A Memory-Efficient YOLO Object Detection Convolutional Neural Network Inference Engine On The KiloCore 2 Manycore Platform

By

YIKAI MAO
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:

Chair, Bevan M. Baas

Member, Soheil Ghiasi

Member, Hussain Al-Asaad

Committee in charge
2021

Abstract

Object Detection is one of the most resource-intensive tasks for Convolutional Neural Networks (CNN). To predict the category of the objects and at the same time determine their location, the Object Detection network has to use a very deep structure, typically 10 to 50 layers, along with a huge number of learnable parameters ranging from a few million to over a billion. Many architectures have been implemented on various hardware platforms to accelerate the inference speed of Object Detection. For example, the cuDNN library on Nvidia GPUs and the Intel DLIA framework on x86 CPUs. However, they either consume a lot of power or require large memory to perform the acceleration algorithm, making them unsuitable for edge-computing use cases where power is limited and energy must be conserved.

This thesis presents an efficient and high-throughput network inference implementation for the YOLOv3-Tiny Object Detection system on the KiloCore 2 manycore platform. YOLOv3-Tiny is a light-weight and accurate Object Detection network with only 13 Convolution layers and 8,861,918 learnable parameters, and the KiloCore 2 platform is a low-power manycore processor chip with 697 programmable cores and a high-speed on-chip communication network.

Specifically, this thesis presents two software optimization techniques to relax the memory requirements of YOLOv3-Tiny, and a scalable hardware architecture for calculating convolutions. On the software side, low-precision quantization reduces all the parameters from 16-bits to 8-bits while still maintaining 90% of the accuracy, and Batch Normalization (BN) Folding is used to compress the computation complexity of the network, removing all the BN layers together with 12,736 parameters. This thesis describes a standardized process of applying quantization and BN Folding so that these optimization techniques can be implemented on any CNN. On the hardware side, a scalable and modular architecture to calculate convolution utilizing a maximum of 536 cores on KiloCore 2 is presented, achieving a high throughput per chip area of 1.002 frames per second/cm^2 and offers a low energy consumption of 2.232 J/image.

Compared with other hardware platforms such as general-purpose CPUs and specialized GPU accelerators, this implementation achieves a less than 5% reduction in throughput per chip area, but offers 9.17× to 441× greater throughput per watt. Furthermore, to run the full YOLOv3-Tiny network, this implementation requires only 17.72 MB of off-chip memory for parameters and 896 KB of on-chip memory, which provides a 49.5× to 64.8× memory reduction compared with the GPU implementations.

Acknowledgments

2020 was a very difficult year for everyone, but I consider myself to be extremely lucky because I have met so many great people who have supported me throughout my graduate studies at UC Davis.

I would like to thank Dr. Bevan Baas. Thank you for giving me the opportunity to join the VCL, this research project has been a delightful experience under your guidance.

Also, I want to thank Dr. Soheil Ghiasi and Dr. Hussain Al-Asaad. Thank you for your time and effort in reviewing my thesis.

Everybody in VCL was so friendly and helpful. Thank you to Sharmila and Brent for helping me with the Project Manager and KiloCore 2 hardware, and thank Shifu for giving me many pieces of advice for my research project.

I'm grateful to meet Haotian, Ziyuan, and Weitai in Davis. Thank you all for your support.

Finally, I want to thank my family. This thesis would be impossible to finish without their love.

Contents

Abstract ii

Acknowledgments iii

List of Figures vi

List of Tables ix

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Thesis Structure . . . 3

2 Background - YOLOv3-Tiny 4
- 2.1 Overview . . . 4
- 2.2 YOLOv3-Tiny Object Detection System . . . 4
- 2.3 Convolutional Neural Network of YOLOv3-Tiny . . . 6
  - 2.3.1 Convolution Layer of YOLOv3-Tiny . . . 6
  - 2.3.2 Activation Layer of YOLOv3-Tiny . . . 7
  - 2.3.3 Maxpool Layer of YOLOv3-Tiny . . . 7

3 Background - KiloCore 2 Platform 9
- 3.1 Overview . . . 9
- 3.2 Processors . . . 10
- 3.3 On-Chip SRAM . . . 11
- 3.4 Programming . . . 12

4 YOLOv3-Tiny CNN Optimization for KiloCore 2 14
- 4.1 Overview . . . 14
- 4.2 Batch Normalization Folding . . . 14
- 4.3 16-bit Fixed-Point Quantization . . . 16
- 4.4 Final CNN Structure . . . 19

5 YOLOv3-Tiny Architecture on KiloCore 2 23
- 5.1 Overview . . . 23
  - 5.1.1 Overview - System Level . . . 24
  - 5.1.2 Overview - Chip Level . . . 24
  - 5.1.3 Overview - Core Level . . . 24
- 5.2 System Level Operarion . . . 25

5.3 Chip Level Architectures . . . 26
5.3.1 Data Distribution Stage . . . 27
5.3.2 Computation Stage . . . 28
5.3.3 Output Buffering Stage . . . 29
5.3.4 Chip Level Summary . . . 30
5.4 Core Level Implementation . . . 31
5.4.1 im2col . . . 32
5.4.2 Image Distributor (Weight-stationary Architecture) . . . 33
5.4.3 Image Distributor (Input-stationary Architecture) . . . 34
5.4.4 Weights Distributor (Weight-stationary Architecture) . . . 37
5.4.5 Weights Distributor (Input-stationary Architecture) . . . 38
5.4.6 2D Convolution Module (Inside 3D Convolution Module) . . . 39
5.4.7 3D Convolution Module . . . 41
5.4.8 Output Buffer . . . 42
5.4.9 Core Level Summary . . . 45

6 Inference Evaluation **50**
6.1 Overview . . . 50
6.2 Inference Result . . . 51
6.3 Simulation Measurements . . . 54

7 Performance Comparison with Other Hardware Platforms **58**
7.1 Overview . . . 58
7.2 Comparison of YOLOv3-Tiny Manycore Implementation with Other Platforms . . . 58
7.2.1 Performance . . . 61
7.2.2 Power/Energy-Efficiency . . . 62
7.2.3 Memory-Efficiency . . . 64
7.3 Summary . . . 65

8 Future Work **66**
8.1 8-bit Quantization . . . 66
8.2 Alternative Mappings of YOLOv3-Tiny . . . 67
8.3 Quantization Aware Training . . . 68

9 Thesis Summary **69**

Glossary **70**

Bibliography **73**

List of Figures

1.1 Example outputs of the YOLOv3 Object Detection System, the predicted category is accompanied with a confidence score. The maximum possible score is 1.00. 2

2.1 YOLO Object Detection System workflow. The input image is divided into grid blocks (step a) and multiple predictions will be made in relative to the center of the grid blocks (step b), predictions in the same catogories are grouped together (step c) and the low-scoring detections are filtered out to form the final output. [1]. 5
2.2 YOLOv3-Tiny Convolutional Neural Network structure, each dot is a layer and the arrows indicate residual connection. Image produced by MATLAB Deep Learning Toolbox. 5
2.3 Example of 3×3 convolution and 1×1 convolution. A 3×3 convolution operates on the 2D plane to extract spatial information, whereas a 1×1 convolution compresses the depth of the input matrix in the 3D space. 6
2.4 Plot of the Leaky ReLU activation function. 7
2.5 Example of 2×2 maxpool with stride = 2, the image size is reduced by half. 8

3.1 Overview of the KiloCore 2 chip [2] [3]. 9
3.2 A single standard processor tile of the KiloCore 2 chip [2]. 10
3.3 A single high-speed processor tile of the KiloCore 2 chip [2]. 11
3.4 On-chip SRAM module [2]. 12
3.5 Project Manager GUI [2]. 13

4.1 Batch Normalization Algorithm. [4]. 14
4.2 Visualization of BN Folding. Left side is a typical CNN without BN Folding, so after each convolution layer there is a Batch Normalization layer with additional parameters that must be streamed into the processor. Right side is a CNN with BN Folding, all the BN parameters are "folded" into the weights and biases so there is no need to use Batch Normalization layer during inference. 15
4.3 Distribution of all the weights for the YOLOv3-Tiny network. Only 5.5% of weights are below precision using a Q7.9 fixed-point quantization. Most of the weights are inside the range between 2^{-3} and 2^{-7}. 17
4.4 Distribution of all the biases for the YOLOv3-Tiny network. Only 6 (0.2%) biases are below precision using a Q7.9 fixed-point quantization. Most of the biases are inside the range between 2^{2} and 2^{-2}. 18
4.5 Max/Min trend of the weights and biases. The dynamic range shrinks significantly as the layer gets deeper. 18

4.6 Final YOLOv3-Tiny CNN architecture after optimization. Each dot is a layer and the arrows indicate residual connection. Notice that all the Batch Normalization layers are removed comparing with the original CNN structure in Chapter 2 section 2 (Figure 2.2). 19

5.1 3 level of the implementation. When designing the architecture, programs are written for the cores, then the cores are connected together on the chip level, taking inputs from the test system (Direction of Design). When the implementation is in operation, data is saved on the system level and streamed into the chip, then processed inside the cores (Direction of Operation). 23
5.2 System level operation. The full set of weights/biases and the input image are stored in the off-chip storage, and they are streamed layer by layer into KiloCore 2. For each iteration KiloCore 2 processes conv/activation/maxpool in one shot and outputs the data back to the off-chip storage. 25
5.3 Operation timing diagram. Green means KiloCore 2 is processing that layer, and red means KiloCore 2 is switching architectures, which is discussed in section 5.3. Red arrow indicates data flow. 26
5.4 Left: Weight-stationary Architecture. Right: Input-stationary Architecture. Red path highlights the data stream. 28
5.5 High level dataflow of the computation stage. Data are streamed into the 2D and 3D Convolution modules, the convolution outputs might be incomplete due to the layer being too deep so the output is connected to the Output Buffering Stage for post-processing. 28
5.6 High level dataflow of the output buffering stage. Notice that SRAM has bi-directional communication with the adder. The final output is either after activation or maxpool, depending on that layer's configuration. 29
5.7 Difference showing the active and idle region of KiloCore 2 when processing a layer. When new data is being streamed into the chip, the computation and output buffering stages are in idle state. When the chip has all the data for the current layer, it starts processing and the chip is fully utilized. 30
5.8 Chip level parallel operation. This Figure gives an overview of one full KiloCore 2 chip. Notice that each of the 3D convolution modules in the computation stage contains multiple 2D convolution modules inside, depending on the input layer depth. 31
5.9 im2col algorithm demo showing in the bottom. The multi-dimension convolution is flattened in to a single matrix multiplication. 32
5.10 Image distributor, weight-stationary architecture. 33
5.11 Image distributor, input-stationary architecture. The left side distributes all the even indexes, showing in blue. The right side distributes all the odd indexes, showing in orange. To distribute all 64 channels, the distributor needs to loop 4 times, outputting 16 channels at a time. 35
5.12 Weights distributor, weight-stationary architecture. Four distribution cores are used because four 3D convolution modules are processing in parallel. SRAM is saving the full weights/biases for the current layer. 38
5.13 Weights distributor, input-stationary architecture. SRAM is buffering four weights/biases at a time, then distribute them in parallel to the 2D convolution modules. 39
5.14 2D convolution module. Example dataflow is showing on the right. 40
5.15 3D convolution module. Each 2D conv module inside can process one image channel. So if there are 16 2D conv modules inside one 3D conv module ($n = 16$), then the 3D conv module can process 16 channels in one shot. 42

5.16 3D deep convolution without output buffer. Layer 8 has 1,024 channels, but using 1,024 2D conv modules in one 3D conv module will cost at least 6,144 cores, which is impossible to place on 1 KiloCore 2 chip. 42
5.17 3D deep convolution with output buffer. The 3D conv module can process 16 channels at a time, and the partial 3D conv output is saved inside the output buffer. 43
5.18 Fly line diagram of the weight-stationary architecture, each dot is a processor core, and the green lines indicate inter-core connection. 47
5.19 The complete error-free weight-stationary architecture mapping to the manycore processor array. The various link colors signify different types of communication links: nearest neighbor, long distance, or packet network. 48
5.20 Fly line diagram of the input-stationary architecture, each dot is a processor core, and the green lines indicate inter-core connection. 48
5.21 The complete error-free input-stationary architecture mapping to the manycore processor array. The various link colors signify different types of communication links: nearest neighbor, long distance, or packet network. 49

6.1 Test image used in this section with three detections expected: dog, person, and horse. 50
6.2 Mean and max error plot for each layer. 51
6.3 Output image of each layer. Red arrow indicates data flow for generating the coarse prediction in 13×13, Blue arrow indicates data flow for generating the fine prediction in 26×26 53
6.4 Final detection output. Left side is the result generated by KiloCore 2, right side is the Golden Reference output produced by the original YOLOv3-Tiny system. 54

7.1 Throughput per area comparison. Memory area for the GPUs and CPUs is not included in this data because it is publically unavailable, however memory area is included for the KiloCore 2 implementation. 61
7.2 Throughput per watt comparison. Core i3-8145U not included due to value too small. 62
7.3 EDP comparison. Core i3-8145U not included due to value too large. 63
7.4 Memory requirement comparison. 65

8.1 8-bit weights of layer 8, the SQNR is 12.7% with all the bits used as fraction bits. . 66
8.2 Bigger chip configuration using n output ports. 67
8.3 Pipelined connection of 13 KiloCore 2 chips. The critical path along with the two slowest layers are marked in red. 68

List of Tables

4.1 Detail of the YOLOv3-Tiny Concolutional Neural Network 20
4.2 Detail of the YOLOv3-Tiny Concolutional Neural Network, Cont'd. 21
4.3 Detail of the YOLOv3-Tiny Concolutional Neural Network, Cont'd. 22

5.1 Detailed information showing the data size of each layer. Green means less than 896KB, red means larger than 896KB (larger than the capacity of the on-chip SRAM). 27
5.2 Summary of the code structure for all the code level blocks 46
5.3 Core counts for all the basic modules. n equals to the number of 2D conv modules inside one 3D conv module. 47

6.1 Mean and max error for each layer compared to the Golden Reference 51
6.2 Performance data of this work. 56
6.3 Detailed simulation measurements of each layer. 57

7.1 The polynomial coefficient values and delay factors calculated with equation 7.1. [5] 59
7.2 The polynomial coefficient values and energy factors calculated with equation 7.2. [5] 59
7.3 Factors used for area scaling [5]. 60
7.4 Performance data for all hardware platforms [6] [7] [8] [9] [10]. CPU Power is assumed to be one half of the TDP. Memory area for the GPUs and CPUs is not included in this data because it is publically unavailable, however memory area is included for the KiloCore 2 implementation.
*No public data available 60
7.5 Throughput per area comparison. Data taken from table 7.4. Memory area for the GPUs and CPUs is not included in this data because it is publically unavailable, however memory area is included for the KiloCore 2 implementation. 61
7.6 Throughput per watt and EDP comparison. Data taken from table 7.4 62
7.7 Memory requirement comparison. Data taken from table 7.4, [8], and [11].
*DNR = did not run, caused by limited memory capacity, unsupported network layers, or hardware/software limitations [11]. 64

Chapter 1

Introduction

1.1 Motivation

Convolutional neural networks (CNN) have become one of the most commonly used Machine Learning (ML) algorithms for processing image-related tasks. Following the breakthrough development of AlexNet [12], numerous advanced CNN architectures have been developed that can achieve near-human performance. For example, the Inception V3 model with 78.8% accuracy [13] and the FixResNeXt-101 model with 86.4% accuracy [14], all performing image classification on the ImageNet [15] dataset.

