0
11:             end loop
12:             loop(i1 < depth_port0)                        ▷ Change the depth output
13:                 SRAM[x + y * image_height + i1 * image_height * image_width] → Output_1
    ▷ The output for port 1
14:             end loop
15:         end loop
16:     end loop
17: end loop
```

The purpose of sub-stage 2 is to regulate and combine the output from sub-stage 1, which allows the structure to apply different dimensions of the input data. The core Com_s21 first directs eight numbers from input port 0 to output port 0. Then 16 numbers are directed from input port 1 to output port 0. This sub-stage does not contain any computation tasks, so the processing speed is fast. Two input ports and one output port, which allow both sub-stage close to each other to reduce the data transfer time between stages.

Squeeze Layer: Weight and Bias Distributor

Figure 4.9 presents the core level block diagram of the weight distribution process, which contains 17 cores. One of the cores is used to separate the weight and bias. The rest 16 cores were used to distribute the weight. The first core weight_bias divides the bias and weights to the corresponding cores by using one loop. Port 1 of weight_bias core sends the bias data to stage 7 (final stage), and port 0 sends the weight data to the core weight_15. For the weight distribution part, each core passes the values which belong to other cores to port 0 first and then output the last 24 numbers of weight to port 1. Port one is connected to stage 3. Sixteen cores distribute the weight corresponding to the number of cores in stage 3. All cores in the weight and bias distribution section have three or fewer input and output ports, increasing the routing efficiency.

Squeeze Layer Stage 3: Kernel and Weight Combination

The core level structure of stage 3 is shown in Figure 4.10. This stage combines the kernel input from stage 2 and the weight input from the weight distributor, as introduced in the last section. There are 16 cores in total and each core has two input ports and one output port. The same weight data repeatedly used $image_size \times image_size$ times for different kernels, so the data is saved in an array for reuse. The 24 kernel values are first read from port 1 and saved into an array, then the image data is read from port 0. Finally, the weights and images are interleaved and output to port 0 for computation. The Algorithm 8 shows the code of the combine_weight_kernel core.

Squeeze Layer Stage 4: Parallel Computing Distributor

The parallel computing stage is implemented using a tree structure as shown in Figure 4.11. Each 24 kernel and corresponding weights is treated as a dataset from stage 3. The dataset is

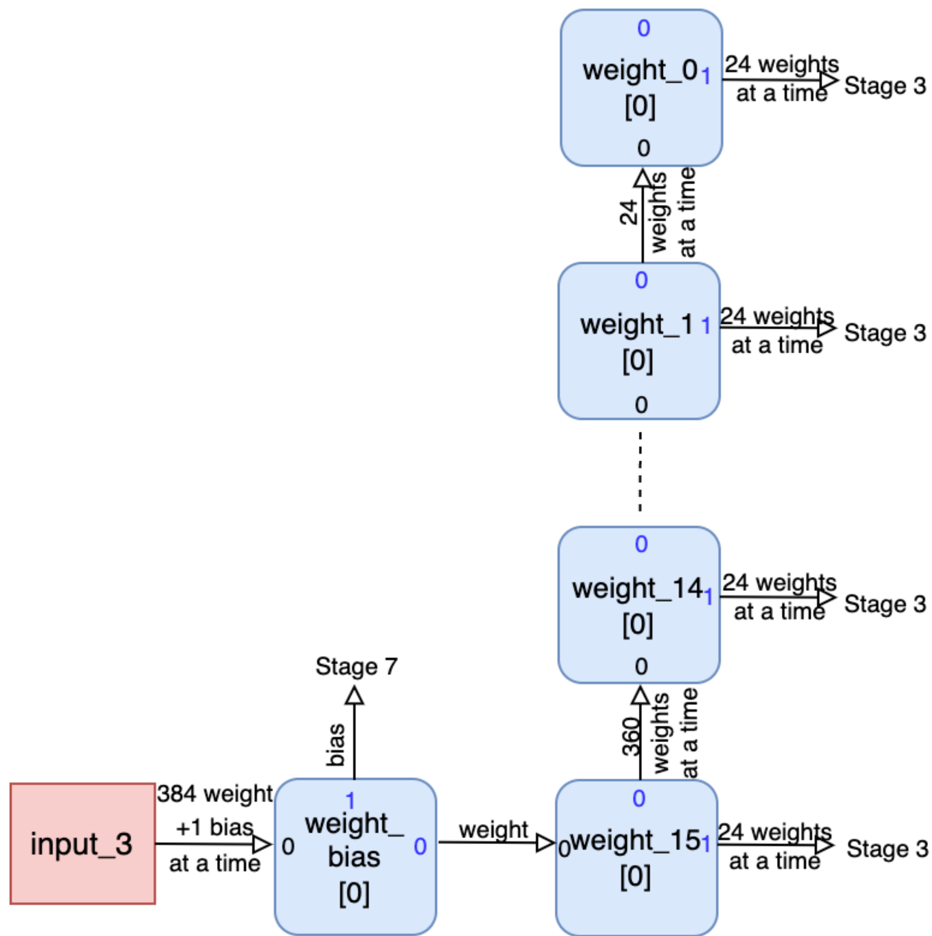


Figure 4.9: The block diagram of the weight and bias distributor stage of the fire squeeze layer

Figure 4.10: The core diagram of the weight and bias combination stage

Algorithm 8 Squeeze Weight and Kernel Combine: Stage 3

1: **loop**($i < total_layer_size/16$)
2: $weight[i] \leftarrow Input_0$ ▷ Save data to an array
3: **end loop**
4: **loop**($k < image_size \times image_size$) ▷ Repeat the weight for 27×27 times
5: **loop**($i < total_layer_size/16$)
6: $Output_0 \leftarrow Input_0$ ▷ Send the kernel to output port 0
7: $Output_0 \leftarrow weight[i]$ ▷ Send the weight value from the array
8: **end loop**
9: **end loop**

distributed from 16 cores to 128 cores to enable parallel computing. Two of the sub-stages are involved in this parallel computing stage. The first sub-stage extends the 16 input ports to 32 output ports. The second stage extends the 32 input ports to 128 output ports. By adding this extra stage, the whole design achieve an $8\times$ speedup in total. The algorithm used in this stage is the same as Algorithm 6 used in the maxpool section.

Squeeze Layer Stage 5: Convolution Stage

Stage 5 is the convolution stage. There are 128 in total, and each core has one input port and one output port, as shown in Figure 4.12. Each core summarizes the convolution computation of 24 depth of kernels and sends the result to the next stage by port 0. The computation tasks for different layers are different, the task calculation amount is calculated by total depth over 16. Kernels and weights are input sequentially interleaved, which speeds up the calculation by preventing the kernel or weight are written into an array and read out. Algorithm 9 presents how the summarized result is calculated. The kernel width and kernel height both have one times one dimension weight and height in the squeeze layer.

Squeeze Layer Stage 6: Adder and Output Buffer Stage

Stage 6 is built up of two separate sub-stages. The first sub-stage uses the adder cores to add up the convolution result of each depth together. The second sub-stage changes the parallel result to the sequential result.

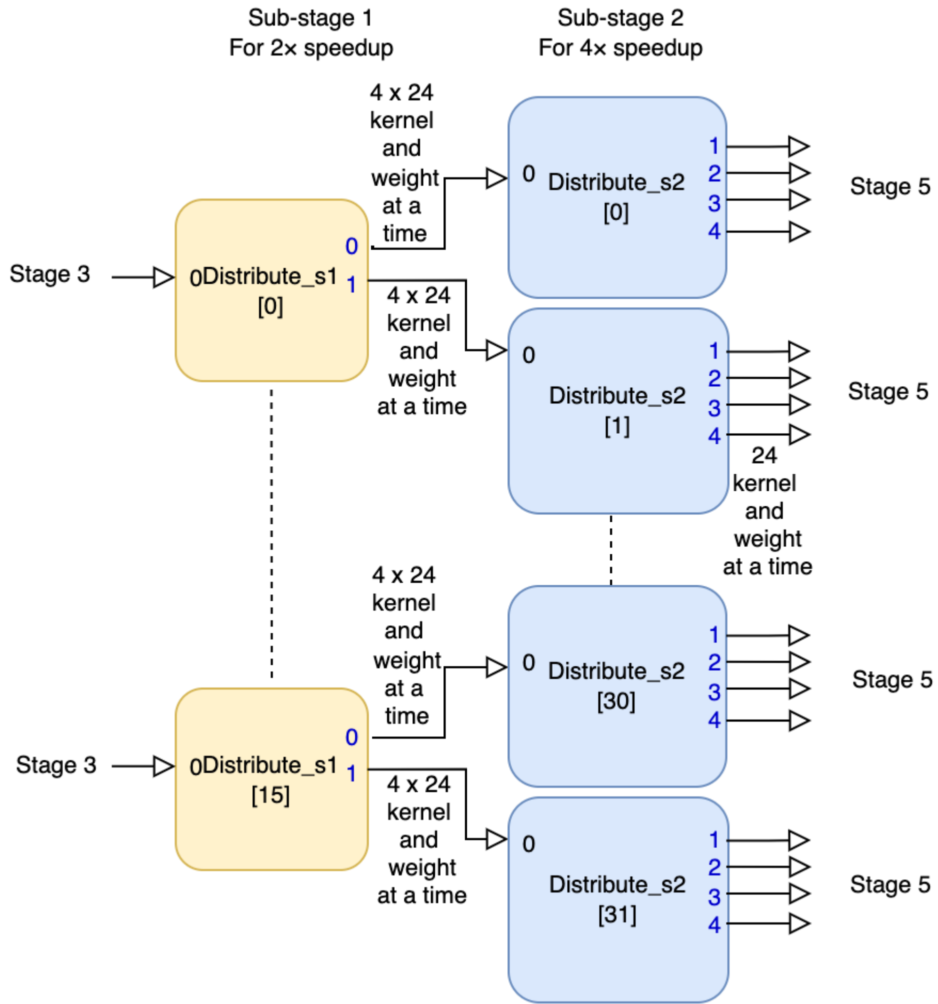


Figure 4.11: The block diagram of the parallel computing distributor

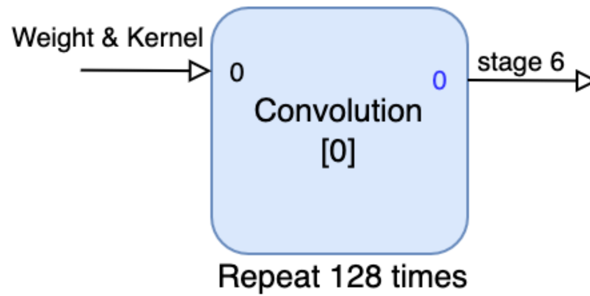


Figure 4.12: The core diagram of the convolutional layer

Algorithm 9 Squeeze Computation: Stage 5

1: **loop**($i < total_layer_size/16 \times kernel_width \times kernel_height$)
2: ▷ Repeat different kernel and different depth
3: $kernel \leftarrow Input_0$
4: $weight \leftarrow Input_0$
5: $sum = sum + kernel \times weight$ ▷ Sum is equal to kernel times corresponding weight
6: **end loop**
7: $Output_0 \leftarrow sum$ ▷ Sum is outputted to the output port 0

The core number needs to calculate the sum of all channels, as shown in Equation 4.2 because the maximum input port number is two. There are eight kernels calculated at the same time for layer 7/8, so the total core need for this sub-stage is 120×8. Figure 4.13 shows the example of the adder stage.