After successfully applying CNN to Image Classification tasks, Object Detection becomes the next challenge for computer scientists and hardware engineers. For Object Detection, not only the CNN has to tell the category of the object, but it also has to predict where the object is, along with a bounding box showing the predicted area of that object. One of the most popular and high-performance Object Detection Systems is YOLO (You Only Look Once), which uses a deep CNN feature extractor to predict both the objects and their location in one single network evaluation. Although the YOLO model shows a slightly lower accuracy score for the ImageNet dataset at 78.5%, it is incredibly lightweight and fast, making it suitable for running on low-power hardware.

Because CNN training for Object Detection requires massive storage for the dataset, it is usually executed by high-performance and specialized GPUs and ASICs, like Google's Tensor Processing Unit (TPU) [16]. However, when deploying the trained Object Detection System to the

Figure 1.1: Example outputs of the YOLOv3 Object Detection System, the predicted category is accompanied with a confidence score. The maximum possible score is 1.00.

field, its inference speed is often limited by the processing capability, system memory, or power ratings of the real-world hardware. To accelerate the YOLO system, and to optimize CNN inference in general, various designs have been implemented on GPU [8], general-purpose CPU [17] and FPGAs [18]. GPUs can offer very low latency, but they are associated with a high power rating and a large memory area. CPUs consume less power but the throughput is limited by their parallel processing capability. For example, the YOLOv3-Tiny implementation by Han et al. [8] using Nvidia's Jetson GPUs require memory more than 1GB and consumes a maximum power of 9.6W, while the same algorithm running on an Intel CPU(i5-8365UE) shows a performance loss around 50% in terms of Frames per Second. FPGAs can achieve high throughput by having a custom design, but their performance is heavily bottlenecked by the memory bandwidth and capacity. The YOLO inference architecture made by Yu and Bouganis [18] using Xilinx XC7Z020 shows that when memory is limited to 512MB DDR3, the latency can be 10× slower compared to Zynq7035 with double the memory space.

This thesis presents a power-efficient and memory-efficient YOLO inference architecture on the KiloCore 2 manycore platform that can offer high-performance, and occupies minimal silicon chip area. The manycore implementation utilizes a modular design that breaks down the CNN computation into small and simple tasks for parallel acceleration, as well as reducing the memory requirement by spreading out the large matrix calculations to each core. The implementation is also scalable, which makes it easily adapts to different manycore configurations and other CNN architectures.

1.2 Thesis Structure

The thesis is organized as follows:

- Chapter 2: Software background about the YOLO Object Detection System. Convolutional Neural Network structure, and the YOLOv3-Tiny architecture.
- Chapter 3: Hardware background about the KiloCore 2 manycore platform. Hardware implementation, processor architecture, and memory architecture.
- Chapter 4: Optimization techniques used to pre-process the YOLOv3-Tiny CNN on the software level. Batch Normalization Folding, 16-bit Fixed-Point Quantization.
- Chapter 5: The implementation of YOLOv3-Tiny on the hardware level. Modular components, algorithms, and implementation notes.
- Chapter 6: Simulation result and measurements.
- Chapter 7: Compare the hardware performance with other implementations.
- Chapter 8: Future work. Alternative mappings of the architecture, other possible optimizations.
- Chapter 9: Summary of the thesis.

Chapter 2

Background - YOLOv3-Tiny

2.1 Overview

In addition to image classification tasks where an image is categorized with a confidence score, object detection systems can identify and predict a region where the object is located. Traditional object detection systems apply image classifiers at different locations on different scales, then regions with a high confidence score are selected as detections [19]. This architecture can achieve high accuracy but the workflow is very inefficient, because the same image must be processed multiple times in order to find every detectable object. To attack the Object Detection problem from a different angle, the YOLO [1] Object Detection System was invented.

2.2 YOLOv3-Tiny Object Detection System

YOLO (you only look once) is designed with a completely new architecture. The input image is processed only once, and the final output of the Convolutional Neural Network is a 3D tensor containing all the predictions indexed by virtually divided grid blocks [1].

Figure 2.1 shows the workflow of the YOLO Object Detection System. The Neural Network is applied to the full image, this divides the image into blocks. If the center of a detectable object falls into one block, then that block is in charge of predicting the object. Each block will predict multiple bounding boxes and objectiveness scores, and the low-scoring predictions are filtered out before outputting the final detection result. This design eliminates the need for Fully-connected layer and Softmax layer, which makes YOLO extremely fast and efficient.

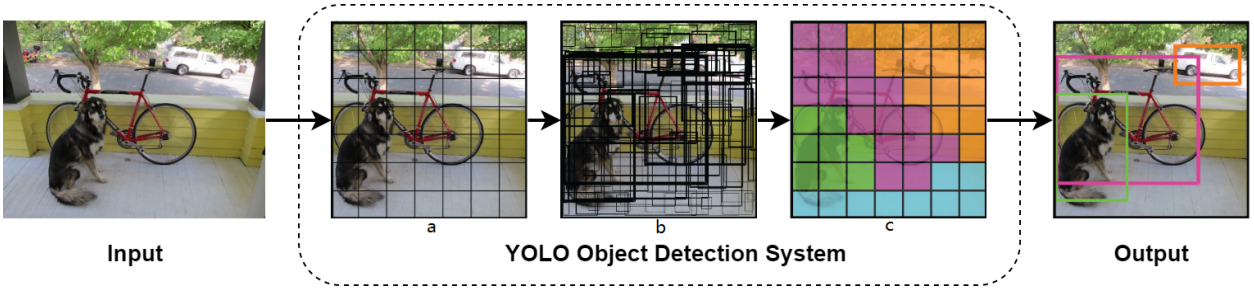

Figure 2.1: YOLO Object Detection System workflow. The input image is divided into grid blocks (step a) and multiple predictions will be made in relative to the center of the grid blocks (step b), predictions in the same catogories are grouped together (step c) and the low-scoring detections are filtered out to form the final output. [1].

YOLOv3 is the third iteration of the YOLO detection system. It's faster, more accurate, and lightweight compared with YOLOv1 and YOLOv2. The Tiny version of YOLOv3 is a small model designed for constrained environments. With only 8,861,918 learnable parameters and 13 convolutional layers, YOLOv3-Tiny can achieve an impressive 33.1 mAP (mean Average Precision) score on the COCO dataset [19].

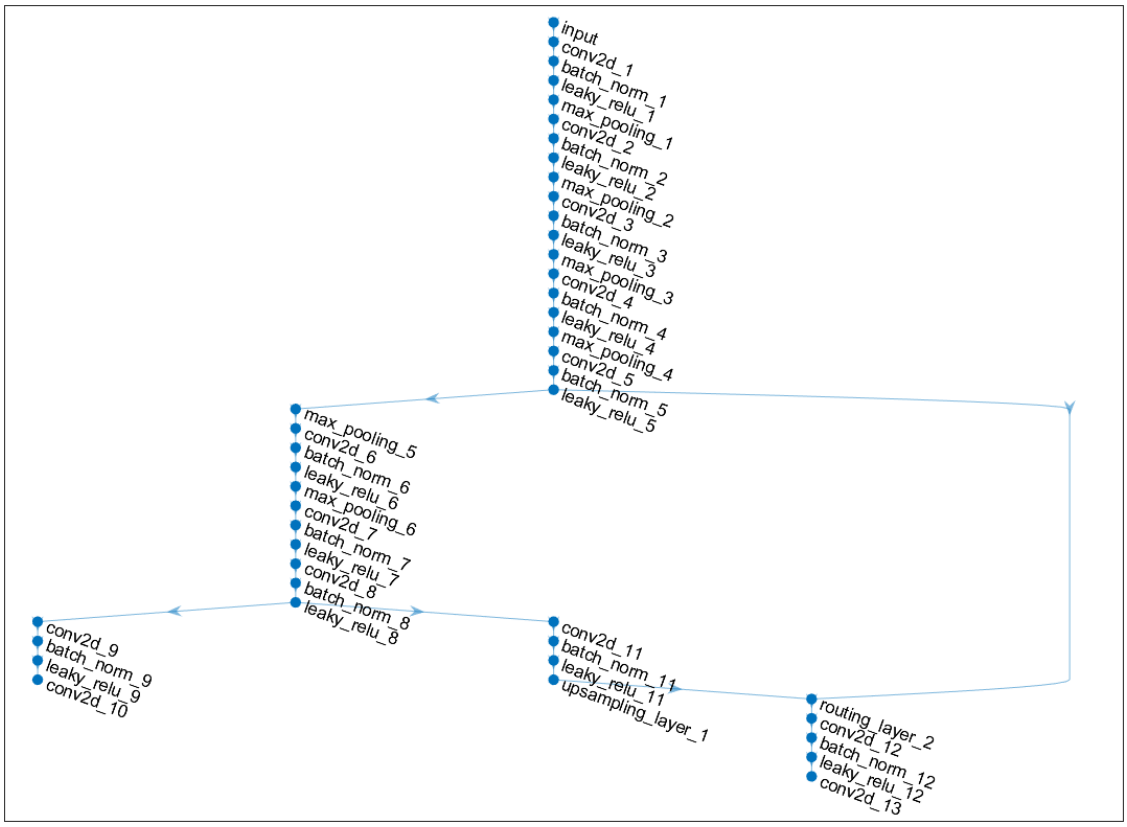

Figure 2.2: YOLOv3-Tiny Convolutional Neural Network structure, each dot is a layer and the arrows indicate residual connection. Image produced by MATLAB Deep Learning Toolbox.

2.3 Convolutional Neural Network of YOLOv3-Tiny

The basic building block of a Convolutional neural network (CNN) is a three-layer trio: convolution layer, activation layer, and maxpool layer [12]. The convolution layer first extracts various features from the input image to produce a feature map, then passes the feature map to the activation layer to apply non-linearity, and the maxpool layer can reduce the variance of the feature map by downsampling. These are explained further in section 2.3.1 to 2.3.3.

The typical structure of a CNN includes multiple building blocks connected together to successively extract features and compress the image. Because this structure is simple yet effective, it is used by various state-of-the-art Object Detection Systems as their backbone Neural Network, including YOLOv3-Tiny.

2.3.1 Convolution Layer of YOLOv3-Tiny

The core idea of a convolution layer is to convolve a small filter matrix on top of a large input matrix, therefore extracting 2D spatial information of the input matrix. This makes CNN suitable for processing complicated and multi-dimensional image data.

YOLOv3-Tiny is using 3×3 and 1×1 convolution filters for its convolution layers. For the layers with a 3×3 filter, they are running with padding set to 1 and stride equals to 1, this ensures that the output has the same 2D dimension as the input. A convolution layer with a 1×1 filter is sometimes called Depth-wise Pooling layer or Channel-wise Pooling layer, since it does not change the 2D dimension of the image but can be used to compress the depth of the output.

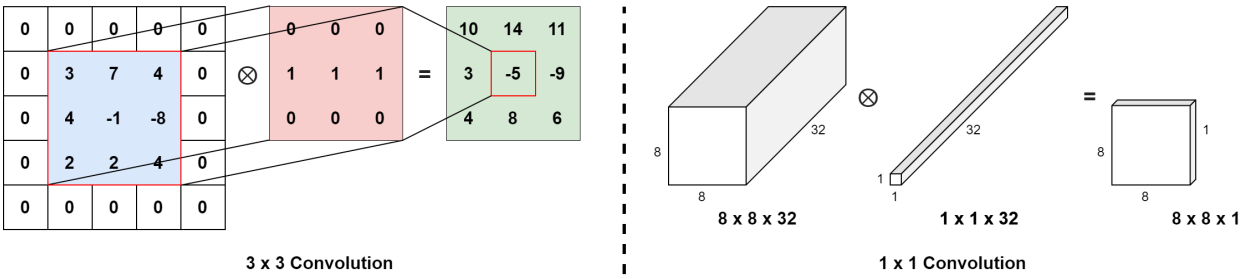

Figure 2.3: Example of 3×3 convolution and 1×1 convolution. A 3×3 convolution operates on the 2D plane to extract spatial information, whereas a 1×1 convolution compresses the depth of the input matrix in the 3D space.

2.3.2 Activation Layer of YOLOv3-Tiny

The activation layer applies non-linearity to the output of the convolution layer. Intuitively, they dynamically select which pixel describes a feature, and which pixel contains no useful information. This operation is crucial for Neural Networks, because without a non-linear activation function, all the stacked convolution layers will collapse into one single linear transformation.

YOLOv3-Tiny uses Leaky ReLU as its activation function:

$$f(x) = \begin{cases} x & \text{if x>0,} \\ \alpha x & \text{otherwise.} \end{cases} \qquad \alpha = 0.1 \text{ for YOLO} \tag{2.1}$$

This is an improved version of the original ReLU activation function, which allows the non-active (negative) pixels to pass with a small gradient. α is 0.1 for YOLOv3-Tiny.

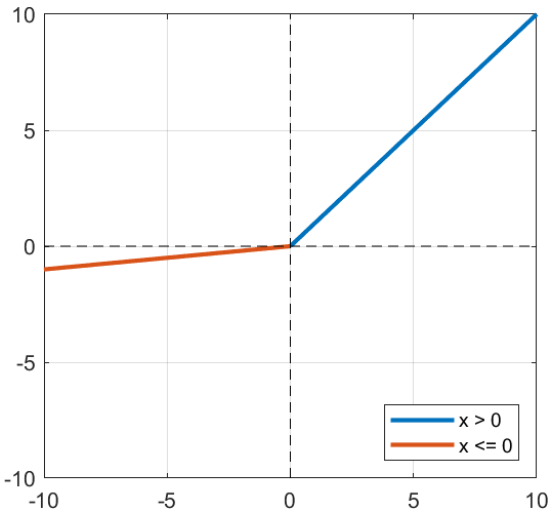

Figure 2.4: Plot of the Leaky ReLU activation function.

2.3.3 Maxpool Layer of YOLOv3-Tiny

Maxpooling is an efficient way to select sharp features from the image, and at the same time remove noise. Since YOLOv3-Tiny is using a 2×2 maxpool filter for all of its maxpool layers, they will reduce the width and height of the image by a factor of 2. One exception happens at maxpool layer 6, where padding is used and the stride is changed to 1 to keep the image size fixed.

The initial input to the CNN is 416×416, and the final outputs are two feature maps (matrices) reduced to 13×13 and 26×26.

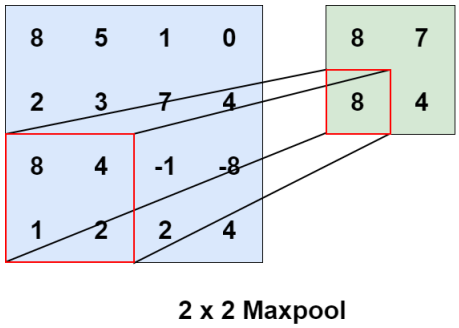

Figure 2.5: Example of 2×2 maxpool with stride = 2, the image size is reduced by half.