Overflow can possibly happen in the adder tree stage, which requires some logic to prevent it from happening. Algorithm 10 shows the overflow checker implemented in the adders, which uses saturation logic to cap the result at maximum or minimum to preserve the sign bit of the convolution output.

Algorithm 10 Overflow Handling

1: num1 ← Input
2: num2 ← Input
3: sum ← Input
4: **if** (sign of num1 == sign of num2) **then**
5: **if** (num1 is negative & sum is positive) **then** ▷ neg + neg = pos - overflow
6: 0x8000 → Output ▷ Return negative min
7: **else if** (num1 is positive & sum is negative) **then** ▷ pos + pos = neg - overflow
8: 0x7FFF → Output ▷ Return positive max
9: **end if**
10: **else**
11: sum → Output ▷ No overflow
12: **end if**

The output buffer of the sub-stage uses eight cores to output the paralleled eight computa-

tion results back into the sequence order. Each core has two input ports and one output port to transfer the data. The structure is as same as the weight distributor stage, and the diagram is the same as the right side of Figure 4.9.

$$Adders_number = 2^{Paralleled_input_number} - 1 \tag{4.2}$$

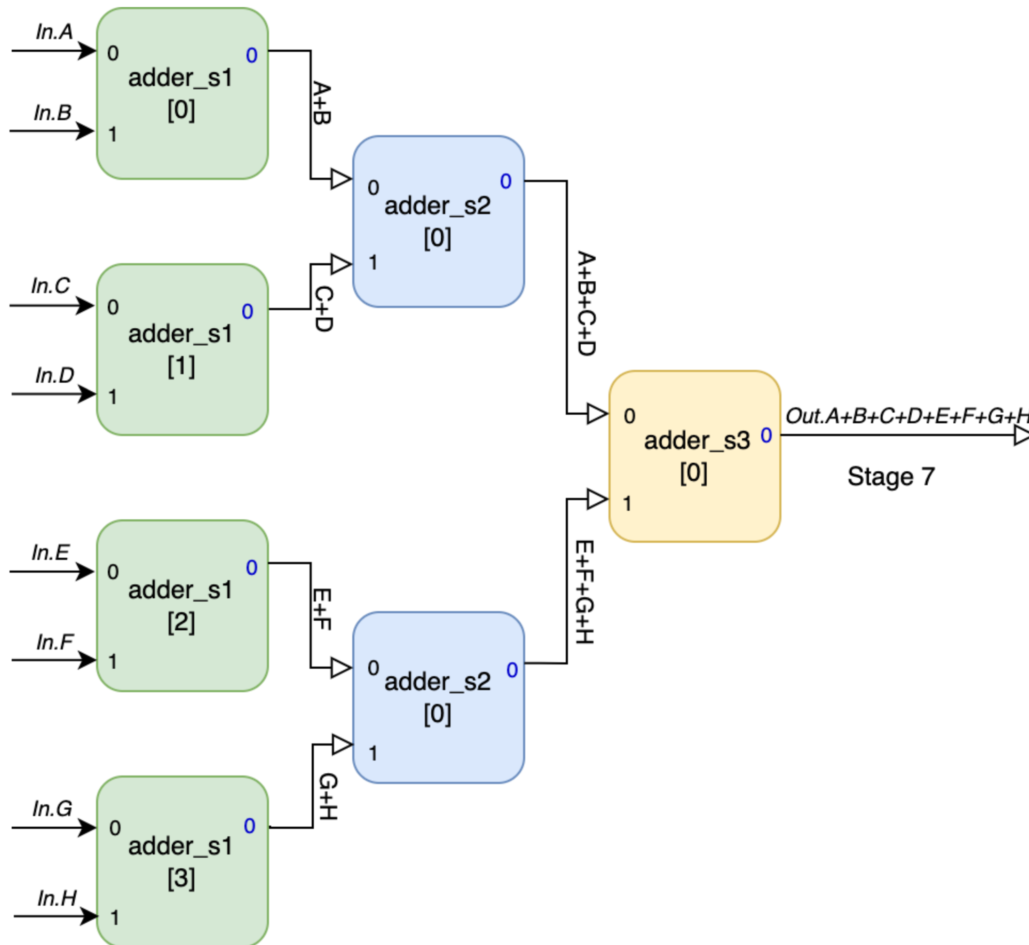


Figure 4.13: The block diagram of the adder stage, which adds eight numbers together.

Squeeze Layer Stage 7: Bias and ReLU Stage

The last stage of the squeeze layer adds bias value into the convolution result and filters out the negative number. There is only a small amount of computation tasks in the ReLU and bias stage, so using multiple cores is unnecessary. The Algorithm 11 presents the details of the logic, and the Algorithm 10 is used while adding the bias to prevent the overflow. Due to the bias value needs to be used multiple times so the bias value is first saved to a temporal variable and it is added

$image_width \times image_height$ times to the convolution result. Finally, the final output is sent to output port 0.

Algorithm 11 Squeeze Computation: Stage 7

```
1: bias ← Input_1
2: loop(i < image_width × image_height)
3:                                          ▷ Repeat add bias for the full image
4:     conv_rst ← Input_0
5:     rst = conv_rst + bias
6:     if rst < 0 then
7:         Output_0 ← 0
8:     else
9:         Output_0 ← rst
10:    end if
11: end loop
12: Output_0 ← sum                          ▷ Output the result to output port 0
```

Squeeze Layer Mapping

The mapping diagram of 8, 12, and 13 SRAM mapping diagrams are shown in Figure 4.14, Figure 4.15, and Figure 4.16. The bottom row represents the SRAMs. Figure 4.14 is the architecture of eight SRAMs which uses least amount of SRAMs and cores to distribute the image before the computation stage. Figure 4.15 and Figure 4.16 are the architecture of using 12 and 13 SRAMs to distribute the image before the computation stage respectively. There are no routing congested links in the squeeze layer mapping.

4.4.2 Expand: 1×1 and 3×3 convolutional layers

The expand layer is built up by 1×1 convolutional layer and 3×3 convolutional layer. There are two design options to approach the chip-level design of the expand layer. The first option is to use the same workflow for both 1×1 convolution and 3×3 convolutions. The advantage is that more of the cores can be used for both cases so the bandwidth is higher. The disadvantage is that each core needs to have more inside logic to handle both tasks. An internal indicator needs

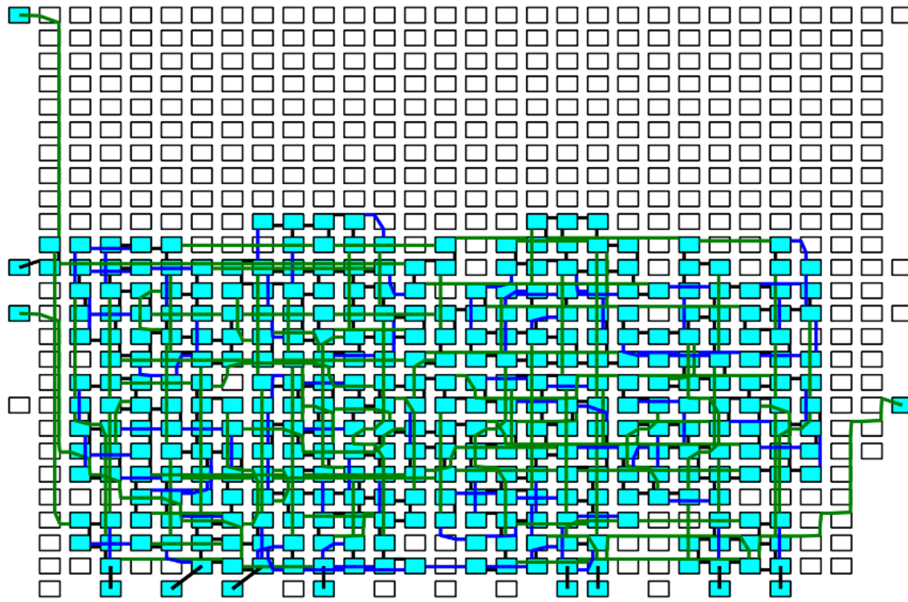


Figure 4.14: The squeeze layer architecture mapping to the KiloCore processor array with eight SRAMs.

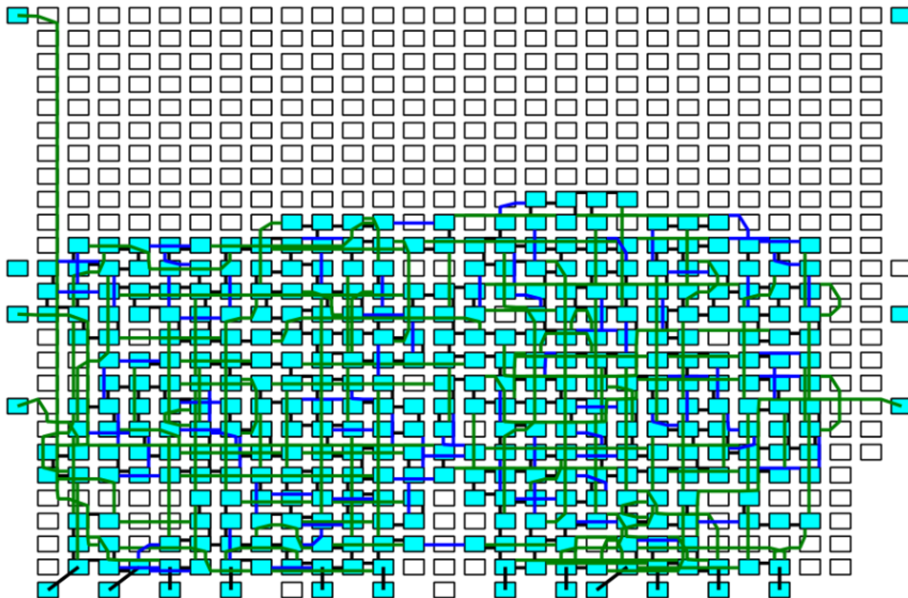


Figure 4.15: The squeeze layer architecture mapping to the KiloCore processor array with 12 SRAMs.

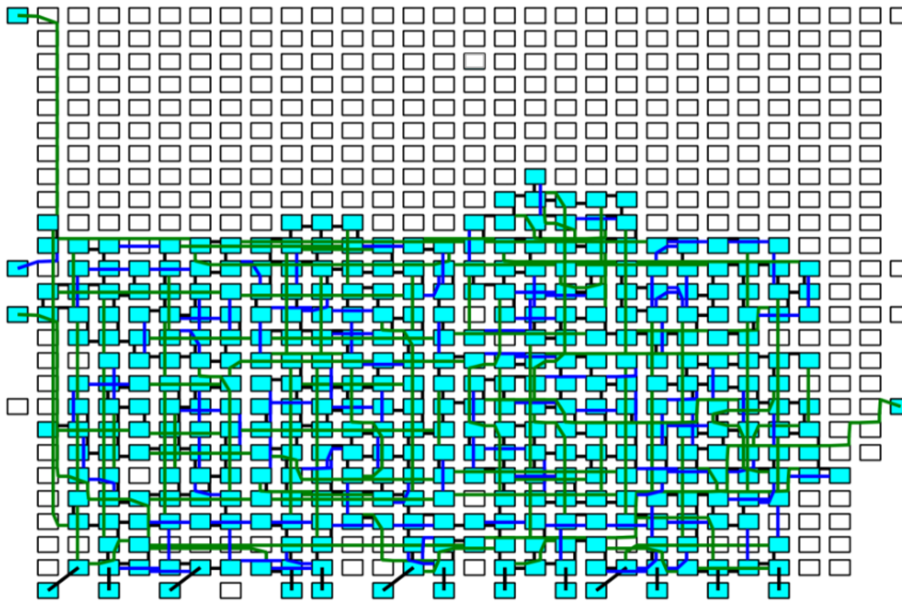


Figure 4.16: The squeeze layer architecture mapping to the KiloCore processor array with 13 SRAMs.