Chapter 3

Background - KiloCore 2 Platform

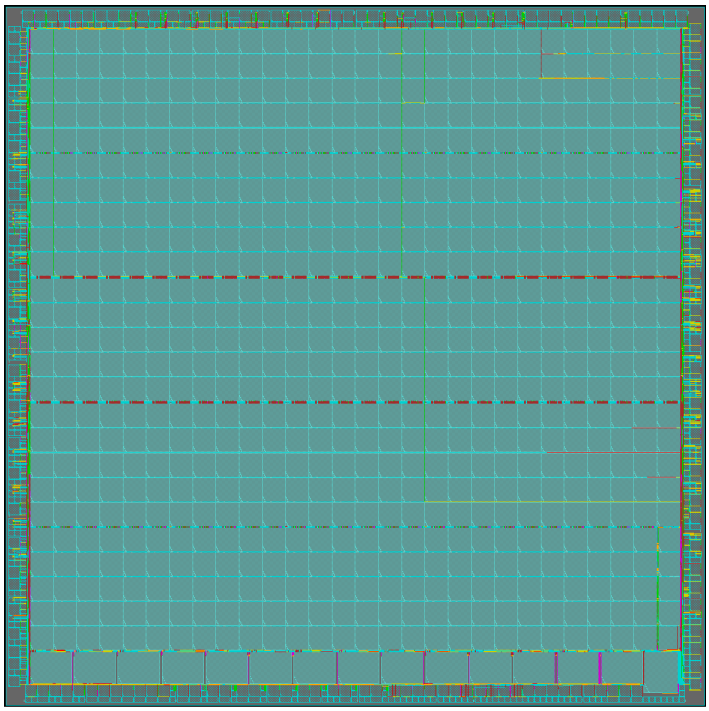

Figure 3.1: Overview of the KiloCore 2 chip [2] [3].

3.1 Overview

KiloCore 2 is the 4th generation of the Asynchronous Array of simple Processors (AsAP) manycore platform [20] [21] [22]. Compared with the previous generations [23] [24], KiloCore 2 comes with various enhancements such as circuit switch network, packet switch routers for

better long-distance communication, 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 that are each paired with a processor, 2 Viterbi accelerators, 1 FFT accelerator, and 14 memory modules containing 64KB of memory each that may be used for data or instructions [2] [25] [26]. [27]

3.2 Processors

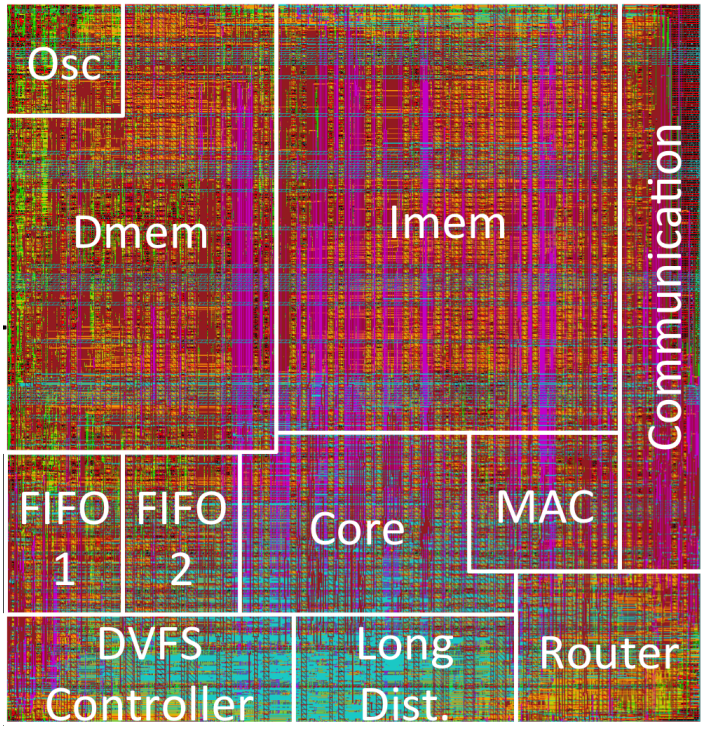

Figure 3.2: A single standard processor tile of the KiloCore 2 chip [2].

The KiloCore 2 chip is made up of small, user-programmable processor cores. These cores execute RISC-type instructions with their in-order, single-issue central datapath, and are interconnected using a mixture of statically configured circuit links, dynamic packet links, and dynamic circuit links. Each core has an independent oscillator for per-core Globally Asynchronous Locally Synchronous (GALS) clock generation [28]. A Dynamic Voltage Frequency Scaling (DVFS) controller selects power supply voltage between three power rails and a cutoff voltage based on

workload [29], so they dissipate exactly zero active power (leakage only) when they are in an idle state [2] [30] [31].

Standard processors and routers are designed to operate at 2.0 GHz at 900 mV. This achieves a 63% higher throughput per processor than the original KiloCore. Specialized high speed processors are designed to operate at 3.85GHz at 900mV. At 1.1V, the standard and high speed processors are projected to reach 2.9 GHz and over 5 GHz respectively. The entire array is projected to achieve over 2 tera-operations per second when running at 1.1V. [2].

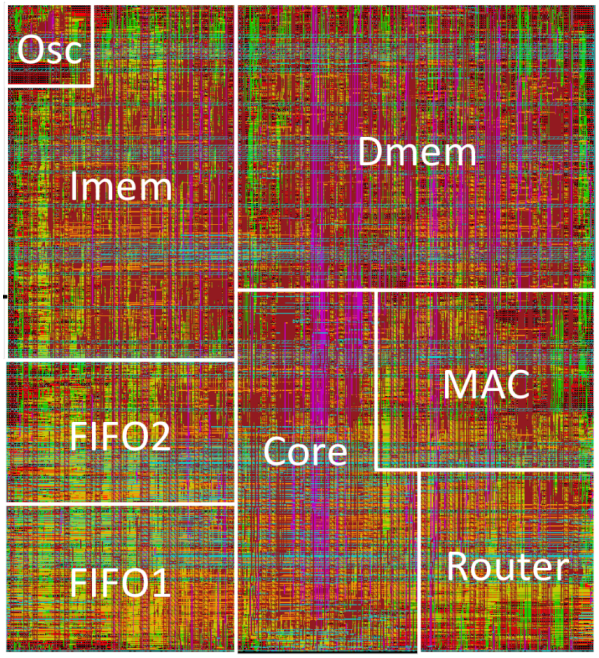

Figure 3.3: A single high-speed processor tile of the KiloCore 2 chip [2].

3.3 On-Chip SRAM

Fourteen 64KB SRAM memories are placed at the bottom of the KiloCore 2 chip, which offers 896KB total shared memory on-die, as shown in Figure 3.4. The memories can also be used to run larger programs on the neighboring core, acting as an expanded 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].

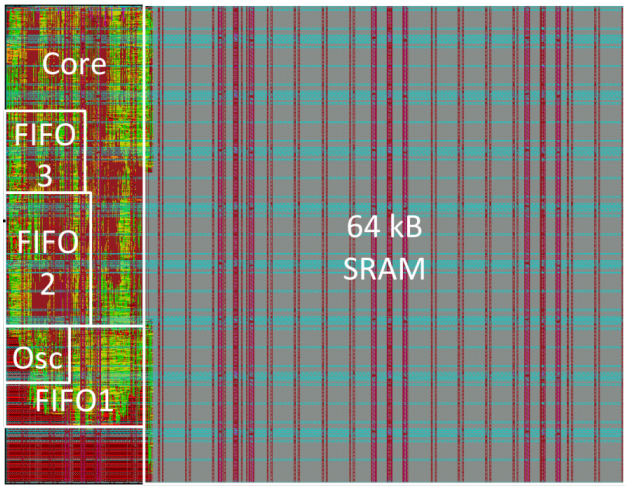

Figure 3.4: On-chip SRAM module [2].

3.4 Programming

The programming environment developed for KiloCore 2 is called Project Manager, which provides software packages for writing task-parallel applications, mapping the architecture on the chip, launching simulations to verify correctness, 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 2 supports 39-bit RISC-type assembly instructions formatted as `"Opcode, Destination, Source1, Source2, 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].

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 3.5 shows the Project Manager GUI after a simulation of a 650-processor version of the FFT application on KiloCore 2. Simulation metrics are recorded for each individual processor, along with a global summary.

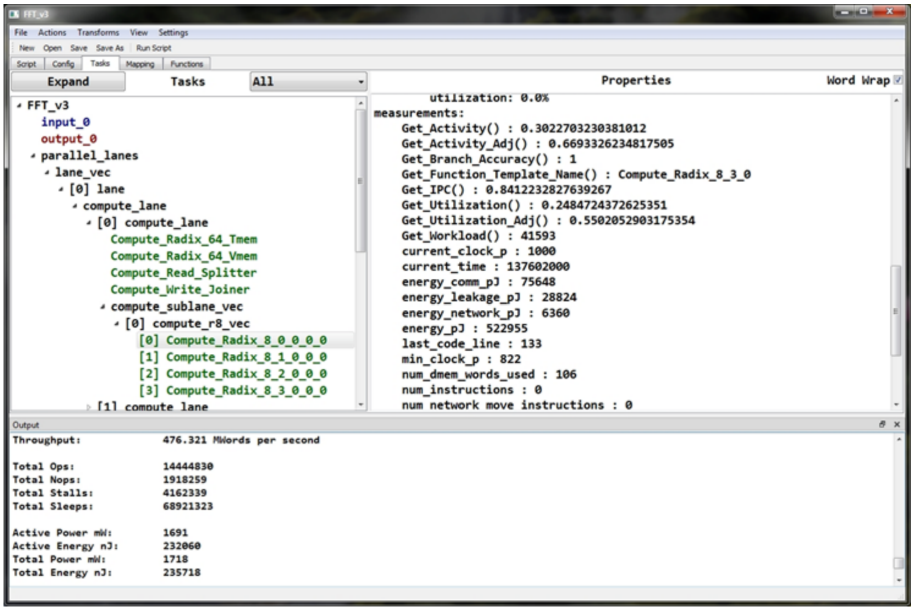

Figure 3.5: Project Manager GUI [2].

Chapter 4

YOLOv3-Tiny CNN Optimization for KiloCore 2

4.1 Overview

Convolutional Neural Networks are known for their intensive computation workload and large memory demand, but effective optimization techniques have been developed to help large CNNs run on low-power edge devices that have a simple architecture. For this work, Batch Normalization Folding is used to reduce computation complexity, and 16-bit fixed-point quantization is used to relax the memory requirements.

4.2 Batch Normalization Folding

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figure 4.1: Batch Normalization Algorithm. [4].

Batch Normalization (BN) is a common technique used in modern Convolutional Neural Networks. The main purpose of Batch Normalization is to achieve a stable distribution of activation values during training, so it is possible to use a much higher learning rate. In addition, random initialization of the weights can perform much better because BN layers can also act as a regularizer [4].

YOLOv3-Tiny employs Batch Normalization as well, but since this work is focused on the inference architecture, having additional BN layers will add significant computation complexity to the CNN. This is due to the operations in Batch Normalization, for example, division and calculating square roots, are all expensive to the ALU. To solve this problem, Batch Normalization Folding is used.

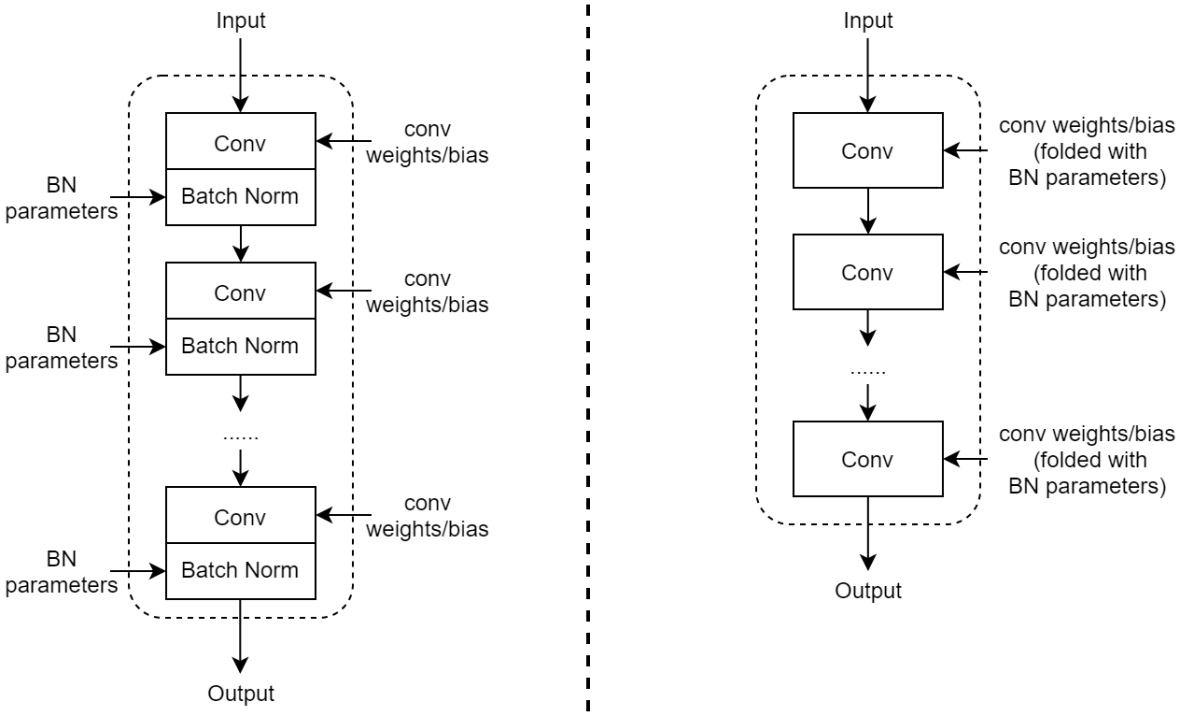

Figure 4.2: Visualization of BN Folding. Left side is a typical CNN without BN Folding, so after each convolution layer there is a Batch Normalization layer with additional parameters that must be streamed into the processor. Right side is a CNN with BN Folding, all the BN parameters are "folded" into the weights and biases so there is no need to use Batch Normalization layer during inference.

During inference time, the Batch Normalization parameters are no longer updating, so a BN layer becomes a simple linear transformation of the convolution layer's output. By "folding" the Batch Normalization parameters into the convolution layer's weights and biases beforehand, all

the BN layers can be removed during inference, and this will not cause any accuracy loss. A total of 12,736 parameters have been removed from YOLOv3-Tiny's CNN after Batch Normalization folding. Equation 4.1 to 4.4 gives the process of Batch Normalization folding.

$$\text{Convolution:} \quad \texttt{conv\_output} = \texttt{conv\_weight} * \texttt{input} + \texttt{conv\_bias} \tag{4.1}$$

$$\text{Batch Normalization} = \quad \boldsymbol{\gamma} \cdot \frac{\texttt{conv\_output} - \boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2 + \boldsymbol{\varepsilon}}} + \boldsymbol{\beta} \begin{cases} \boldsymbol{\mu} & \texttt{mean}\text{, calculated during training} \\ \boldsymbol{\sigma}^2 & \texttt{variance}\text{, calculated during training} \\ \boldsymbol{\gamma} & \texttt{scale}\text{, learnable parameter} \\ \boldsymbol{\beta} & \texttt{offset}\text{, learnable parameter} \\ \boldsymbol{\varepsilon} & \text{constant added for numerical stability} \end{cases} \tag{4.2}$$

$$\text{Folded } \texttt{conv\_weight} = \quad \frac{\boldsymbol{\gamma} \cdot \texttt{conv\_weight}}{\sqrt{\boldsymbol{\sigma}^2 + \boldsymbol{\varepsilon}}} \tag{4.3}$$

$$\text{Folded } \texttt{conv\_bias} = \quad \frac{\boldsymbol{\gamma} \cdot (\texttt{conv\_bias} - \mu)}{\sqrt{\boldsymbol{\sigma}^2 + \boldsymbol{\varepsilon}}} + \beta \tag{4.4}$$

4.3 16-bit Fixed-Point Quantization

To further reduce computation complexity, and to match the width of the hardware datapath, the YOLOv3-Tiny CNN is quantized from 32-bit floating-point down to 16-bit fixed-point. Because Neural Networks are trained to extract useful features from seemingly random inputs using deep connections, they are naturally robust against noise. Moreover, aggressive quantization, like 16-bit or even 8-bit, has been proven to have little to no effects on the accuracy of a Deep Neural Network [32]. Quantization also reduces the storage requirements for the weights and biases by 50%, from a total of 35.45MB down to 17.72MB.

The integer width and fraction width of the weights must be carefully chosen in order to avoid possible overflow and retain as much accuracy as possible. Figure 4.3 shows the distribution of all the weights for the YOLOv3-Tiny network. As the data suggests, the dynamic range of all

the weights is [-16.967, 20.564], this requires 6-bits integer width to cover the full range. However, to minimize the occurrence of overflow during convolution, one extra integer bit is used. With a Q7.9 number format, the representable range extends to [-64, 63.998]. A total of 488,483 weights are below precision using 9-bits fraction width, but the Signal to Quantization Noise Ratio (SQNR) is still below 10%. No weights are outside the representable range of Q7.9.

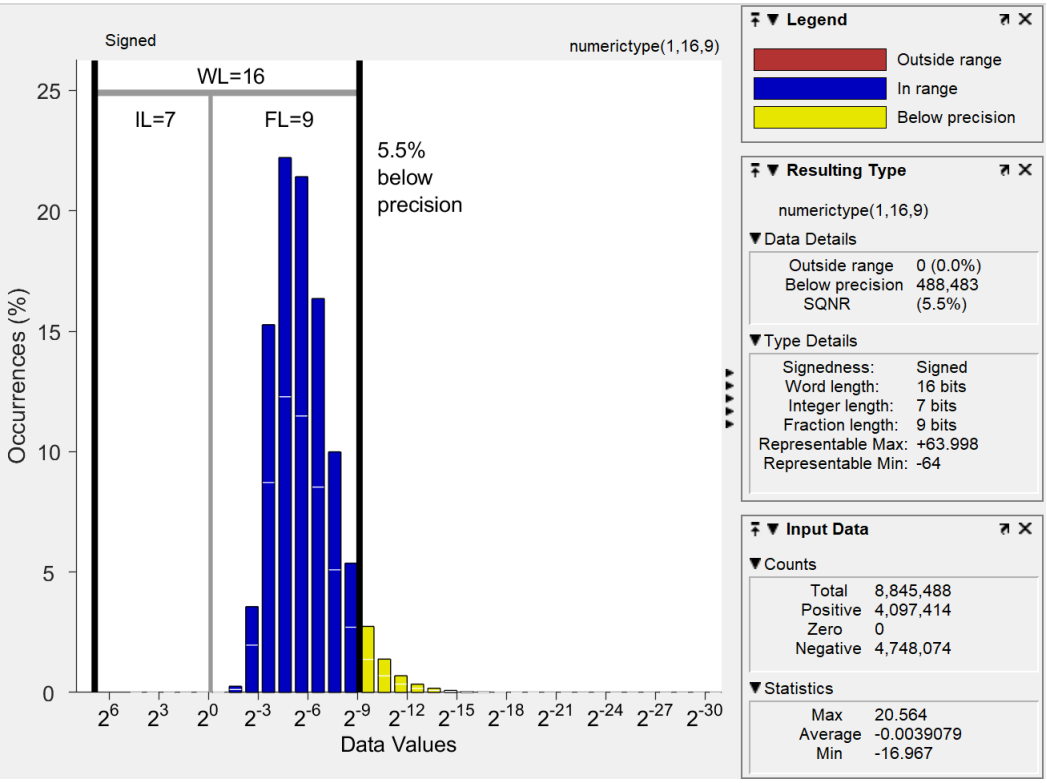

Figure 4.3: Distribution of all the weights for the YOLOv3-Tiny network. Only 5.5% of weights are below precision using a Q7.9 fixed-point quantization. Most of the weights are inside the range between 2^{-3} and 2^{-7}.

Figure 4.4 gives the distribution of all the biases. Since the dynamic range of all the biases is [-15.38, 9.3989], Q7.9 is also sufficient to represent the full range of biases. Only 6 biases are below precision, which is 0.2% of 3,694 biases.

A layer-by-layer analysis in Figure 4.5 shows that the dynamic range of the weights and biases shrinks significantly for the deeper layers. To preserve accuracy, starting from layer 6, the number format is changed from Q7.9 to Q5.11.

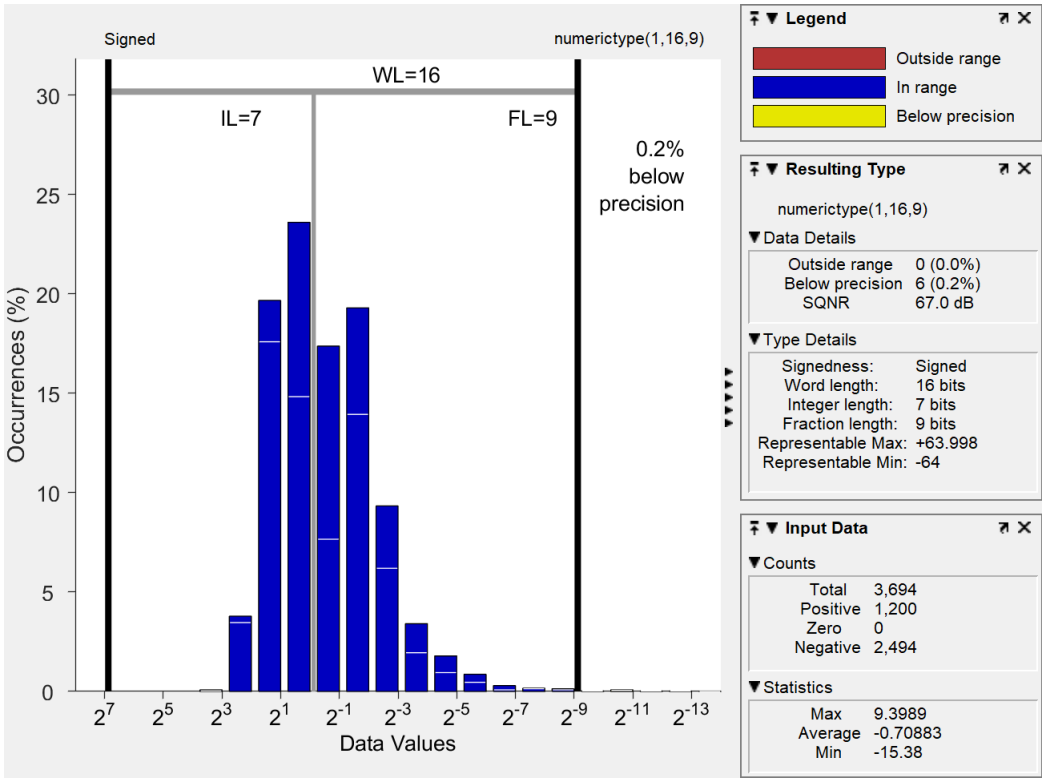

Figure 4.4: Distribution of all the biases for the YOLOv3-Tiny network. Only 6 (0.2%) biases are below precision using a Q7.9 fixed-point quantization. Most of the biases are inside the range between 2^2 and 2^{-2}.

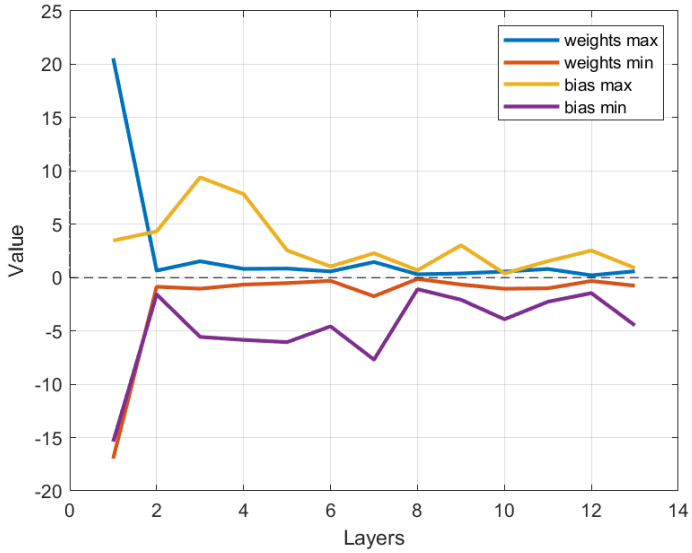

Figure 4.5: Max/Min trend of the weights and biases. The dynamic range shrinks significantly as the layer gets deeper.

4.4 Final CNN Structure

After applying Batch Normalization Folding and 16-bit fixed-point quantization, the optimized CNN now has 8,845,488 weights and 3,694 biases. The largest layer is layer 7, with a weight size of 3×3×512×1024 and an input size of 13×13×512. There are also two special layers in YOLOv3-Tiny: upsampling layer and routing layer. Upsampling layer is only used once between layer 11 and layer 12, and it is simply repeating the elements in the x and y direction to expand the input image from 13×13 to 26×26. The routing layer is performing matrix concatenation of the two inputs along the 3rd dimension. Intuitively, this is bringing the fine-grained features from the previous layer to the deeper layer so it can make a prediction with more details of the image.

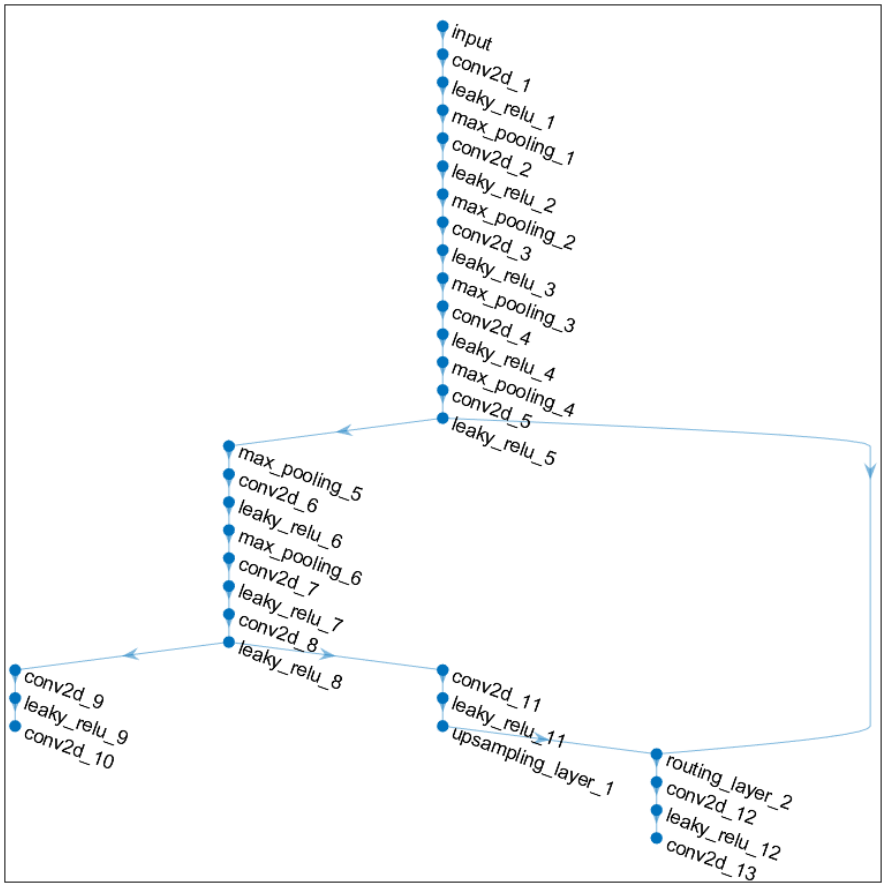

Figure 4.6: Final YOLOv3-Tiny CNN architecture after optimization. Each dot is a layer and the arrows indicate residual connection. Notice that all the Batch Normalization layers are removed comparing with the original CNN structure in Chapter 2 section 2 (Figure 2.2).

Name	Type	Activations	Learnables	Total Learnables
input 416x416x3 images	Image Input	416x416x3	-	0
conv2d_1 16 3x3x3 convolutions with stride [1 1] and padding 'same'	Convolution	416x416x16	Weights 3x3x3x16 Bias 1x1x16	448
leaky_relu_1 Leaky ReLU with scale 0.1	Leaky ReLU	416x416x16	-	0
maxpool_1 2x2 maxpool with stride [2 2]	Maxpool	208x208x16	-	0
conv2d_2 32 3x3x16 convolutions with stride [1 1] and padding 'same'	Convolution	208x208x32	Weights 3x3x16x32 Bias 1x1x32	4640
leaky_relu_2 Leaky ReLU with scale 0.1	Leaky ReLU	208x208x32	-	0
maxpool_2 2x2 maxpool with stride [2 2]	Maxpool	104x104x32	-	0
conv2d_3 64 3x3x32 convolutions with stride [1 1] and padding 'same'	Convolution	104x104x64	Weights 3x3x32x64 Bias 1x1x64	18496
leaky_relu_3 Leaky ReLU with scale 0.1	Leaky ReLU	104x104x64	-	0
maxpool_3 2x2 maxpool with stride [2 2]	Maxpool	52x52x64	-	0
conv2d_4 128 3x3x64 convolutions with stride [1 1] and padding 'same'	Convolution	52x52x128	Weights 3x3x64x128 Bias 1x1x128	73856
leaky_relu_4 Leaky ReLU with scale 0.1	Leaky ReLU	52x52x128	-	0
maxpool_4 2x2 maxpool with stride [2 2]	Maxpool	26x26x128	-	0
conv2d_5 256 3x3x128 convolutions with stride [1 1] and padding 'same'	Convolution	26x26x256	Weights 3x3x128x256 Bias 1x1x256	295168

Table 4.1: Detail of the YOLOv3-Tiny Concolutional Neural Network

leaky_relu_5 Leaky ReLU with scale 0.1	Leaky ReLU	26x26x256	-	0
maxpool_5 2x2 maxpool with stride [2 2]	Maxpool	13x13x256	-	0
conv2d_6 512 3x3x256 convolutions with stride [1 1] and padding 'same'	Convolution	13x13x512	Weights 3x3x256x512 Bias 1x1x512	1180160
leaky_relu_6 Leaky ReLU with scale 0.1	Leaky ReLU	13x13x512	-	0
maxpool_6 2x2 maxpool with stride [1 1] and padding 'same'	Maxpool	13x13x512	-	0
conv2d_7 1024 3x3x512 convolutions with stride [1 1] and padding 'same'	Convolution	13x13x1024	Weights 3x3x512x1024 Bias 1x1x1024	4719616
leaky_relu_7 Leaky ReLU with scale 0.1	Leaky ReLU	13x13x1024	-	0
conv2d_8 256 1x1x1024 convolutions with stride [1 1] and padding 'same'	Convolution	13x13x256	Weights 1x1x1024x256 Bias 1x1x256	262400
leaky_relu_8 Leaky ReLU with scale 0.1	Leaky ReLU	13x13x256	-	0
conv2d_9 512 3x3x256 convolutions with stride [1 1] and padding 'same'	Convolution	13x13x512	Weights 3x3x256x512 Bias 1x1x512	1180160
leaky_relu_9 Leaky ReLU with scale 0.1	Leaky ReLU	13x13x512	-	0
conv2d_10 255 1x1x512 convolutions with stride [1 1] and padding 'same'	Convolution	13x13x255	Weights 1x1x512x255 Bias 1x1x255	130815
conv2d_11 128 1x1x256 convolutions with stride [1 1] and padding 'same'	Convolution	13x13x128	Weights 1x1x256x128 Bias 1x1x128	32896
leaky_relu_11 Leaky ReLU with scale 0.1	Leaky ReLU	13x13x128	-	0

Table 4.2: Detail of the YOLOv3-Tiny Concolutional Neural Network, Cont'd.

upsample_1 [1 1] upsampling for YOLOv3	Upsample 2D	13x13x128	-	0
routing_2 Depth concatenation of 2 inputs	Depth Concatenation	26x26x384	-	0
conv2d_12 256 3x3x384 convolutions with stride [1 1] and padding 'same'	Convolution	26x26x256	Weights 3x3x384x256 Bias 1x1x256	884992
leaky_relu_12 Leaky ReLU with scale 0.1	Leaky ReLU	26x26x256	-	0
conv2d_13 255 1x1x256 convolutions with stride [1 1] and padding 'same'	Convolution	26x26x255	Weights 1x1x256x255 Bias 1x1x144	65535

Table 4.3: Detail of the YOLOv3-Tiny Concolutional Neural Network, Cont'd.