to be placed on each core to distinguish the tasks. The internal indicator will slow down the total speed of the processing speed and increase the computation tasks for each core.

The other design choice is to have two independent sets of cores to compute the result for 1×1 convolutions and 3×3 convolutions. The benefit is that the hardware resources can be divided corresponding to the computation tasks. By separating two convolution tasks, each core would have fewer computation tasks, although the peak bandwidth is lower than the design option 1. Option 2 is chosen in the design for clearer logic and higher scalability.

All expand layers are implemented in KiloCore with the same core-level architecture because the input data of any expand layer is small enough to save to four SRAMs. The chip-level diagram is shown in Figure 4.17. The channel number is the ratio of four so it can be fit on either four SRAMs or eight SRAMs. The 1×1 and 3×3 convolution part are totally independent to each other. The 1×1 is shown on the top of figure 4.17 and the 3×3 is shown on the bottom of figure. The image and weight of 1×1 convolution input are saved in the input_0 and input_1 files. The output data is save to the output_0. For the 3×3 convolutional layer, the image and the weight are saved in the input_2 and input_3 and the output is saved to the output_1. The computation speed between the 1×1 and 3×3 convolution is $9\times$ difference if both computations uses the same amount of hardware resources. Therefore, in order to balance the output time, much more cores

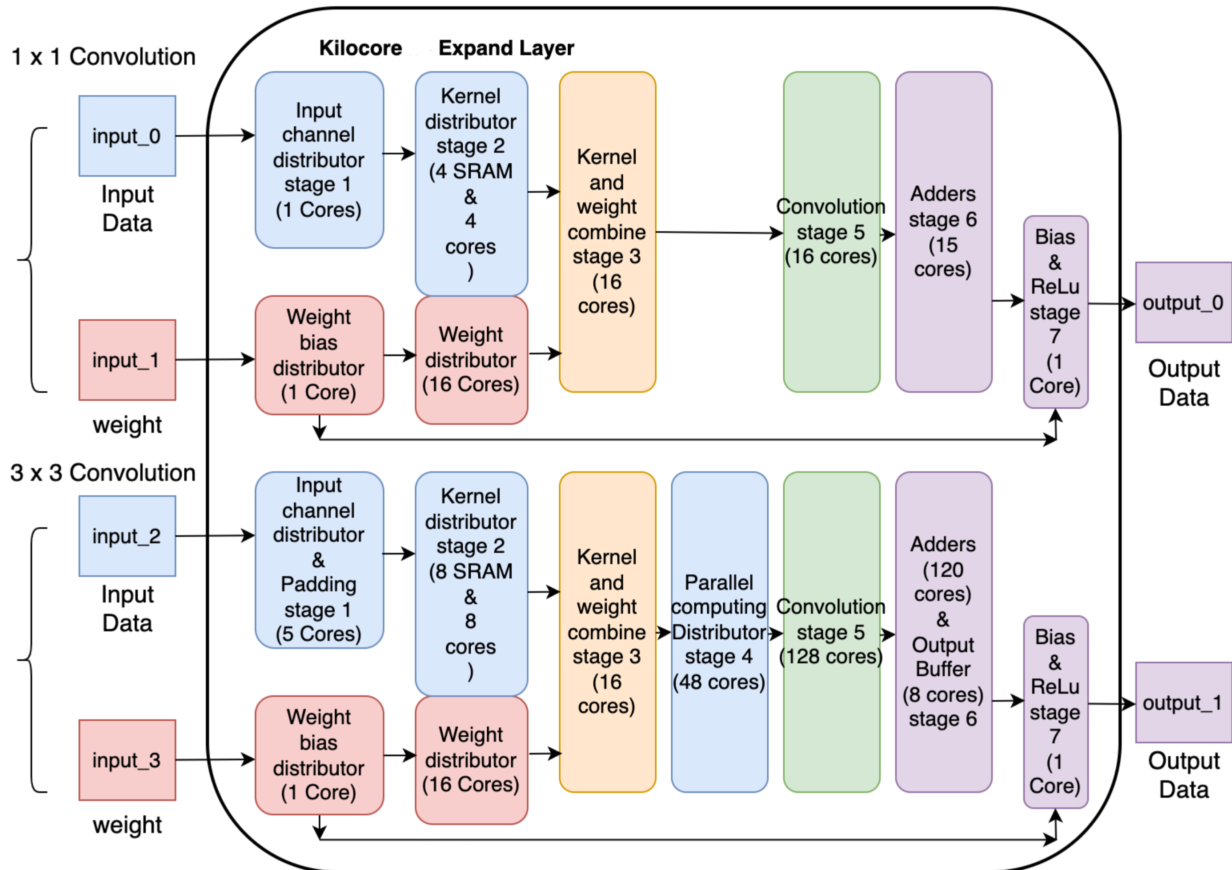


Figure 4.17: The chip-level diagram of the expand Layer which is built up by applying both 1×1 convolutional layer (upper) and 3×3 convolutional layer (lower) to the same input data.

and SRAMs are used on the 3×3 part. The three main differences between the 1×1 and 3×3 core level difference are padding, SRAM core number and convolution core number. The padding is only required for 3×3 convolution calculations to maintain the same output weight and height with 1×1 convolution result. The padding structure of 3×3 convolution is added in stage 1 before being saved to the SRAM in stage 2.

Stage 2 (Kernel distributor) of the 1×1 convolution uses four SRAMs for image storage and kernel distribute. The data is evenly distributed into four pieces in the Z direction. Each core has four output ports, so there are 16 (4×4) output ports in total to match stage 3. The output kernel generated in stage 2 is sent to stage 3 for kernel and weight combination. Both stage 3 and stage 5 (convolution stage) have 16 cores so the parallel computing distributor stage is not needed for the 1×1 stage. The 1×1 convolution is 9 times faster than 3×3 convolution. To allow both calculation complete at similar time, stage 5 for 3×3 convolution needs to have more cores than 1×1 convolutional layer. The 3×3 convolution needs the parallel computing distribution stage to distribute the data to 16 cores to 128 core convolution cores. It is same as the squeeze layer shown in Figure 4.11 on page 35. The core number ratio speeds the computation speed for the 3×3 convolution computation 8 times faster when compared with the 1×1 convolution.

In conclusion, the expand stage uses 421 cores and 12 SRAMs in total. The 1×1 convolution uses 70 cores and the 3×3 uses 351 cores. The mapping diagram of expand layer is shown in Figure 4.18. There are a few red routing congested links in the squeeze layer mapping. Most of the links are not red which means the tasks are evenly distributed between most of the cores. All of the connected cores are on the bottom of the diagram because the cores stays closer to each other based on the setting in the mapping tool and the SRAMs positions are on the bottom of the map.

4.4.3 Layer 1: 7×7 convolutional layer

The 7×7 convolutional layer only appears in the first layer. The 227×227 RGB format image is the input image. There are two approaches to optimize the performance. The first approach is to read 7 rows at a time from the output file and shift two rows down at a time while the kernel approach to the end of each row. The second approach is to save the entire image on the SRAM for reuse.

In the first approach, the design does not occupy large SRAM memory space but the

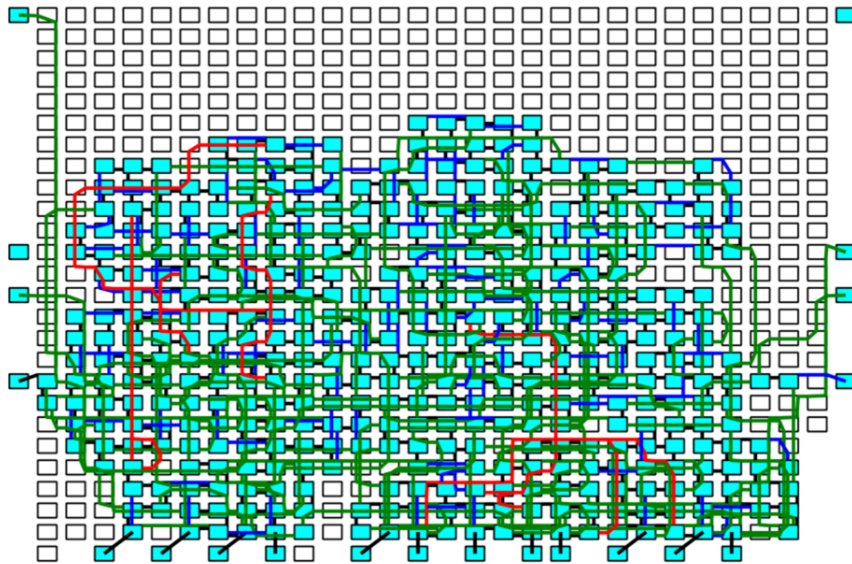


Figure 4.18: The expand layer architecture mapping to the KiloCore processor array with 12 SRAMs.

disadvantages are KiloCore is input triggered, so the full image needs to be saved into the 1GB DRAM multiple times bases on the output channel number. The kernel shifting would require frequent reads and writes of SRAM memory. For the second approach, the SRAM has higher usage and requires more SRAMs to save the entire image but it would save the memory space for both on-chip and off-chip memory. Based on the implemented result, approach two is faster than approach one. One of the reasons is that KiloCore is input triggered, so the input image needs to be sent from outside of the chip for calculating different output channels.

The chip-level block diagram of the 7×7 convolutional layer is shown in Figure 4.19. The structure is same at the 1×1 convolutional layer with some of the core level modifications to fit three input channels.

The significant difference when comparing this layer and the squeeze layer inner structure is in stage 2, as discussed below. For one channel of image ($227 \times 227 = 51{,}529$) numbers of data is too many for the SRAMs (31,768), so each channel is saved to two of the SRAMs. Six SRAMs in total are used to save red, green, and blue channels. To generate the kernel for up half and bottom half of the channel, 5 rows are overlapped, so the 5 rows in the middle need to be saved to both SRAMs for each channel. Then, one sub-stage in stage 2 collects the kernel data from both SRAMs in order, so those two small SRAMs would work as same as one larger SRAM. Then the three stages of the parallelism stage distribute three cores to $3 \times 4 \times 4 \times 4$ (192) convolution cores. Stage 6 adds

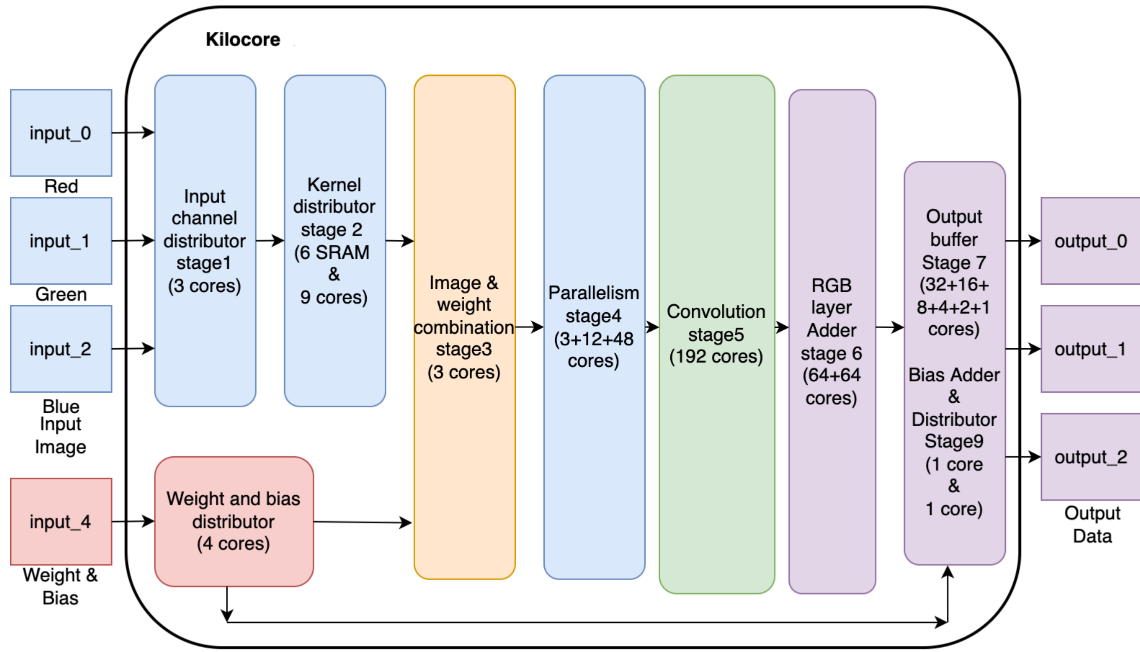


Figure 4.19: The chip-level diagram of the 7 × 7 convolutional layer. This is the layer 1 of the SqueezeNet.

the three different channel convolution results together by adders. Then the output buffer stage changed the parallel format to the string structure by 6 tree structure output buffer as shown in Figure 4.4. The last stage adds the bias to the convolution result and applied the ReLU filter to the result. In the end, the 96 output channels are evenly distributed to three output files by two cores to match the maxpool layer as discussed on page 19.

The mapping architecture of the 7 × 7 convolutional layer is shown in Figure 4.20. The cores are evenly distributed around the KiloCore. The six SRAMs are in the bottom of mapping figure.

4.4.4 Summary

In conclusion, the SqueezeNet implementation is built up by three main parts, which are distribution, computation, and output buffer part. Each part has several stages to achieve the goal. The distribution part distributes the image and the weight first and then paralleled the data for calculation. The computation part calculates the convolutions and adds different channels together. The output part changes the parallel computation result back into the sequential order and sends the data to the next layer.

Figure 4.21 and Figure 4.22 give a summary of the chip-level architecture of the whole

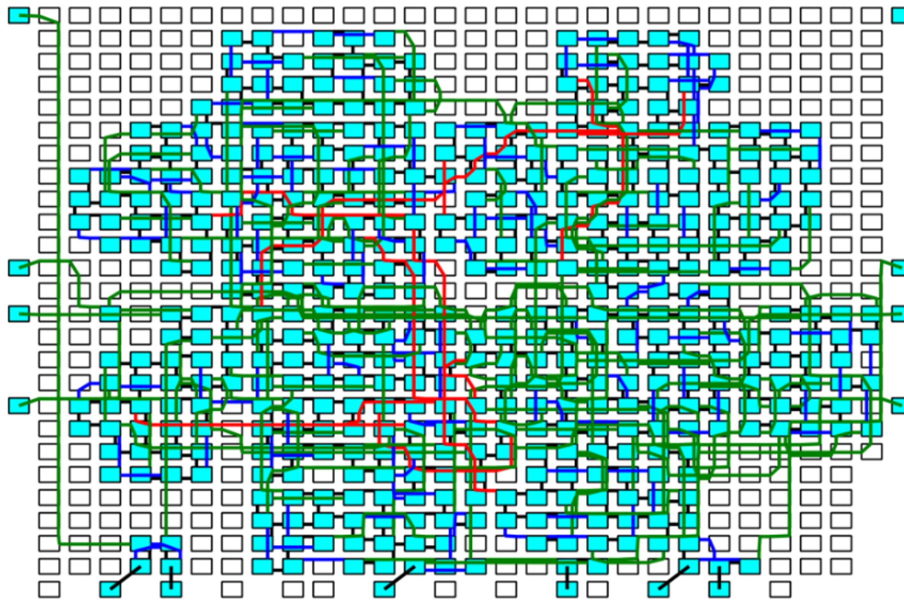


Figure 4.20: The 7×7 convolutional layer architecture mapping to the KiloCore processor array with six SRAMs.

SqueezeNet structure and the connection between different layers. Each set of blue and red boxes represents one KiloCore chip. There are 21 KiloCores in total because the Conv10 and Avgpool layers share one KiloCore chip. The port number of the adjacent layer is matched for transferring the data between layers.

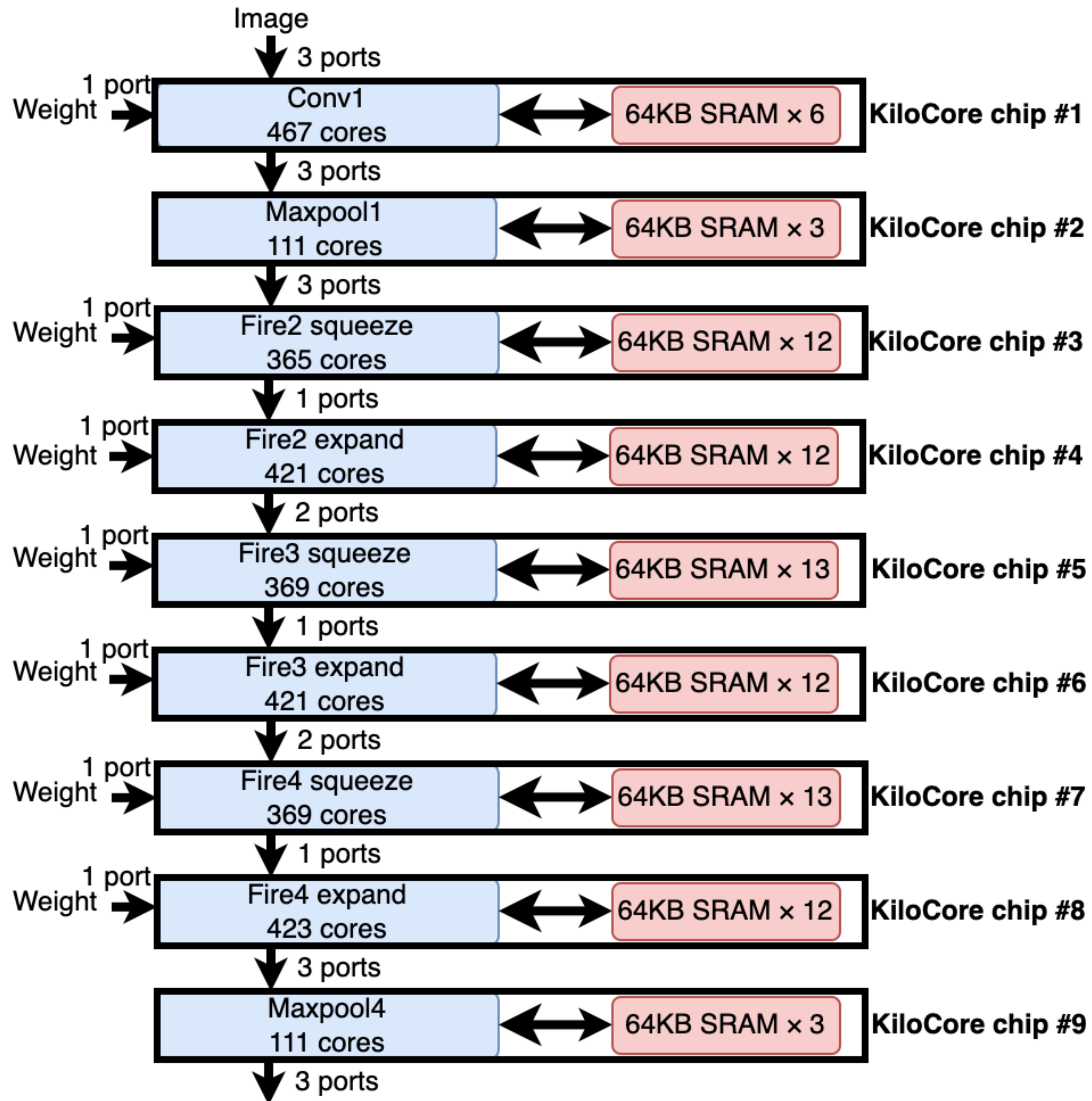


Figure 4.21: The top 9 layers of the SqueezeNet. Each set of blue and red boxes represents one KiloCore chip.

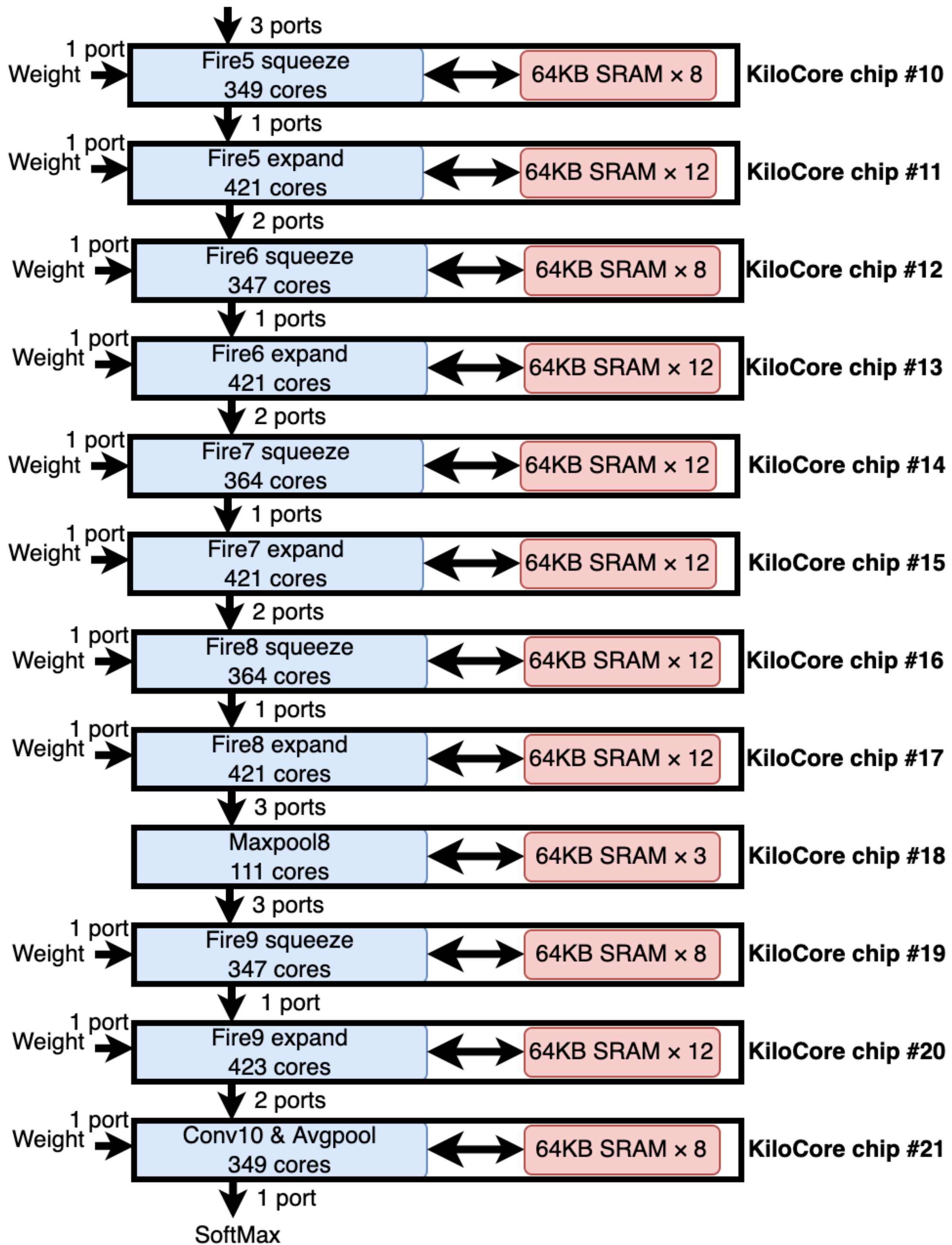


Figure 4.22: The bottom 13 layers of the SqueezeNet. 12 KiloCore chips in total because last two layers are combined.