Chapter 5

YOLOv3-Tiny Architecture on KiloCore 2

5.1 Overview

One major goal of this thesis is to show that the KiloCore 2 chip is capable of running a large, modern Convolutional Neural Network, especially on the real KiloCore 2 Test System hardware. Conceptually, the program workflow can be described as a 3-level model, the levels are: System Level, Chip Level, and Core Level.

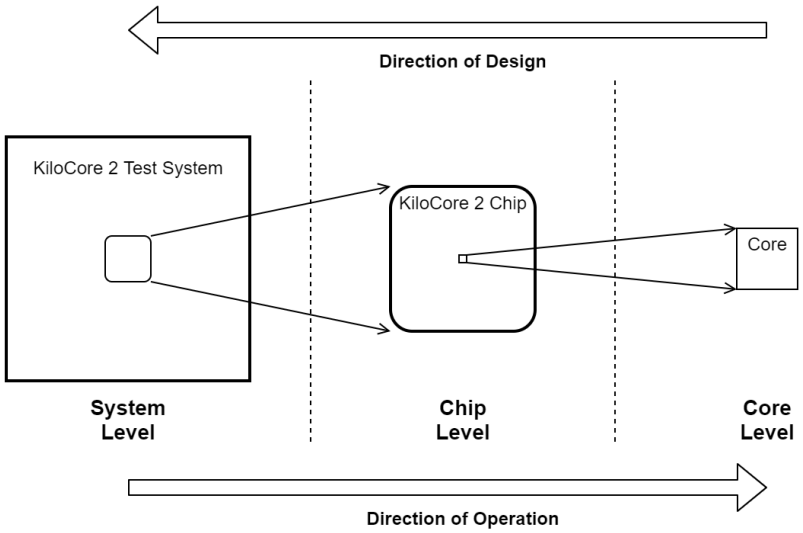

Figure 5.1: 3 level of the implementation. When designing the architecture, programs are written for the cores, then the cores are connected together on the chip level, taking inputs from the test system (Direction of Design). When the implementation is in operation, data is saved on the system level and streamed into the chip, then processed inside the cores (Direction of Operation).

5.1.1 Overview - System Level

This is the top level of the implementation. For this level, the available hardware is the KiloCore 2 Test System, equipped with one KiloCore 2 chip and 1GB of DDR3 DRAM. The inputs to this level are the original image, full weights and biases, and configuration parameters of the CNN. All the data is stored inside the 1GB DRAM, which is controlled by the TE0712 FPGA. The FPGA will stream the CNN layer by layer into the KiloCore 2 chip to produce the final output, and the final output of this level is the CNN detection result, which are two matrices in $13 \times 13 \times 255$ and $26 \times 26 \times 255$.

5.1.2 Overview - Chip Level

For one KiloCore 2 chip, there are 697 user-programmable cores and fourteen 64KB shared SRAMs. Since it is impossible to store all the weights and biases inside one chip, the CNN in this level is processed layer by layer. To make the program more efficient, three layers (`conv-activation-maxpool`) are combined into one program as three stages: **data distribution, computation,** and **output buffering**. The inputs to this level are the configuration parameters, weights/biases, and the input image of the current layer. The output of this level is the convolution result of the current layer, which is stored off-chip temporarily in the 1GB DRAM, so it can be streamed back into the chip as the input to the next layer.

5.1.3 Overview - Core Level

On the core level, the `conv-activation-maxpool` workflow is broken into simple and small tasks to achieve higher throughput, massive parallelism, and lower power consumption. In the **data distribution** stage, depending on the size of the image and the weights/biases, the data distribution cores direct some of the input data to the on-chip SRAM to be reused throughout the convolution process. The second stage is **computation**, where the convolution cores and adder cores do 3D convolution. For the **output buffering** stage, the intermediate 3D convolution results are temporarily stored in the on-chip SRAM. So after getting all the convolution outputs from the computation stage, Leaky-ReLU activation and Maxpool are applied here before finally outputting the data to the off-chip DRAM.

5.2 System Level Operarion

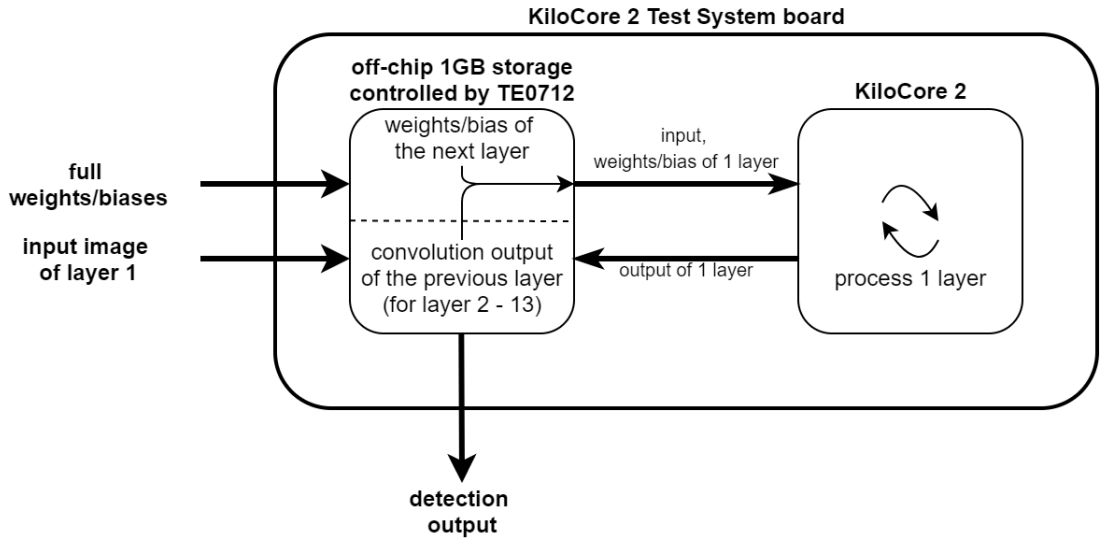

Figure 5.2: System level operation. The full set of weights/biases and the input image are stored in the off-chip storage, and they are streamed layer by layer into KiloCore 2. For each iteration KiloCore 2 processes conv/activation/maxpool in one shot and outputs the data back to the off-chip storage.

On the system level, KiloCore 2 is in charge of all the CNN computations, and the 1GB DRAM is controlled by the TE0712 FPGA board. In the beginning, the FPGA will send the input image and the weights/biases of layer 1 from the DRAM to KiloCore 2. After Kilocore 2 finishes outputting the result of layer 1 to the DRAM, they are combined with the weights/bases of layer 2, then streamed back into KiloCore 2. The loop continues until all 13 layers are finished.

As Figure 5.3 shows, the KiloCore 2 chip starts by loading the programs into its cores, and take the input image at layer 1. The CNN of YOLOv3-Tiny is being processed one layer at a time, and it produces two output matrices after layer 13 of YOLOv3-Tiny is done. No reprogramming is needed between layer 1 to layer 3, and between layer 4 to layer 13. However, during operation, KiloCore 2 needs to be stopped once and reprogrammed at layer 4, because the hardware architecture needs to be switched in order to process the large amount of weights and biases. Details are explained further in the next section.

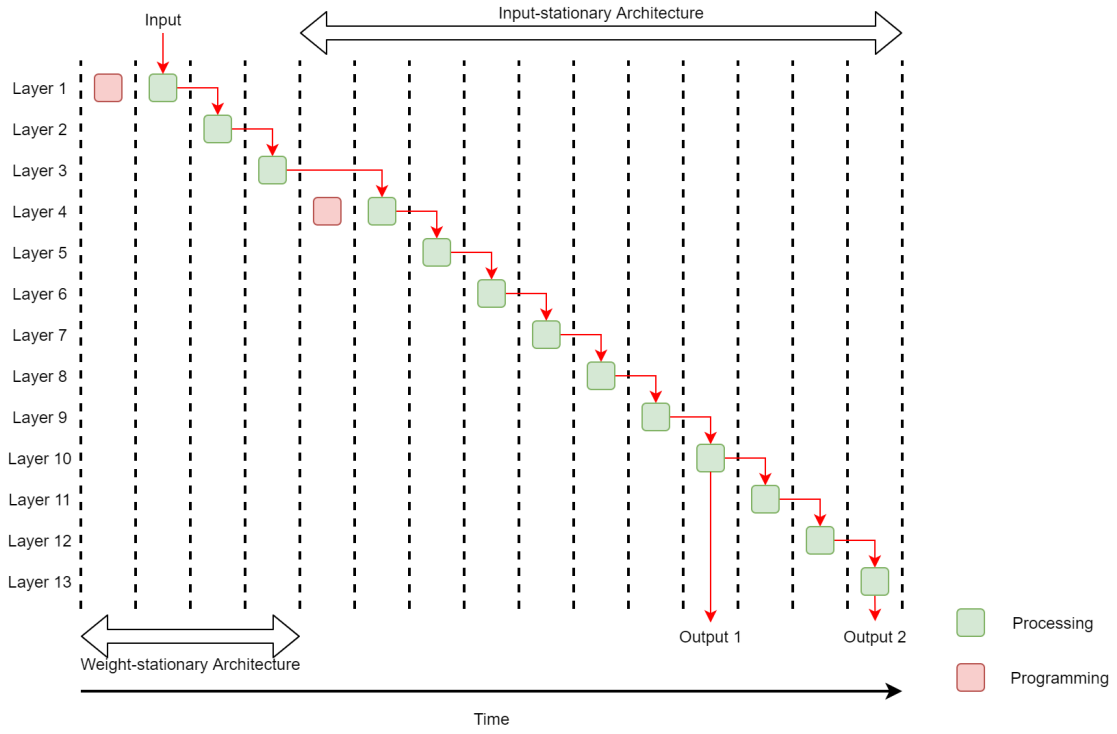

Figure 5.3: Operation timing diagram. Green means KiloCore 2 is processing that layer, and red means KiloCore 2 is switching architectures, which is discussed in section 5.3. Red arrow indicates data flow.

5.3 Chip Level Architectures

For a typical CNN dataflow, as the layer gets deeper, the image is compressed smaller and the depth of the weights becomes larger. This pattern applies to YOLOv3-Tiny as well. The largest 2D image size is 416×416 (layer 1), and the smallest 2D image size is 13×13 (layer 6, 7, 8, 9, 10, 11). For the weights, the smallest weight is 3×3×3×16 (layer 1), and the largest weight is 3×3×512×1024 (layer 7). A detailed analysis of every layer is shown in table 5.1.

The input image and weights/biases should be stored very close to the processor since CNN optimization relies heavily on data reuse. However, although the CNN is processed layer by layer on the chip level, it is still impossible to save the full input image plus the weights/biases of one layer in the on-chip 896KB SRAM. To bring data as close to the processor as possible, while still being flexible enough to fit different layers of YOLOv3-Tiny on the KiloCore 2 chip, this work presents two chip level architectures to process CNN with minimal memory usage: **weight-stationary architecture** and **input-stationary architecture.**

Layer	Weights	Bias	Total	Parameter Size (KB)	Input	Input Size (KB)	Output	Output Size (KB)
1	3x3x3x16	1x1x16	448	0.875	416x416x3	1014	208x208x16	1352
2	3x3x16x32	1x1x32	4640	9.0625	208x208x16	1352	104x104x32	676
3	3x3x32x64	1x1x64	18496	36.125	104x104x32	676	52x52x64	338
4	3x3x64x128	1x1x128	73856	144.25	52x52x64	338	26x26x128	169
5	3x3x128x256	1x1x256	295168	576.5	26x26x128	169	13x13x256	84.5
6	3x3x256x512	1x1x512	1180160	2305	13x13x256	84.5	13x13x512	169
7	3x3x512x1024	1x1x1024	4719616	9218	13x13x512	169	13x13x1024	338
8	1x1x1024x256	1x1x256	262400	512.5	13x13x1024	338	13x13x256	84.5
9	3x3x256x512	1x1x512	1180160	2305	13x13x256	84.5	13x13x512	169
10	1x1x512x255	1x1x255	130815	255.5	13x13x512	169	13x13x255	84.2
11	1x1x256x128	1x1x128	32896	64.25	13x13x256	84.5	13x13x128	42.25
12	3x3x384x256	1x1x256	884992	1728.5	26x26x384	507	26x26x256	338
13	1x1x256x255	1x1x144	65535	128.0	26x26x256	338	26x26x255	336.7

Table 5.1: Detailed information showing the data size of each layer. Green means less than 896KB, red means larger than 896KB (larger than the capacity of the on-chip SRAM).

5.3.1 Data Distribution Stage

For YOLOv3-Tiny, layer 1 to layer 3 are using the weight-stationary architecture, layer 4 to 13 are using the input-stationary architecture. In other words, the weights/biases are stored on-chip for layer 1 to layer 3, and the input is stored on-chip for layer 4 to layer 13. The major difference between the two architectures is in the data distribution stage, where the weight-stationary architecture is distributing the weights from the SRAM to multiple convolution cores, and the input-stationary architecture is distributing the input image along its 3rd dimension (usually called channels). Figure 5.4 shows the different distribution paths for the two architecturs.

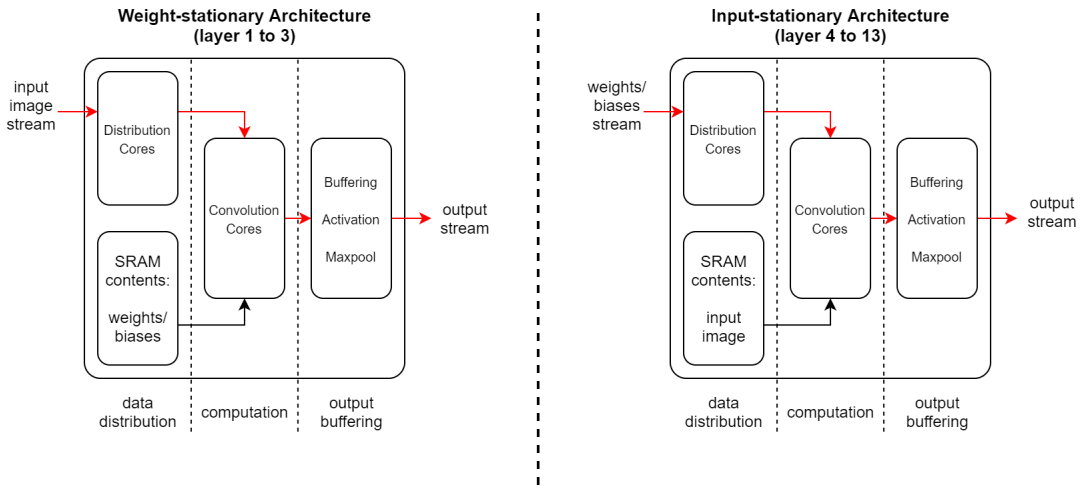

Figure 5.4: Left: Weight-stationary Architecture. Right: Input-stationary Architecture. Red path highlights the data stream.

5.3.2 Computation Stage

During the first stage, the dimension of the input data is reduced to 1D vectors. So in the computation stage, the direction of the dataflow is pointing to the higher dimension to rebuild the final 3D output tensor. As the vectorized image and weights go into the 2D convolution modules, they are multiplied and accumulated together to form a 2D matrix. Then after passing through a series of adders, multiple 2D matrices are combined along the 3rd dimension to form one output image. By repeating this process on the same image with different weights, the final output is in the correct shape and will be outputted in row-major order.

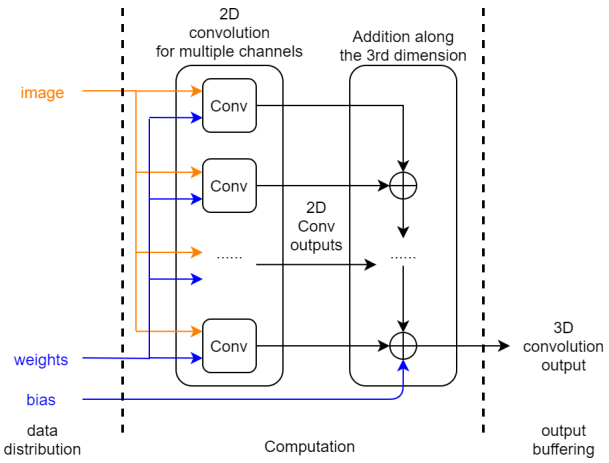

Figure 5.5: High level dataflow of the computation stage. Data are streamed into the 2D and 3D Convolution modules, the convolution outputs might be incomplete due to the layer being too deep so the output is connected to the Output Buffering Stage for post-processing.

5.3.3 Output Buffering Stage

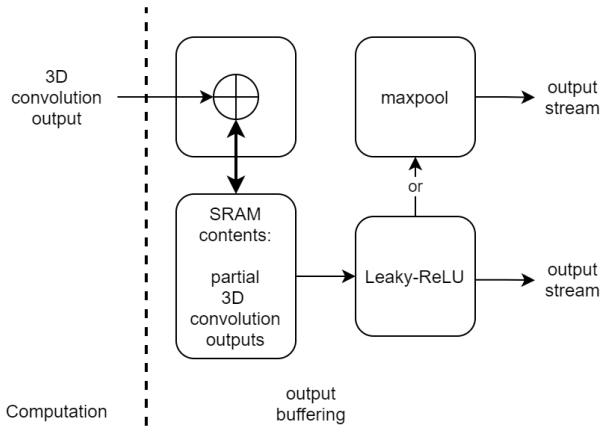

Figure 5.6: High level dataflow of the output buffering stage. Notice that SRAM has bi-directional communication with the adder. The final output is either after activation or maxpool, depending on that layer's configuration.

The output buffering stage is designed to process images that are deep in the 3rd dimension (channels). One 3D convolution module can process 3 to 16 channels at a time, and the output buffers can save the intermediate outputs from the 3D Conv module, combine them with the previous 3D Conv output, then wait for the next batch of outputs. When the computation stage finishes all the channels, the output buffers can apply activation function element-wise and maxpool the image, depending on that layer's configuration. In this way, the 2D and 3D convolution cores don't have to interact with the SRAM and the activation/maxpool algorithms. Instead, they can focus on just performing the MAC operation, which simplifies the code, and makes the hardware pipeline runs more efficiently.

5.3.4 Chip Level Summary

The two chip level architectures share the same logic for the computation stage and output buffering stage, so no reprogramming is needed for these cores when switching architectures for layer 4. This reduces the overall delay and saves power when the program is running on only one KiloCore 2 chip.

Figure 5.7 shows how KiloCore 2 is utilized under different workloads. When taking inputs from the off-chip storage, the computation stage and output buffering stage are staying idle because the input distribution logic is not ready to send data for processing. When data input is finished and SRAMs have all the weights/image, the full chip starts circulating data and all three stages are fully utilized.

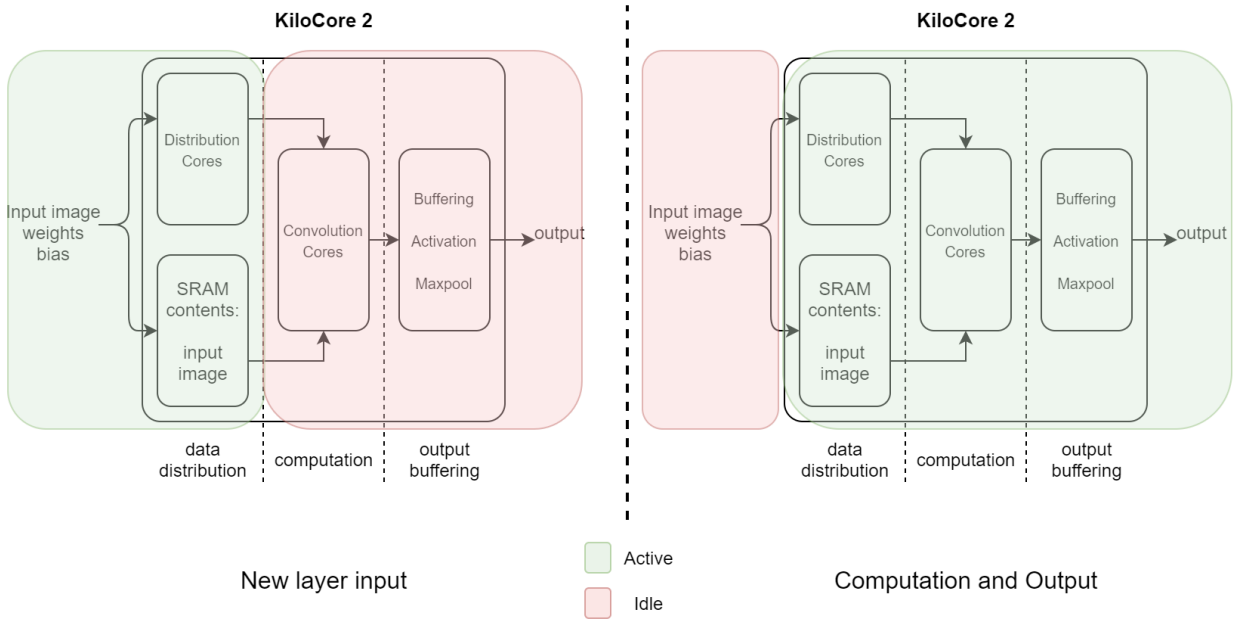

Figure 5.7: Difference showing the active and idle region of KiloCore 2 when processing a layer. When new data is being streamed into the chip, the computation and output buffering stages are in idle state. When the chip has all the data for the current layer, it starts processing and the chip is fully utilized.

Furthermore, all the hardware modules in the two architectures are designed to be scalable. Since KiloCore 2 has four circuit I/O ports that support standard 16-bit data, processing four sets of weights in parallel can increase the throughput by four times, as shown in Figure 5.8.

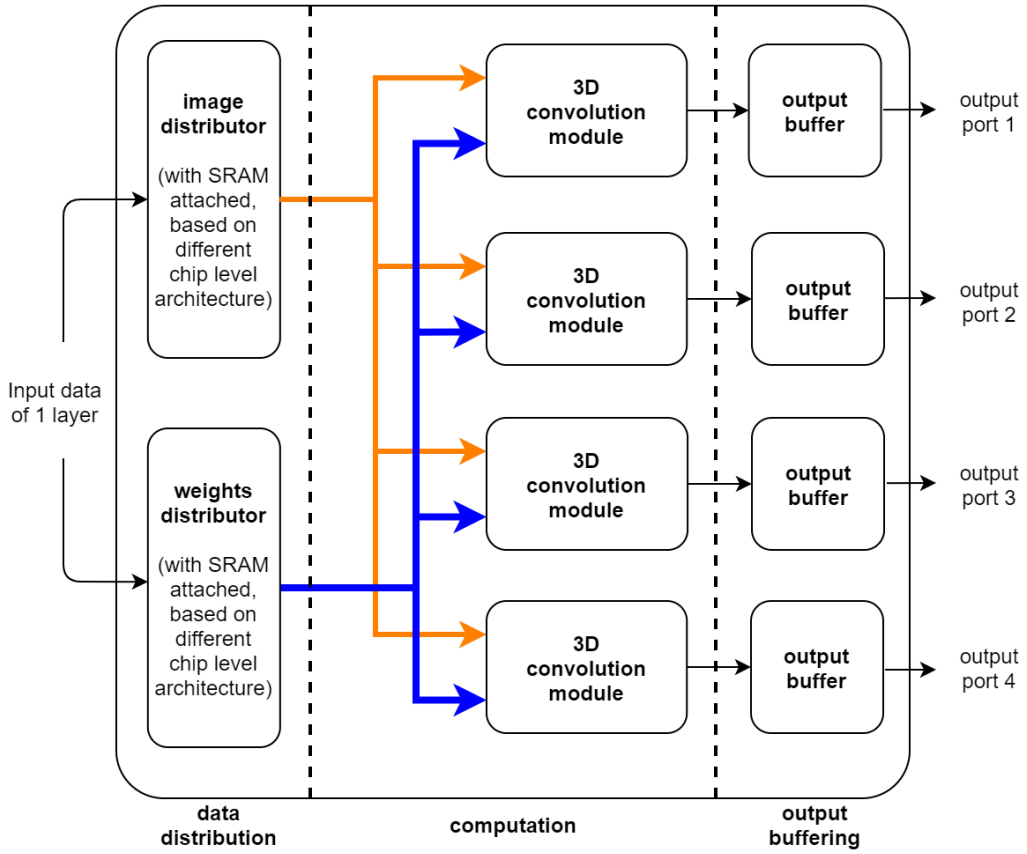

Figure 5.8: Chip level parallel operation. This Figure gives an overview of one full KiloCore 2 chip. Notice that each of the 3D convolution modules in the computation stage contains multiple 2D convolution modules inside, depending on the input layer depth.

5.4 Core Level Implementation

The principle of designing programs for manycore platforms is to break down the large application into small and simple tasks, then deploy the tasks to the cores so that each core can focus on doing only one job. Practically, this helps eliminate branch instructions and can reduce the number of function calls.

The user-programmable cores in KiloCore 2 support standard C++ language, working with the Clang and LLVM infrastructure, modern compiler optimizations are also available. This section will discuss the detailed algorithms of the core level implementations.

5.4.1 im2col

The fundamental convolution algorithm implemented in this work is `im2col`, Figure 5.9 gives an example comparing `im2col` with intuitive convolution. In short, `im2col` reshapes the complex multi-dimensional convolution process into a simple 2D matrix multiplication [33] [34]. This algorithm suits KiloCore 2 well for two reasons. First, the processors in KiloCore 2 have a dedicated MAC module with a 9-stage hardware pipeline, which naturally accelerates matrix multiplication. Second, traditional `im2col` implementations require large memory to save the reshaped image and weights, and the throughput is usually bottlenecked by memory bandwidth. But with its manycore architecture, KiloCore 2 can avoid this problem by distributing the matrix to be processed to a large pool of cores, thus reducing the memory requirements for each core.

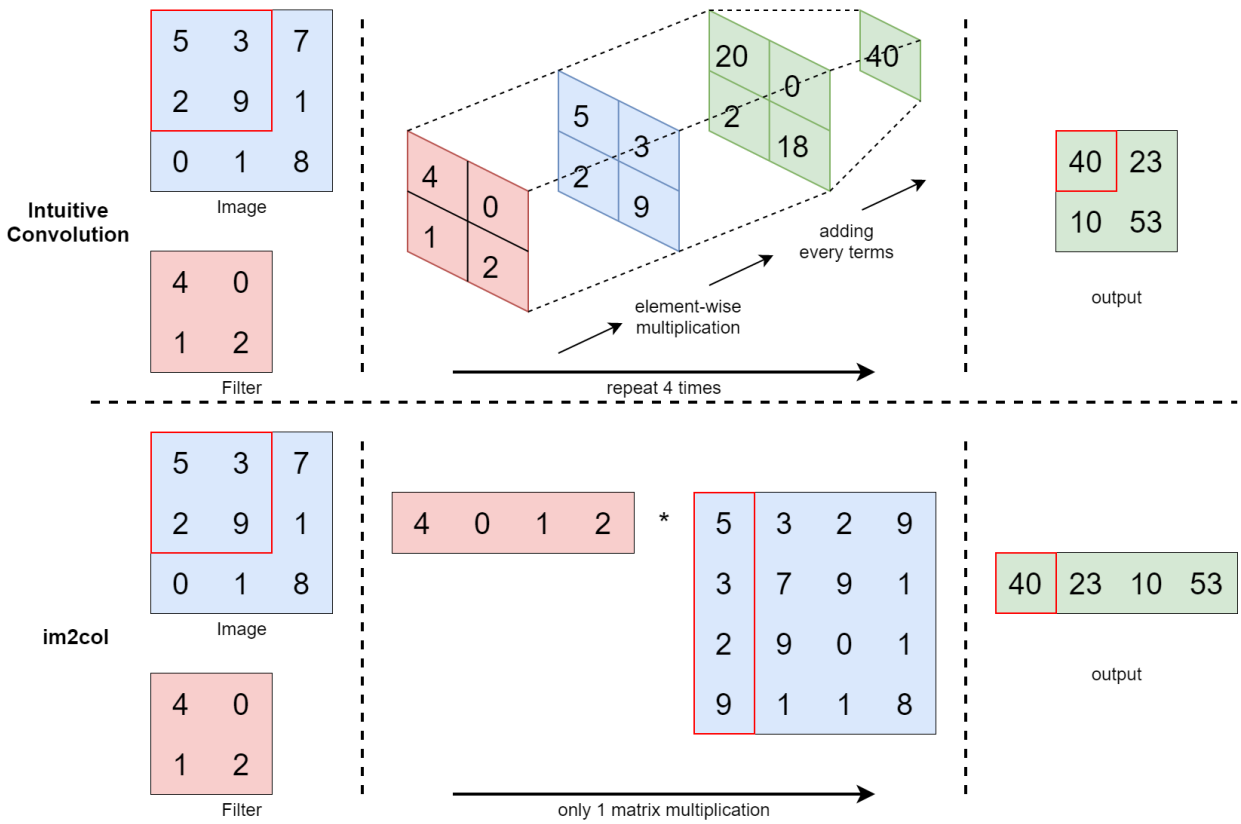

Figure 5.9: im2col algorithm demo showing in the bottom. The multi-dimension convolution is flattened in to a single matrix multiplication.