Chapter 5

Inference Evaluation

5.1 Inference Evaluation

5.1.1 Overview

The functionality of the chip-level inference architectures on KiloCore has been verified using random inputs generated by MATLAB. A real-world traffic light image has also been used as the input data to test inference performance, as shown in Figure 5.1. The sample image is cropped to a square shape and converted to the RGB format. The color information of each pixel is quantized to the 16-bit fixed-point, and the final prediction output is compared with the original 32-bit floating-point SqueezeNet implementation. The data type conversions and the result visualization are performed in MATLAB.

All of the simulation results are generated by the Project Manager and a cycle-accurate C++ simulator for KiloCore platform. The simulator generates the metrics, including power, energy and throughput measurements which can be used for performance optimization and the comparison between different hardware platforms.

5.1.2 Inference Result

MATLAB implementation [30] of the SqueezeNet is the golden reference for comparison with the KiloCore inference. The element-wise mean and max error are calculated by Equation 5.1 and Equation 5.2. The relative mean error is calculated by Equation 5.3 and Equation 5.4. The calculation is not element-wise because the zero output with a tiny error will cause the error

Figure 5.1: The traffic light image being used to simulate the SqueezeNet

percentage to be infinite. Table 5.1 shows the calculation result of the error of each layer. The result is plotted in Figure 5.2. The final result is a vector with a length of 1,000 numbers which is fed into the softmax layer. The softmax layer is done by MATLAB using the softmax function directly to get the classification result because softmax has the exponential computation. For the sample image prediction accuracy, the golden reference results in a confidence of 94.72% and the confidence of KiloCore implementation with the confidence of 94.13%.

$$mean\ error = \texttt{mean}(|image_{\ Golden\ Reference} - image_{\ KiloCore\ output}|) \quad \text{(element-wise)} \tag{5.1}$$

$$max\ error = \texttt{max}(|image_{\ Golden\ Reference} - image_{\ KiloCore\ output}|) \quad \text{(element-wise)} \tag{5.2}$$

$$relative\ mean\ error = \frac{mean\ error}{max\ value} \tag{5.3}$$

$$relative\ max\ error = \frac{max\ error}{max\ value} \tag{5.4}$$

5.1.3 Simulation Measurements

The simulation environment is set to have the same amount of available resources as one KiloCore chip, so the result of each layer can be reproduced on the real KiloCore Test System board. The simulation measurement calculated in this section is set to a KiloCore processor array to calculate the off-chip DRAM power consumption accurately. The frequency is @ 1.24 GHz, and the voltage is 900 mV. There is a total of 21 KiloCore implementations, because the conv10 and

Layers	Mean error	Max error	Relative mean error	Relative max error	Max value
Conv1	0.0217	0.1992	0.0000	0.0003	653.22
Maxpool1	0.0302	0.1986	0.0000	0.0003	653.22
Fire2 Squeeze	0.0729	0.9472	0.0000	0.0006	1504.78
Fire2 expand	0.0330	1.3858	0.0000	0.0013	1054.29
Fire3 squeeze	0.1294	2.4537	0.0001	0.0023	1072.5
Fire3 expand	0.0510	2.3496	0.0001	0.0024	974.9
Fire4 squeeze	0.0647	2.2609	0.0001	0.0021	1094.49
Fire4 expand	0.0406	3.1144	0.0001	0.0045	696.37
Maxpool4	0.1195	2.8491	0.0002	0.0041	696.37
Fire5 squeeze	0.2296	5.8648	0.0002	0.0057	1034.17
Fire5 expand	0.0901	5.1269	0.0001	0.0048	1062.08
Fire6 squeeze	0.1813	5.6470	0.0002	0.0051	1101.46
Fire6 expand	0.0396	4.8342	0.0000	0.0054	888.53
Fire7 squeeze	0.1108	4.0189	0.0002	0.0066	608.42
Fire7 expand	0.1625	4.3742	0.0003	0.0085	512.81
Fire8 squeeze	0.3728	8.1547	0.0008	0.0177	459.67
Fire8 expand	0.0727	6.0500	0.0002	0.0151	401.05
Maxpool8	0.1961	5.4176	0.0005	0.0135	401.05
Fire9 squeeze	0.7775	7.9546	0.0014	0.0143	556.936
Fire9 expand	0.0638	7.8393	0.0002	0.0216	363.03
Conv10 & Avgpool	0.1832	3.4784	0.0015	0.0287	121.3

Table 5.1: The mean and max error with the relative error of each layer in SqueezeNet calculated by Equation 5.1, 5.2, 5.3, 5.4.

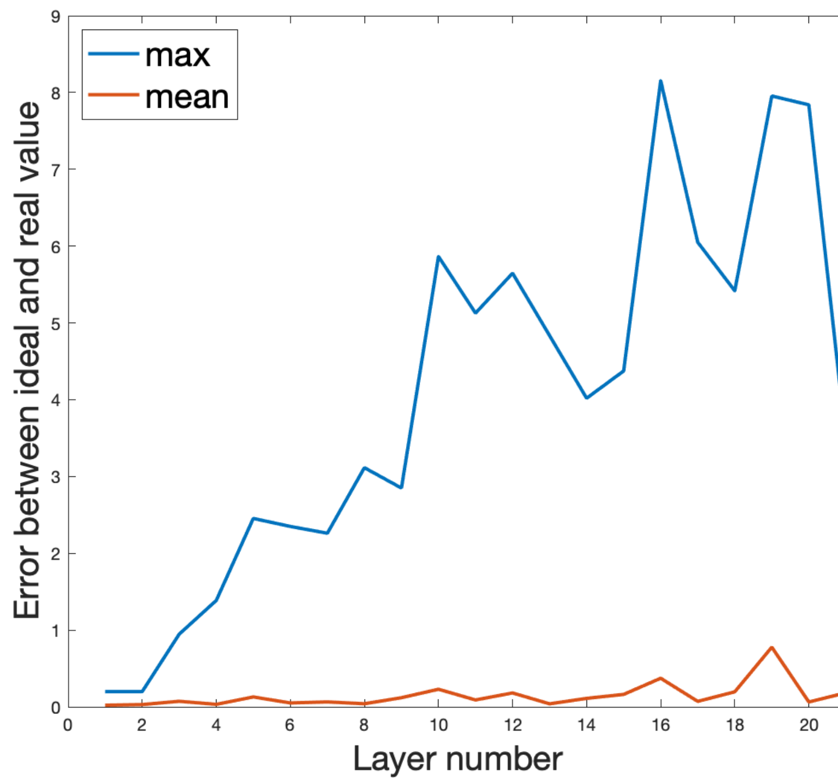


Figure 5.2: The mean and max error plot of each layer in SqueezeNet. The blue line is the max value and the orange line shows the mean value.

avgpool layers share one KiloCore chip. The total cores required to maintain the same routing architecture are 697×21 = 14,637, with 14×21 = 294 SRAMs. For the total number of input ports, three 16-bit input ports for image and 18 input ports for weight and bias as presented in Figure 4.21 and Figure 4.22. One 16-bit output port is used to output the final result. Energy measurements from the 32 nm CMOS fabricated chip are used as the inputs to the simulator to obtain energy data. Area usage is physically measured from the fabricated 32 nm CMOS chip, where each processor occupies 265 μm × 274.5 μm of the area, and each SRAM memory occupies 356.2 μm × 475.41 μm of area. The total area used is calculated by Equation 5.5, where *nProc* and *nMem* are the maximum number of processors and memory modules used. For the SqueezeNet inference, *nProc* is 14,637 and *nMem* is 294.

$$Area\ (mm^2) = nProc \times 0.0727\ (mm^2) + nMem \times 0.169\ (mm^2) \tag{5.5}$$

The KiloCore array works for multiple images, and it works as a pipeline so that each layer can process different images and transfer the result to the next stage. The latency is the layer with the maximum last output time, which is the first layer. Throughput (Frame/sec) is the amount data that being processed in a given amount of time. This property is called Frame rate, or frames per second (FPS). *Energy* is the sum of the total energy consumed by each layer, using Equation 5.6. Power is calculated as in Equation 5.7, which is the total energy per frame divided by total time per frame.

$$Energy\ per\ Frame\ (J/Frame) = \left(\sum_{i=1}^{Total\ Layers} Energy\ Per\ Layer_i\ (nJ/Layer) \right) \times 10^{-9} \tag{5.6}$$

$$Power\ (Watt) = Energy\ Per\ Frame\ (J/Frame) \times Throughput\ (FPS) \tag{5.7}$$

Throughput per area is calculated as the throughput (FPS) divided by the die area in mm^2 and is given by Equation 5.8.

$$Throughput\ per\ Area\ (FPS/mm^2) = \frac{Throughput\ (FPS)}{Area\ (mm^2)} \tag{5.8}$$

The Energy-Delay Product (EDP) is calculated as the energy divided by throughput as shown in Equation 5.9. A lower Energy-Delay Product means the system is more efficient.

$$EDP\ (J \times s/Frame^2) = Energy\ (J/Frame) \times \frac{1}{Throughput\ (FPS)} \tag{5.9}$$

Total memory requirement is calculated by the minimum required DRAM number multiplied by the size of each DRAM. The total 22 layers need 205 SRAMs as shown in Figure 4.21 and Figure 4.22. Therefore, the total memory requirement is 12.81 MB.

Area (mm^2)	Latency (s)	Throughput (FPS)	Power (Watt)	Energy per Frame (J/Frame)	Throughput/ Area (FPS/mm^2)	EDP (J*s/$Frame^2$)
1113.79	0.995	1.005	1.245	1.240	0.0009	1.233

Table 5.2: KiloCore performance data when processing each frame 1.24 GHz @ 900 mV by using Equation 5.5, 5.6, 5.7, 5.8, 5.9.

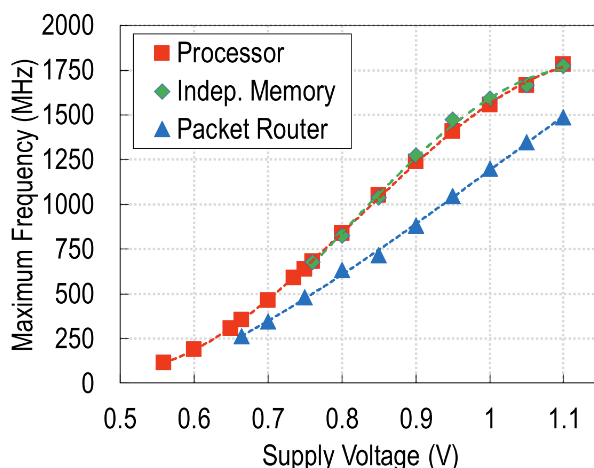


Figure 5.3: Maximum operating frequency of processors, memories, and routers [4]. The frequency of 1.24 GHz at 1.1 V condition is used for measurement.

Energy Consumption

The energy consumption @ 1.24 GHz for each layer is presented in Table 5.2. When compared between layers, layer 1 of SqueezeNet consumes the highest energy and total computation time which is mainly because layer 1 has the largest 7×7 filter size, which meets design strategies 1 and 2 mentioned in Chapter 2, therefore all other layers have filter size either 1×1 or 3×3.

Maxpool layer consumes the least power when compared to other convolutional layers. The maxpool layer does not require many hardware resources to store and process data. The output results of one channel are not affected by the data in a different channel, making the processing speed of maxpool layer faster and more efficient. Comparing the energy and output time between the squeeze layer and expand layer, the expand layer consumes 5× to 10× more than the squeeze

layer in each fire module. Squeeze layer functions as a rapid data pre-processing layer to lower the workload of the expand layer, which explains why it has a low power consumption and fast processing speed.

In summary, the energy consumption is proportional to the input image channel number, input image size, and filter size when the chip level structure remains the same. When comparing the active energy and the total energy for each layer, all of the convolutional layers have high activation energy and total energy ratio. The maxpool layer has a lower ratio due to the lower percentage of cores on the KiloCore chip used. Therefore, the higher percentage of cores on the KiloCore leads to higher energy efficiency.

High Performance Condition

The frequency and voltage in the simulator are default set to 1.24 GHz and 900 mV. The maximum performance of the KiloCore varies based on the supply voltage as shown in Figure 5.3. The maximum performance condition can be scaled to 1.1 V and 1.78 GHz and the result can be calculated based on actual the measured values in Figure 5.4, 5.5. The Energy and power is scaled based on the processor trend line in both Figures. The latency time can be calculated by dividing the energy to power, then the rest of the value can be calculated by Equation 5.8, 5.9, which is shown in Table 5.4. The result shows a large increase in throughput and energy per image.

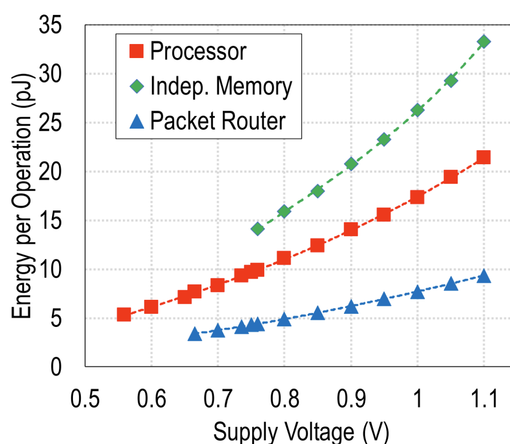


Figure 5.4: Energy per typical operation for processors, memories, and routers [4].

Layer	Active energy (nJ)	Total energy (nJ)	First output time (ps)	Last output time (ps)
Conv1	303,572,016	368,188,800	70,119,441	995,302,799,568
Maxpool1	2,487,917	5,075,606	32,304,573	18,189,762,904
Fire2 squeeze	13,045,592	14,356,027	132,153,775	24,182,918,492
Fire2 expand	31,455,102	37,938,492	168,942,051	70,520,458,933
Fire3 squeeze	4,754,659	5,532,622	506,674,749	8,577,111,116
Fire3 expand	31,429,447	37,912,841	168,941,318	70,518,284,334
Fire4 squeeze	9,412,255	10,925,765	506,673,916	16,686,775,952
Fire4 expand	125,693,405	151,658,685	337,914,780	282,072,796,132
Maxpool4	4,826,624	9,786,983	7,357,889	34,931,485,148
Fire5 squeeze	6,138,728	7,414,158	364,359,198	14,095,990,181
Fire5 expand	31,589,599	36,589,481	82,301,233	68,002,874,365
Fire6 squeeze	9,182,214	11,079,048	364,356,699	20,963,748,645
Fire6 expand	76,831,332	87,993,312	123,540,564	151,818,115,120
Fire7 squeeze	11,454,766	12,686,561	528,766,243	21,306,823,027
Fire7 expand	77,061,539	88,257,291	118,363,536	151,818,219,902
Fire8 squeeze	15,272,443	16,915,131	528,771,352	28,409,285,782
Fire8 expand	136,468,040	156,360,792	164,797,388	270,564,544,208
Maxpool8	2,244,564	4,552,345	1,931,727	16,251,665,899
Fire9 squeeze	6,973,953	8,422,190	310,299,323	16,832,448,970
Fire9 expand	31,761,038	36,386,213	50,853,817	62,634,156,322
Conv 10 & Avgpool	108,948,111	131,548,129	4,848,422,011	263,004,015,235
Sum	-	1,239,580,472	-	2,606,684,280,235

Table 5.3: Detailed active and total energy and output time measurements of each layer of 1.24 GHz at 900 mV when processing one frame.

Area (mm^2)	Latency (s)	Throughput (FPS)	Power (Watt)	Energy per Frame (J/Frame)	Throughput/ Area (FPS/mm^2)	EDP (J*s/$Frame^2$)
1113.79	0.608	1.640	2.804	1.705	0.0015	1.04

Table 5.4: KiloCore performance data when processing each frame 1.74 GHz @ 1100 mV by using Equation 5.8, 5.9, and Figure 5.4, 5.5.

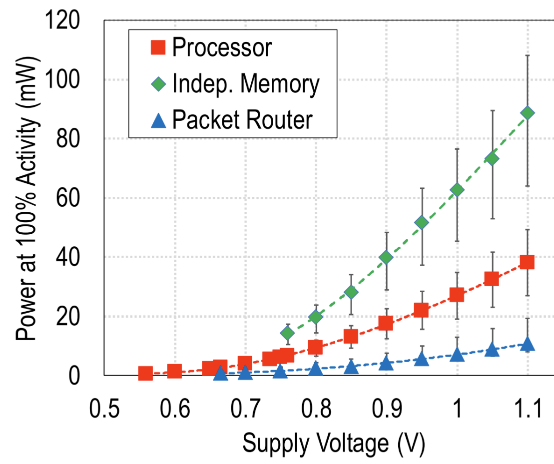


Figure 5.5: Power of a processor, memory, and router when 100% active and operating at the maximum clock frequency at the indicated supply voltage [4].

Chapter 6

SqueezeNet Performance Comparison Between Different Hardware Platforms

6.1 Overview

The metrics to be compared include throughput, throughput per area, energy, energy-delay product (EDP), and memory as mentioned in the previous section. These metrics are chosen to show the high performance and power-efficiency of the KiloCore implementation.

6.2 Comparison of SqueezeNet KiloCore Implementation with Other Platforms

All the measurements used for comparison are scaled to 32 nm technology to match KiloCore's standard, because fabrication technologies used in each platform can significantly affect performance, and power. The scaling method uses predictive polynomial models with calculated coefficients, which produces an accurate scaling factor of CMOS device performance between different technology [5,31]. The scaling factors are calculated using Equation 6.1 and 6.2 with the coefficients provided in Table 6.1, 6.2 from the article introducing scaling techniques proposed by Stillmaker from VCL lab [5].

$$DelayFactor = a_{d3}V^3 + a_{d2}V^2 + a_{d1}V + a_{d0} \tag{6.1}$$

$$EnergyFactor = a_{e2}V^2 + a_{e1}V + a_{e0} \tag{6.2}$$

The scaling factors are calculated using Equation 6.1 and 6.2 with the coefficients provided in Table 6.1, 6.2, and 6.3 proposed in the paper introducing scaling [5].

The scaled data is calculated using Equations 6.3, 6.4, 6.5, and 6.6. For area scaling, the factors are given in Table 6.3.

$$Area_x = AreaFactor_y \times Area_y \tag{6.3}$$

$$Delay_x = \frac{DelayFactor_x}{DelayFactor_y} \times Delay_y \tag{6.4}$$

$$Energy_x = \frac{EnergyFactor_x}{EnergyFactor_y} \times Energy_y \tag{6.5}$$

$$Power_x = \frac{EnergyFactor_x \cdot DelayFactor_y}{EnergyFactor_y \cdot DelayFactor_x} \times Power_y \tag{6.6}$$

Process	a_{d3}	a_{d2}	a_{d1}	a_{d0}	Delay factor
32nm HP @ 0.9V	-1047	2982	-2797	873.5	8.357
20nm HP @ 1.0V	0	34.63	-66.37	41.15	9.41
16nm HP @ 1.0V	0	24.8	-47.52	28.87	6.15
14nm HP @ 1.0V	-40.66	109.2	-100.6	35.92	3.86
10nm HP @ 1.0V	-34.95	93.65	-85.99	30.4	3.11

Table 6.1: The polynomial coefficient values and delay factors calculated with Equation 6.1 [5].

Process	a_{e2}	a_{e1}	a_{e0}	Energy factor
32nm HP @ 0.9V	0.8367	-0.4341	0.1701	0.4114
20nm HP @ 1.0V	0.373	-0.1582	0.04104	0.25584
16nm HP @ 1.0V	0.2958	-0.1241	0.03024	0.20194
14nm HP @ 1.0V	0.2363	-0.09675	0.02239	0.16194
10nm HP @ 1.0V	0.2068	-0.09311	0.02375	0.13744

Table 6.2: The polynomial coefficient values and energy factors calculated with Equation 6.2 [5].

The unscaled data for different hardware platforms is shown in Table 6.4. Energy data is not available for ARMv71 (Raspberry Pi 3) implementation, so we assume 80% of power is consumed during the inference time. The scaled performance comparison is shown in Table 6.5. The scaled clock frequency and scaled throughput value are scaled by the 32 delay factor in Table

Process	Scale factor
32nm	1
20nm	2.2
16nm	2.4
14nm	2.7
10nm	4.5
7nm	7.8

Table 6.3: Factors used for area scaling [5].

	Technology (nm)	Area (mm^2)	Clock Freq (GHz)	Throughput (FPS)	Energy per image (J/Frame)
Xeon E3-1275 v5 [32]	14	122	3.6	16.67	0.75
Core i5-5250U [32]	14	133	2.7	14.28	0.98
KNightsLanding 68 cores [33]	14	N/A	1.4	8.33	4.5
Snapdragon 810 [32]	20	N/A	1.5	2.0	1.3
ARMv8+NVIDIA Pascal [32]	16/14	729	3.0	16.66	0.51
ARMv71 [32]	14	2.2	0.9	0.46	8.28

Table 6.4: Unscaled raw data for different programmable processors.