5.4.2 Image Distributor (Weight-stationary Architecture)

The image distributor in weight-stationary architecture has four cores: one for processing the inputs and communicating with the SRAM, three for distributing the image according to the width of the filters. One channel of the input image is streamed into the input core in row-major order, since YOLOv3-Tiny uses 3x3 filters, the input core will always buffer three rows of the image in the SRAM, then distribute them concurrently. Therefore, later in the computation stage, the 2D convolution module will get the full `im2col` window, and can reshape the three rows of image into a column vector.

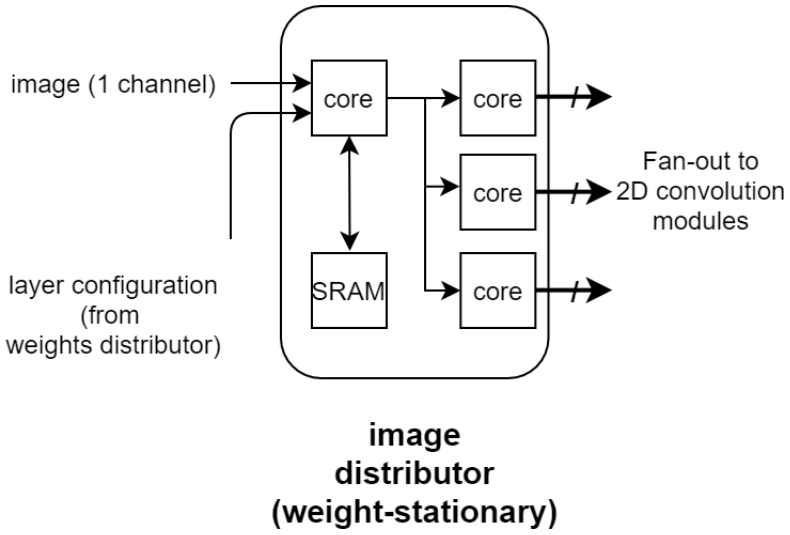

Figure 5.10: Image distributor, weight-stationary architecture.

Another job for the image distributor is padding. YOLOv3-Tiny always uses "same" padding for its inputs, this means adding two extra rows of zeros to the first and last row, plus two extra columns of zeros on the left side and right side. To reduce the number of SRAM reads and writes, the zeros are added dynamically during distribution. Configuration information including image size and filter width is sent from the weights distributor.

The distribution core is configured like a single-input, multiple-output (SIMO) broadcaster. The outputs of the image distributor go directly to the 2D convolution module in the computation stage.

Algorithm 1 image distributor (weight-stationary architecture)

1: image_dimension $\leftarrow$ INPUT

2: loop($i <$ image_depth)

```
    loop(j < image_height)
        SRAM ← 3 rows of image ← INPUT                          ▷ buffer 3 rows
        loop(k < image_width+2)
            0 → all OUTPUT rows                    ▷ first column, padding zeros
            if (j == 0) then
                0 → OUTPUT row 0                      ▷ first row, padding zeros
                SRAM row j → OUTPUT row 0
                SRAM row j+1 → OUTPUT row 1
            else if (j == image_height) then
                SRAM row j+1 → OUTPUT row 1
                SRAM row j+2 → OUTPUT row 2
                0 → OUTPUT row 2                       ▷ last row, padding zeros
            else
                SRAM row j → OUTPUT row 0
                SRAM row j+1 → OUTPUT row 1
                SRAM row j+1 → OUTPUT row 2
            end if
            0 → all OUTPUT rows                     ▷ last column, padding zeros
        end loop
    end loop
end loop
```

5.4.3 Image Distributor (Input-stationary Architecture)

This image distributor is designed to use a large portion of the SRAM to store the whole input image. Since the input-stationary architecture is applied from layer 4 to layer 13, it must have enough memory to store the largest input image among those layers, which is at layer 12. The input dimension of layer 12 is 26×26×384 (259,584 numbers), because each SRAM can store 32,768 16-bit numbers, this means the image distributor needs at least 8 SRAM modules to accommodate for layer 12 (32,768×8 = 262,144 numbers).

The distribution flow uses a hierarchical structure to repeatedly collect 16 channels of the

input image from the 8 SRAM modules and send them to the 2D convolution cores. Figure 5.11 shows an example using the input dimension of layer 4, which is 52×52×64. To distribute all 64 channels, the loop needs to run four times. The same logic applies to all the later layers, and the loop count is calculated at runtime, so no reprogramming is needed between layers for this image distributor. Same as the image distributor in the weight-stationary architecture, padding is done dynamically during distribution.

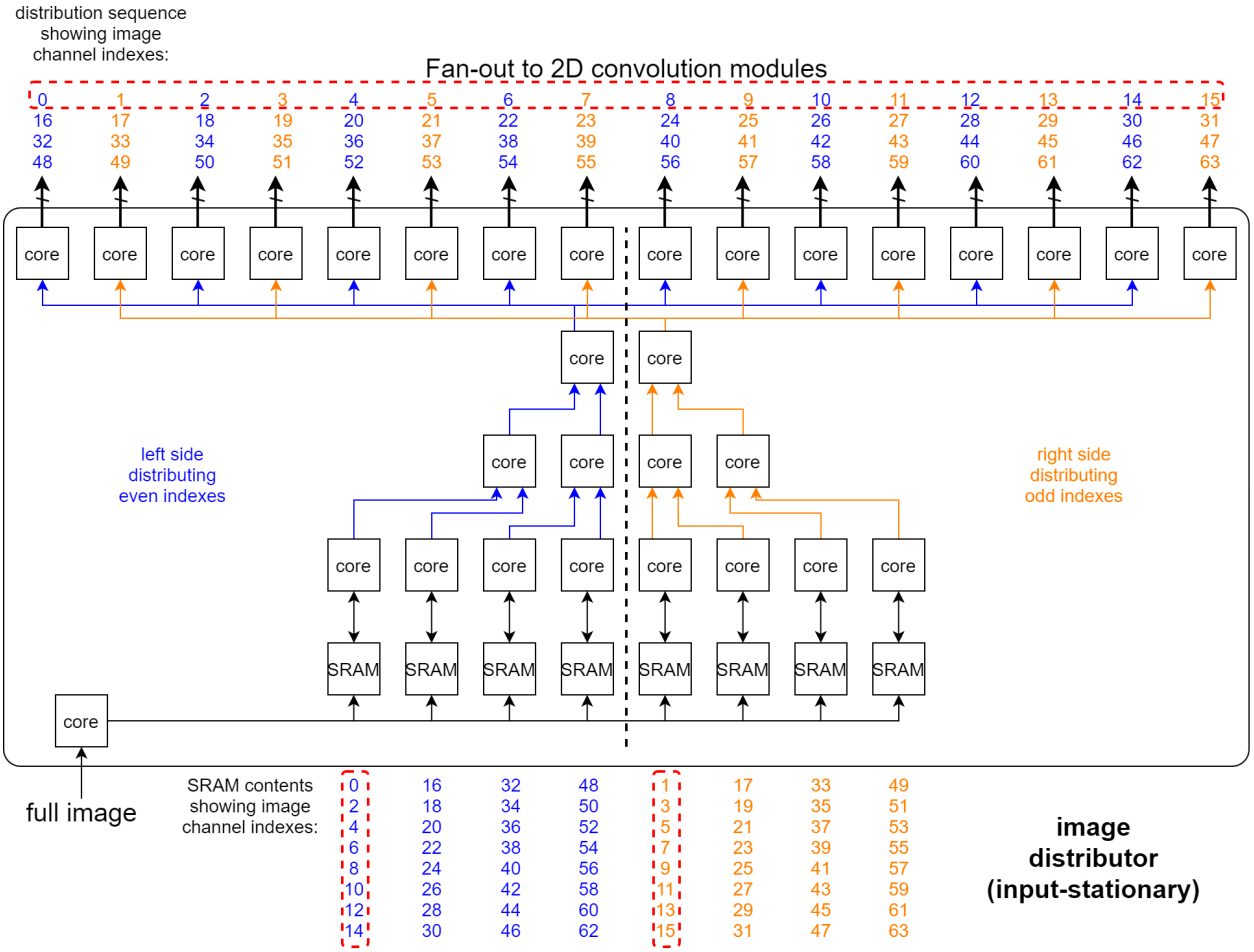

Figure 5.11: Image distributor, input-stationary architecture. The left side distributes all the even indexes, showing in blue. The right side distributes all the odd indexes, showing in orange. To distribute all 64 channels, the distributor needs to loop 4 times, outputting 16 channels at a time.

The image channels are saved into the SRAM based on the parity of their index, the even indexes are saved to the left four SRAMs, and the odd indexes are saved to the right four SRAMs. This design ensures that the two final distribution cores can output the 16 channels in series together, so the adder tree in the 3D convolution module doesn't have to stall.

If no parity check is implemented when saving the channels into the SRAMs, then for the

two final distribution cores, the left core will output the first eight channels (channel 0 to 7), and the right core will output the last eight channels (channel 8 to 15). This distribution scheme will cause the adders in the 3D convolution module to stall because the adders are working in series. Although the adder at channel 8 can have one of its addend ready when the convolution for channel 8 is finished, it must wait for the first eight adders to finish their job in order to have another addend appears at its input port.

Algorithm 2 image distributor (input-stationary architecture), input logic

```
1: image_dimension ← INPUT
2: loop(i < image_depth)
3:     image_one_channel ← INPUT
4:     if channel is even then                    ▷ save image to left four SRAMs
5:         SRAM_index ← (channel_index / 16 mod 4)
6:         SRAM ← SRAM_index ← image_one_channel
7:     else if channel is odd then                ▷ save image to right four SRAMs
8:         SRAM_index ← (channel_index / 16 mod 4 + 4)
9:         SRAM ← SRAM_index ← image_one_channel
10:    end if
11: end loop
```

Algorithm 3 image distributor (input-stationary architecture), internal logic

```
1: image_dimension ← INPUT
2: num_image_sets ← image_depth / 8
3: for (every 2 image_sets) do                                  ▷ loop unrolling
4:     loop(i < image_height)
5:         loop(j < image_weight)
6:             image_one_channel ← INPUT port 0     ▷ read in and pass left input
7:             image_one_channel → OUTPUT
8:         end loop
9:     end loop
10:    loop(i < image_height)
11:        loop(j < image_weight)
12:            image_one_channel ← INPUT port 1    ▷ read in and pass right input
```

```
            image_one_channel → OUTPUT
        end loop
    end loop
end for
```

Algorithm 4 image distributor (input-stationary architecture), distribution logic

```
image_dimension ← INPUT
loop(i < image_depth)
    loop(j < image_height)
        loop(k < image_width+2)
            0 → broadcast all OUTPUT ports        ▷ first column, padding zeros
            if (j == 0) then
                0 → broadcast all OUTPUT ports      ▷ first row, padding zeros
            else if (j == image_height) then
                0 → broadcast all OUTPUT ports       ▷ last row, padding zeros
            else
                image_eight_channels ← INPUT
                image_one_channel → broadcast all OUTPUT ports
            end if
            0 → broadcast all OUTPUT ports        ▷ last column, padding zeros
        end loop
    end loop
end loop
```

5.4.4 Weights Distributor (Weight-stationary Architecture)

Because this architecture is weight-stationary, the full set of weights is stored in the SRAM. The configuration inputs include image size and filter dimension, this information is shared with the image distributor through a helper core. The number of distributor cores depends on how many sets of weights are processed in parallel on one chip. For example, four sets of weights are processed in parallel for layer 2, so four distributor cores are used, as shown in Figure 5.12. The outputs of the weights distributor go directly to the 2D convolution modules in the computation stage.

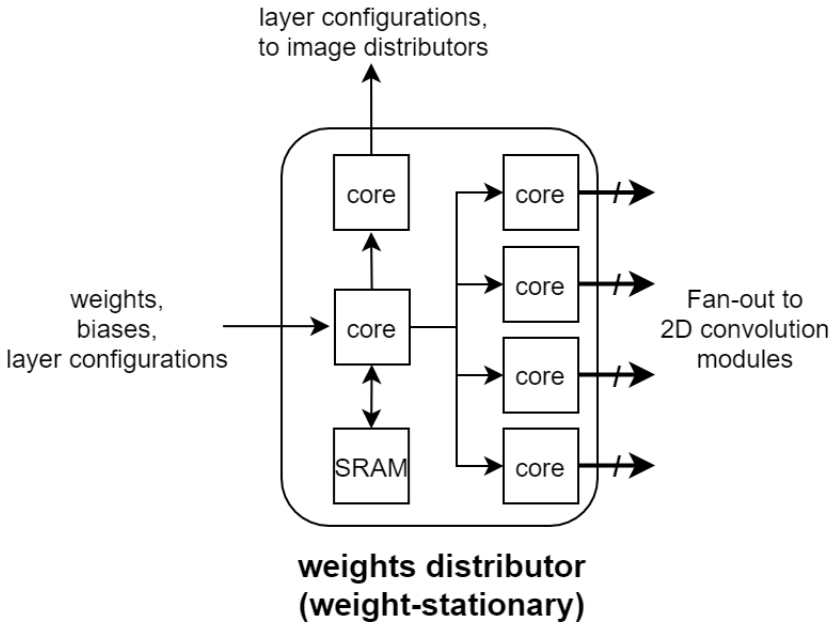

Figure 5.12: Weights distributor, weight-stationary architecture. Four distribution cores are used because four 3D convolution modules are processing in parallel. SRAM is saving the full weights/biases for the current layer.

Algorithm 5 weights distributor (weight-stationary architecture)

```
weights_biases_dimension ← INPUT
weights_biases ← INPUT
config ← INPUT
weights_biases → SRAM                                   ▷ save full set
config → OUTPUT to image distributor
loop(num_weights/4)                                     ▷ parallel outputs
    SRAM → weights_biases → OUTPUT port 0
    SRAM → weights_biases → OUTPUT port 1
    SRAM → weights_biases → OUTPUT port 2
    SRAM → weights_biases → OUTPUT port 3
end loop
```

5.4.5 Weights Distributor (Input-stationary Architecture)

The weights distributor in image-stationary architecture has a simple structure, because they only need to buffer four weights at a time and distribute them to the 2D convolution modules.

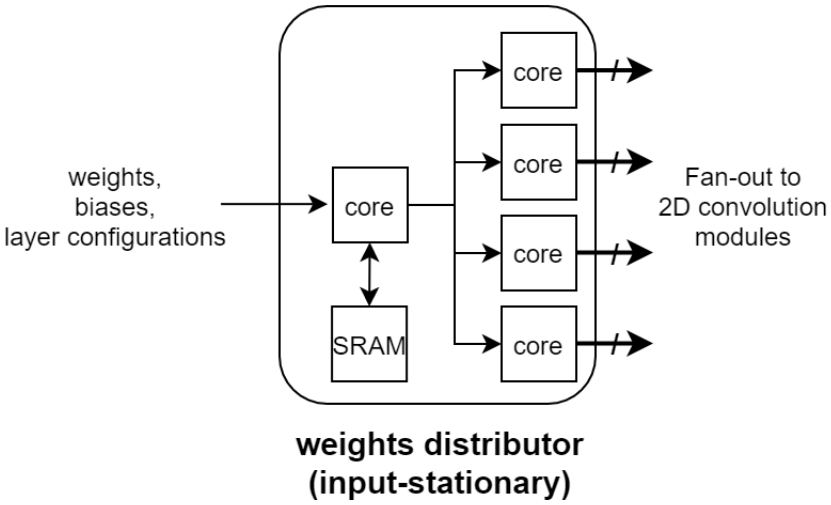

Figure 5.13: Weights distributor, input-stationary architecture. SRAM is buffering four weights/biases at a time, then distribute them in parallel to the 2D convolution modules.