	Area (mm^2)	Clock Freq (GHz)	Thruput (FPS)	Energy/image (J/Frame)	Power (Watt)	Thruput/area (FPS/mm^2)	Energy×Delay (J*s/$Frame^2$)
Xeon E3-1275 v5	329.4	1.66	**7.70**	1.905	14.7	0.023	0.248
Core i5-5250U	359.1	1.25	6.60	2.477	16.3	0.018	0.376
KNightsLanding 68 cores	N/A	0.65	3.85	11.432	44.0	N/A	2.971
Snapdragon 810	N/A	1.10	1.47	2.648	3.9	N/A	1.799
ARMv8+NVIDIA Pascal	1895.4	1.39	7.70	1.283	9.9	0.004	**0.167**
ARMv71	**5.94**	0.42	0.21	21.035	4.5	**0.036**	99.002
KiloCore 900mV	1113.8	1.24	1.01	**1.240**	**1.2**	0.001	1.233
KiloCore 1.1V	1113.8	**1.78**	1.64	1.705	2.8	0.002	1.040

Table 6.5: A comparison of key parameters for a variety of programmable hardware platforms all scaled from original data shown in Table 6.4 to 32 nm CMOS technology. Numbers in this table are calculated using Equations 6.3, 6.4, 6.5, and 6.6.

6.1 and Equation 6.4. The energy per image value in column five is calculated by Table 6.2 and Equation 6.5. The area value in column two is calculated by Table 6.3 and Equation 6.3. The rest of the data is calculated based on Equation 5.7, 5.8, 5.9.

The ARMv71 is the processor for the device Raspberry Pi 3 which is a microcontroller. The NVIDIA Pascal is a GPU that consists of hundreds of smaller cores which fabrication process is either TSMC 16 nm or Samsung 14 nm [34]. In the scaling, the 14 nm is selected to perform the calculation. All other hardware platforms that listed in the table are CPUs. The way to calculate the improvement is by normalizing the smallest value in the row to 1, which is the large number divided by the smallest number.

6.2.1 Throughput Performance

The normalized throughput result is calculated by dividing the large value by the smallest value in the same metric. The throughput value is low which is caused by the slow speed in layer 1 of the SqueezeNet. The KiloCore implementation does not achieve the highest throughput per area because of the large chip area size. The chip area is still smaller than the ARMv8 and NVIDIA pascal combination.

6.2.2 Energy Dissipation

The KiloCore implementation offers, the lowest energy per image with 1.0× − 17.0× performance advantage when compared with other implementations. The visualized comparison is shown in Figure 6.1. The Raspberry Pi 3 has the biggest amount the energy consumption which is due to the long latency. The ARMv8 and NVIDIA Pascal combination is the second most energy efficient way to process the frames. The result shows the advantage of the multi-core architecture on energy consuming.

	Xeon E3-1275 v5	i5 5250U	KNightsLanding 68 cores	Snapdragon 810	ARMv8+NVIDIA Pascal	ARMv71	KiloCore
Normalized Energy	1.54	2.00	9.23	2.14	1.04	17.00	**1**

Table 6.6: Normalized energy per image. The smallest number is normalized to one based on Table 6.5.

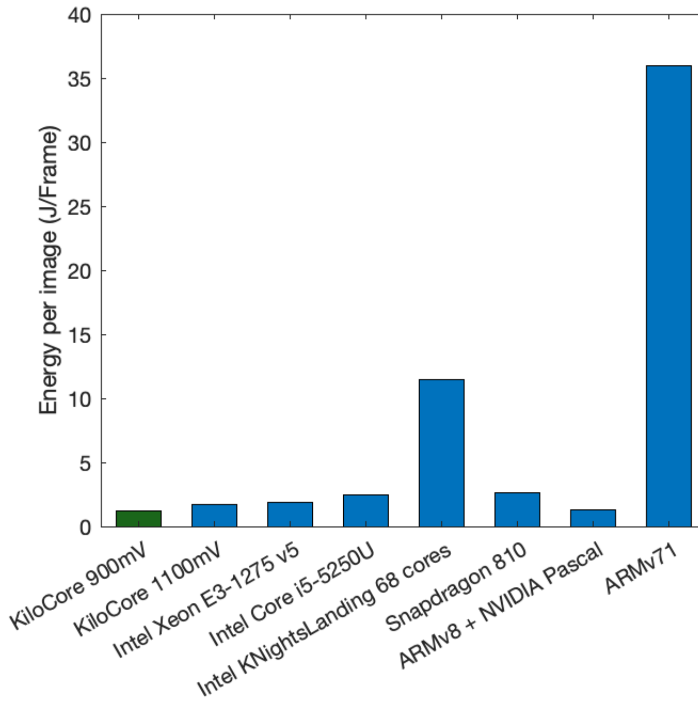


Figure 6.1: Energy comparison between different hardware platforms based on Equation 6.5 and Table 6.5.

6.2.3 Power Dissipation

The power dissipation of the KiloCore many-core processor is 3.1× − 35.3× lower than other processors, as shown in Figure 6.2. The low power and low energy dissipation are benefited from the low power design of the KiloCore chip. The Intel Knights Landing has the highest power dissipation due to the high energy dissipation and the low latency. The results show that KiloCore is suitable for low-power use cases.

6.2.4 Energy × Delay

The Energy-Delay Product is a combination of energy consumption and delay together, showing a processor's energy-efficient. The EDP can be large for a low-power processor if the professor has low throughput to reduce the energy consumption. The KiloCore has a higher EDP value than GPU, because of the low energy and low throughput. The EDP value for KiloCore implementation is in the middle range among other implementations, and the EDP is 95.2× lower compared to the Raspberry Pi 3 implementation.

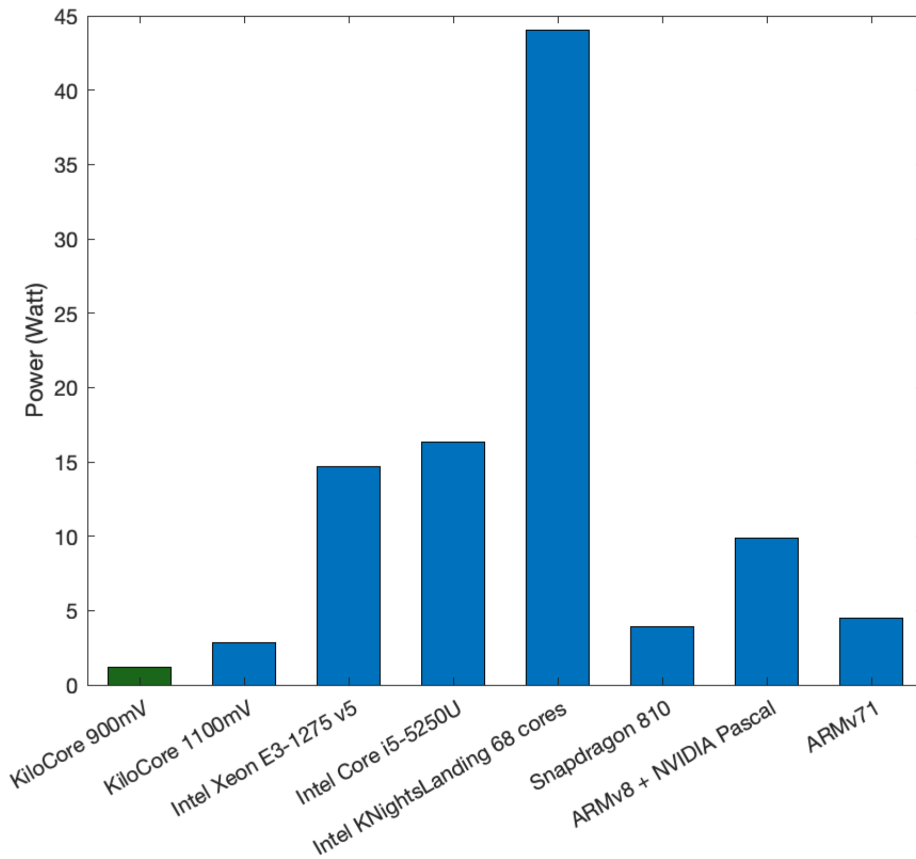


Figure 6.2: Power comparison between different hardware platforms by Table 6.5 and Equation 5.7.

6.2.5 Memory Requirements

For memory requirements, the visualized bar diagram is shown in Figure 6.3 with the values labeled in Figure. The memory usage of this work is 10.3× − 65.3× smaller than other platforms. The low memory requirements of multi-core implementations save energy from large memory chips on other platforms. The other platforms choose to save whole pictures on memory, occupying more memory spaces.

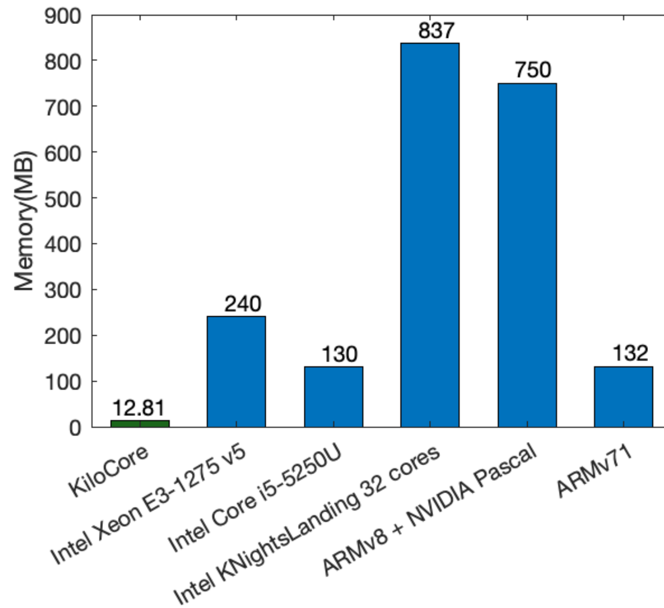


Figure 6.3: Memory comparison between different hardware platforms based on Table 6.5

6.3 Summary

As the comparison in Table 6.5 presents, the SqueezeNet implementation achieves the highest performance in most items compared with other hardware platforms when all chips are scaled to the same manufacture technology.

The high performance metrics compared to other hardware platforms include energy, power, throughput, energy-delay product (EDP), and memory. Regarding energy per image and power, our design achieves a 1.0× − 17.0× lower energy and 3.1× − 35.3× lower power consumption. Regarding throughput performance, the KiloCore implementation is 4.8x higher than ARMv71. The EDP value for KiloCore implementation is in the middle range among other implementations, and the

EDP is 95.2× lower compared to the Raspberry Pi implementation. SqueezeNet implementation on KiloCore uses significantly less memory compared to the other programmable processors. The high frequency and high voltage supply provides a high performance on KiloCore. KiloCore uses significantly less memory compared to the other programmable processors. The GPU has the smallest latency but a larger die area and higher energy consumption. The result of the GPU shows the benefits of the multi-core architecture and the disadvantage of the GPU too.

Chapter 7

Thesis Summary and Future Work

7.1 Thesis Summary

This thesis presents a high-throughput, memory-efficient, and energy-efficient SqueezeNet inference implementation on the KiloCore many-core platform.

Chapter 1 starts with the motivation of the SqueezeNet inference. Chapter 2 outlines the evolutionary history of the SqueezeNet architecture. The basic CNN concept and the component utilized to construct SqueezeNet are briefly discussed. Chapter 3 introduces the KiloCore many-core platform architecture, including the chip implementation, processor architecture, on-chip memory hierarchy, and programming software. Chapter 4 discusses the implementations of SqueezeNet with details of the algorithms, including convolution, pooling, and other layers. Chapter 5 illustrates the simulation results of the SqueezeNet implementation on the KiloCore platform and Chapter 6 compares the power and efficiency comparisons among KiloCore implementation, general-purpose processors, GPUs, and FPGAs.

7.2 Future Work

7.2.1 Increase the Number of SRAMs

The bandwidth of the KiloCore used for this inference is primarily limited by the amount of on-chip SRAMs. There are 14 SRAMs on the KiloCore platform now. If the total number of SRAMs is 16, that is a ratio of 2, 4, and 8. It is much more suitable for convolutional neural networks because each layer's input data can be evenly saved into SRAMs, which will reduce the

amount of work on the channel distributor stage for this design strategy. Two extra SRAMs can also increase the on-chip data storage capacity of the KiloCore, which allows more applications to be inferences on KiloCore.

7.2.2 More Input Ports per Core

In this design, the adder stage and the output buffer stage occupy almost the same amount of cores as the convolution cores. This is mainly due to the limit on the number of input ports per core on the KiloCore. If each core has four input ports, the number of cores in the adder stage can be cut in half. This improvement leaves more cores to compute convolutions and improves overall speed and throughput.

7.2.3 Saturating Addition

The overflow shown is handled via the bit extension as presented in Chapter 5 on page 36. The other way to achieve the same result is via the saturating addition. The saturation addition limits the output result between the minimum and maximum possible value, which is more efficient than the overflow checker in this design. The adder stage is not the bottleneck of my design's output speed, but changing to Saturating addition will lower the energy consumption of the adder stage.

7.2.4 Reduce the Number of Layers in SqueezeNet

The application direction of SqueezeNet is the embedded system devices, and the main force of the embedded environment is real-time applications. Although SqueezeNet can reduce the parameters of the network by replacing the number of parameters with a deeper depth, it loses the parallel ability of the network, and increases the processing time, which is contradictory to the primary objective. By lowering the number of layers, the SqueezeNet may become more efficient and reduce the overall processing time on a variety of systems.

Glossary

ASIC Application-Specific Integrated Circuit. Customized integrated circuits for one particular application.

Clang Compiler front end for the C, C++, Objective-C and Objective-C++ programming languages.

CMOS Complementary Metal Oxide Semiconductor (CMOS) is a type of MOSFET (Metal Oxide Semiconductor Field Effect Transistor) semiconductor device fabrication process used for constructing integrated circuits (ICs).

CNN Convolutional Neural Networks, one of the Deep Learning algorithms that can extract useful information from multi-dimensional input matrix, typically images.

DRAM Dynamic random-access memory. A type of random-access semiconductor memory that stores each bit of data in a memory cell, usually consisting of a tiny capacitor and a transistor, both typically based on metal-oxide-semiconductor technology.

EDP Energy-delay product. The product of the total energy consumed and the inference delay (latency).

Expand The expand layer is a one depth convolution layer that has a mix of 1×1 and 3×3 filters.

Feature map A term used to describe the 3D dataset flow through the neural networks, which is used as input/output of each layer.

Fire A fire module is a building block for SqueezeNet. A fire module is comprised of a squeeze convolution layer, which has only 1×1 filters, feeding into an expand layer that has a mix of 1×1 and 3×3 convolution filters.

FPS Frames per Second. A measurement of how many images can be processed for every second.

KiloCore The 3rd generation of *Asynchronous Array of simple Processors* design inspired by the AsAP platform.

MATLAB MATLAB is a proprietary multi-paradigm programming language and numeric computing environment developed by MathWorks. MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, and creation of user interfaces.

Squeeze The squeeze layer is a one depth convolution layer that has 1×1 filters.

SRAM Static Random Access Memory. A type of volatile memory that is typically used as the first-level cache to the processor. Extremely fast but expensive.

Bibliography

[1] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *ArXiv*, abs/1602.07360, 2016.

[2] Brent Bohnenstiehl. The kilocore 2 architecture and chip design. In preparation.

[3] Aaron Stillmaker, Brent Bohnenstiehl, and Bevan Baas. The design of the kilocore chip. In *ACM/IEEE Design Automation Conference*, Austin, TX, Jun. 2017.

[4] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. Kilocore: A 32 nm 1000-processor array. In *IEEE HotChips Symposium on High-Performance Chips*, August 2016.

[5] A. Stillmaker and B. Baas. Scaling equations for the accurate prediction of cmos device performance from 180 nm to 7 nm. *Integration, the VLSI Journal*, 58:74–81, 2017.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[8] Forrest N. Iandola, Khalid Ashraf, Matthew W. Moskewicz, and Kurt Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters, 2015.

[9] Nelson Yalta, Shinji Watanabe, Takaaki Hori, Kazuhiro Nakadai, and Tetsuya Ogata. Cnn-based multichannel end-to-end speech recognition for everyday home environments, 2018.

[10] Marco Parola, Alice Nannini, and Stefano Poleggi. Web image search engine based on lsh index and cnn resnet50, 2021.

[11] Rafat Jamal Tazim, Md. Messal Monem Miah, Sanzida Sayedul Surma, Mohammad Tariqul Islam, Celia Shahnaz, and Shaikh Anowarul Fattah. Biometric authentication using cnn features of dorsal vein pattern extracted from nir image. In *TENCON 2018 - 2018 IEEE Region 10 Conference*, pages 1923–1927, 2018.

[12] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

[13] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017.

[14] Ruofei Hu, Binren Tian, Shouyi Yin, and Shaojun Wei. Efficient hardware architecture of softmax layer in deep neural network. In *2018 IEEE 23rd International Conference on Digital Signal Processing (DSP)*, pages 1–5, 2018.

[15] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2015.

[16] Bevan Baas, Zhiyi Yu, Michael Meeuwsen, Omar Sattari, Ryan Apperson, Eric Work, Jeremy Webb, Michael Lai, Tinoosh Mohsenin, Dean Truong, and Jason Cheung. AsAP: A fine-grained many-core platform for dsp applications. *IEEE Micro*, 27(2):34–45, March 2007.

[17] Zhiyi Yu, Michael Meeuwsen, Ryan Apperson, Omar Sattari, Michael Lai, Jeremy Webb, Eric Work, Tinoosh Mohsenin, Mandeep Singh, and Bevan M. Baas. An asynchronous array of simple processors for dsp applications. In *IEEE International Solid-State Circuits Conference, (ISSCC '06)*, pages 428–429, February 2006.

[18] Zhiyi Yu, M.J. Meeuwsen, R.W. Apperson, O. Sattari, M. Lai, J.W. Webb, E.W. Work, D. Truong, T. Mohsenin, and B.M. Baas. AsAP: An asynchronous array of simple processors. *Solid-State Circuits, IEEE Journal of*, 43(3):695–705, Mar. 2008.

[19] D. Truong, W. Cheng, T. Mohsenin, Zhiyi Yu, T. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, P. Mejia, Anh Tran, J. Webb, E. Work, Zhibin Xiao, and B. Baas. A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling. In *VLSI Circuits, 2008 IEEE Symposium on*, June 2008.

[20] Bevan M. Baas. A parallel programmable energy-efficient architecture for computationally-intensive DSP systems. In *Signals, Systems and Computers, 2003. Conference Record of the Thirty-Seventh Asilomar Conference on*, November 2003.

[21] Dean Truong, Wayne Cheng, Tinoosh Mohsenin, Zhiyi Yu Toney Jacobson, Gouri Landge, Michael Meeuwsen, Christine Watnik, Paul Mejia, Anh Tran, Jeremy Webb, Eric Work, Zhibin Xiao, and Bevan Baas. A 167-processor computational array for highly-efficient dsp and embedded application processing. In *IEEE HotChips Symposium on High-Performance Chips (HotChips 2008)*, Aug. 2008.

[22] B. Bohnenstiehl, A. Stillmaker, J. J. Pimentel, T. Andreas, B. Liu, A. T. Tran, E. Adeagbo, and B. M. Baas. Kilocore: A 32-nm 1000-processor computational array. *IEEE Journal of Solid-State Circuits*, 52(4):891–902, 2017.

[23] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. A 5.8 pJ/Op 115 billion Ops/sec, to 1.78 trillion Ops/sec 32 nm 1000-processor array. In *Symposium on VLSI Circuits*, June 2016.

[24] Z. Yu and B. Baas. Implementing tile-based chip multiprocessors with gals clocking styles. In *2006 International Conference on Computer Design*, pages 174–179, 2006.

[25] A.T. Tran, D.N. Truong, and B.M. Baas. A GALS many-core heterogeneous DSP platform with source-synchronous on-chip interconnection network. In *Networks-on-Chip, 2009. NoCS 2009. 3rd ACM/IEEE International Symposium on*, pages 214–223, May. 2009.

[26] Zhiyi Yu and Bevan M. Baas. A low-area multi-link interconnect architecture for GALS chip multiprocessors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 18(5):750–762, May 2010.

[27] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. Kilocore: A fine-grained 1,000-processor array for task-parallel applications. *IEEE Micro*, 37(2):63–69, 2017.

[28] Satyabrata Sarangi and Bevan M. Baas. DeepScaleTool: A tool for the accurate estimation of technology scaling in the deep-submicron era. In *IEEE International Symposium on Circuits & Systems*, May 2021.

[29] Bevan Baas, Zhiyi Yu, Michael Meeuwsen, Omar Sattari, Ryan Apperson, Eric Work, Jeremy Webb, Michael Lai, Daniel Gurman, Chi Chen, Jason Cheung, Dean Truong, and Tinoosh Mohsenin. Hardware and applications of AsAP: An asynchronous array of simple processors. In *IEEE HotChips Symposium on High-Performance Chips (HotChips 2006)*, Aug. 2006.

[30] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. Design space exploration of fpga-based deep convolutional neural networks. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 575–580, 2016.

[31] Aaron Stillmaker, Zhibin Xiao, and Bevan Baas. Toward more accurate scaling estimates of cmos circuits from 180 nm to 22 nm. Technical Report ECE-VCL-2011-4, VLSI Computation Lab, ECE Department, University of California, Davis, December 2011.

[32] Xingzhou Zhang, Yifan Wang, and Weisong Shi. pCAMP: Performance comparison of machine learning packages on the edges. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, Boston, MA, July 2018. USENIX Association.

[33] Amir Gholami, Kiseok Kwon, Bichen Wu, Zizheng Tai, Xiangyu Yue, Peter Jin, Sicheng Zhao, and Kurt Keutzer. Squeezenext: Hardware-aware neural network design. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 1719–171909, 2018.

[34] Hassan Mujtaba. Tsmc receives next-gen nvidia 7nm gpu orders, more than 50 chip tape outs expected by the end of 2018, Jun 2018.