The only difference compared with the other weights distributor is that the configuration inputs are no longer shared with the image distributor.

```
Algorithm 6 weights distributor (input-stationary architecture)
1: weights_biases_dimension ← INPUT
2: loop(num_weights/4)                                          ▷ parallel outputs
3:     SRAM ← weights_biases ← INPUT                              ▷ buffer 4 sets
4:     SRAM → weights_biases → OUTPUT port 0
5:     SRAM → weights_biases → OUTPUT port 1
6:     SRAM → weights_biases → OUTPUT port 2
7:     SRAM → weights_biases → OUTPUT port 3
8: end loop
```

5.4.6 2D Convolution Module (Inside 3D Convolution Module)

Intuitively, one 3x3 filter is convoluted with one channel of the input image inside the 2D convolution module. Since this is the most repeated operation for the entire Convolutional Neural Network, parallel optimization inside this module will offer significant performance improvements for the whole program. To calculate a 3x3 convolution window requires nine MAC operations. Instead of using one core to do MAC nine times, three cores are used here to do MACs in parallel, so each core only needs to do MAC three times. One distribution core is in charge of flattening the 3x3

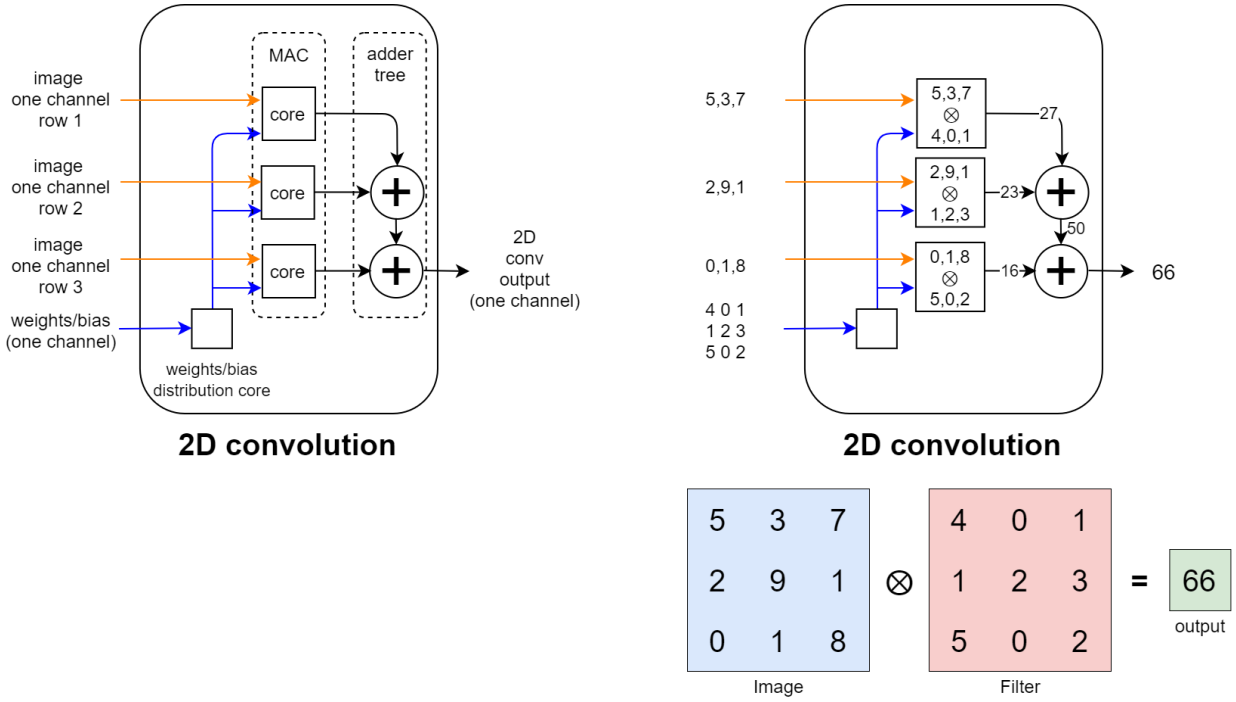

Figure 5.14: 2D convolution module. Example dataflow is showing on the right.

filter into three 1x3 filters, then after the MAC stage, partial convolution results are combined in the adder tree to form the output pixel of that convolution window.

- implementation note: Overflow handling

  16-bits fixed-point multiplication will generate a 32-bits wide result, since the MAC module is 40-bits wide, overflow will not happen during the MAC stage. However, extra logic to prevent overflow is needed inside the adder tree. If two positive numbers are added and the result overflows to a negative number, this means a pixel that is supposed to be activated will be falsely deactivated later by the Leaky-ReLU function. Similarly, a negative pixel that overflows to a positive pixel will be falsely activated by the Leaky-ReLU function. Algorithm 8 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 7 2D convolution

```
image_dimension ← INPUT
weights_dimension ← INPUT
loop(i < image_depth)      ▷ following the output dimension of image distributor
    loop(j < image_height+2)                                  ▷ image with padding
```

```
5:          loop(k < image_width+2)                              ▷ image with padding
6:              image_vector ← 3 rows of image ← INPUT
7:              weight_vector ← weight ← INPUT
8:              conv_result_partial = image_vector × weights_vector
9:              sum(conv_result_partial) → OUTPUT
10:         end loop
11:     end loop
12: end loop
```

Algorithm 8 overflow handling (saturation)

```
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 16-bit negative max
7:     else if (num1 is positive & sum is negative) then     ▷ pos + pos = neg,
   OVERFLOW!
8:         0x7FFF → OUTPUT                                  ▷ return 16-bit positive max
9:     end if
10: else
11:     sum → OUTPUT                                        ▷ overflow impossible
12: end if
```

5.4.7 3D Convolution Module

This module is called 3D convolution because it is adding multiple 2D convolution results along the 3rd dimension (channel-wise) to form one output feature map. The modular design of the 2D and 3D convolution blocks allows the program to be scaled up easily to increase performance. Specifically, the more 2D Conv modules used inside the 3D Conv module, the more image channels

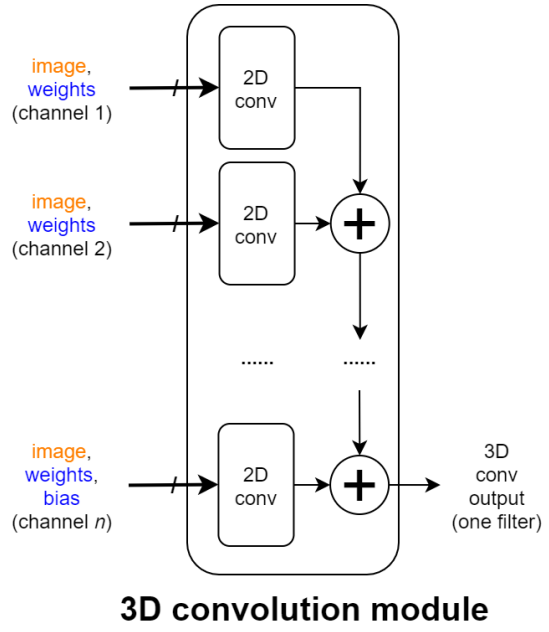

Figure 5.15: 3D convolution module. Each 2D conv module inside can process one image channel. So if there are 16 2D conv modules inside one 3D conv module ($n = 16$), then the 3D conv module can process 16 channels in one shot.

can be processed at once, and multiple 3D Conv modules can work in parallel to improve overall throughput.

5.4.8 Output Buffer

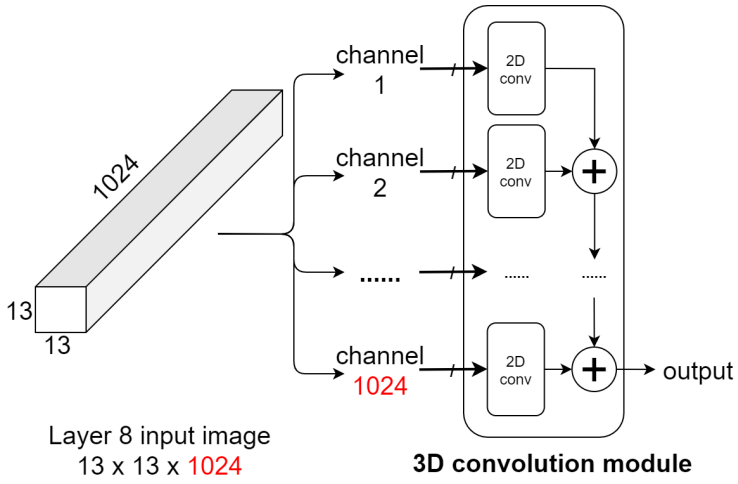

Figure 5.16: 3D deep convolution without output buffer. Layer 8 has 1,024 channels, but using 1,024 2D conv modules in one 3D conv module will cost at least 6,144 cores, which is impossible to place on 1 KiloCore 2 chip.

Theoretically, to offer the best performance and efficiency, the number of 2D Conv modules in one 3D Conv module should match the number of channels of the input image. But to ensure that a large convolution layer can work on a single KiloCore 2 chip with 697 cores, it's not realistic to use all the resources just for one module. For example, as shown in Figure 5.16 the input image to layer 8 has a dimension of 13×13×1024. Since one 2D Conv costs 6 cores, a 3D Conv module that can process 1024 channels will require at least 6,144 cores, which far exceeds the number of available cores on the KiloCore 2.

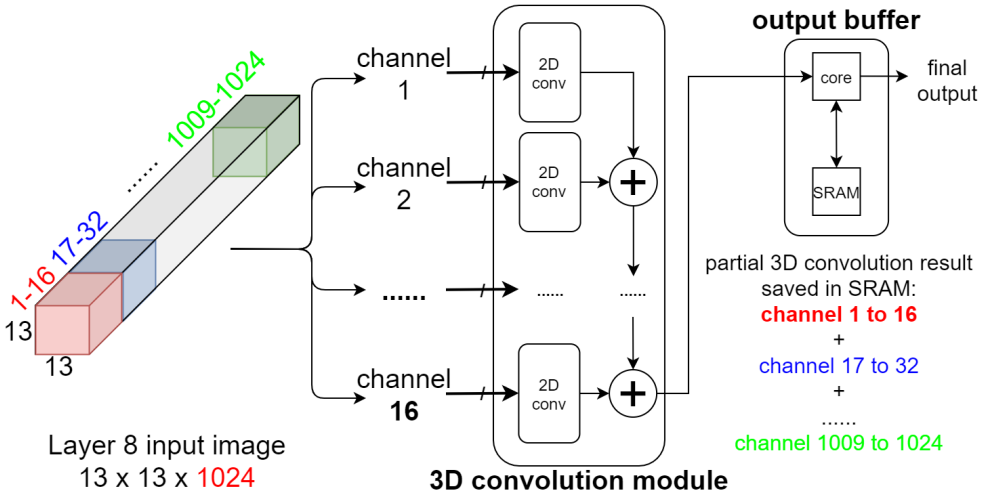

Figure 5.17: 3D deep convolution with output buffer. The 3D conv module can process 16 channels at a time, and the partial 3D conv output is saved inside the output buffer.

The output buffer shown in 5.17 is designed to solve this problem by saving the partial 3D Convolution result, so that the 3D Conv module can use fewer hardware resources and looping through all the channels of a large input image. This also separates all the post-processing tasks including activation and maxpooling from the convolution modules, which makes the architecture more flexible in adapting to other CNNs with different activation functions and maxpooling configurations.

- Implementation note: Leaky-ReLU activation

$$f(x) = \begin{cases} x & \text{if x>0,} \\ 0.1x & \text{otherwise.} \end{cases} \tag{5.1}$$

Algorithm 9 Leaky-ReLU branchless implementation (SLOW ON KILOCORE 2)

```
1: x ← INPUT
2: max(0, x)+0.1×min(0, x) → OUTPUT
```

Algorithm 10 Leaky-ReLU straightforward implementation (FAST ON KILOCORE 2)

```
1: x ← INPUT
2: if x ≥ 0 then
3:     x → OUTPUT
4: else
5:     x×0.1 → OUTPUT
6: end if
```

It is important to design software with the underlining hardware infrastructure in mind, especially when designing software for specialized architecture like KiloCore 2. One common technique for code optimization is branch removal, as shown in algorithm 9. The straightforward implementation of Leaky-ReLU is shown in algorithm 10. For complex hardware with multi-threaded core and out-of-order execution capabilities, the branchless algorithm can achieve high performance. However, in practice, algorithm 10 beats algorithm 9 when running on KiloCore 2. This can be explained from two aspects. First, the processors in KiloCore 2 have a simple datapath for high efficiency, so the cost associated with function calls is higher than the other typical general-purpose processors. Second, the static branch predictor of KiloCore 2 is helping the processor pipeline to use minimal ALU operations. Even if a misprediction occurs, the cost is still low because the logic for each branch is very simple.

- implementation note: maxpool

Algorithm 11 maxpool

```
1: loop(i <image_height)
2:     loop(j <image_width)              ▷ 2x2 maxpool window, read 4 elements
3:         window_0 ← image row i, element j ← SRAM
4:         window_1 ← image row i, element j+1 ← SRAM
```

```
5:          window_2 ← image row i+1, element j ← SRAM
6:          window_3 ← image row i+1, element j+1 ← SRAM
7:          max(window_0, window_1, window_2, window_3) → OUTPUT
8:      end loop
9: end loop
```

Following the concept described in the Leaky-ReLU implementation, the maxpool algorithm is also simple and straightforward. Since the output buffer already have the full output image saved in the attached SRAM, the maxpooling process is just successively reading the maxpool windows from SRAM and selecting the maximum number to send to the output port. YOLOv3-Tiny always uses 2x2 maxpool window with stride equals to 1, so the cost of producing each maxpool output is three operations (subtraction).

5.4.9 Core Level Summary

Concretely, the core level implementations closely follow the mathematical operations of a Convolutional Neural Network, especially from a dimensionality point of view. The dimension of the inputs are flattened in the distribution modules, then the convolution modules rebuild the output dimension from 1D to 2D, according to the configuration of each layer. Finally, multiple 3D modules work in parallel to add the 3$^{\text{rd}}$ dimension to the output matrix.

The weight-stationary architecture uses a maximum of 143 cores on KiloCore 2, and the input-stationary architecture costs 536 cores. Depending on different layer configurations, the actual core count might be lower. For example, layer 10 and layer 11 are using input-stationary architecture but they use only 248 cores. Because the filter dimension for these two layer are all 1×1, so they don't require too many 2D Conv modules to perform convolution. Table 5.2 gives an overview of the code structure for all the standalone modules, and Table 5.3 gives a summary of the core counts.

Core Level Blocks	**Functions**	**C++ Code Number Of Lines**	**Number of Assembly Instructions**
image distributor (weight-stationary architecture)	image_distributor	70	119
	input_divider	51	73
	image_1_input_to_4_output	7	6
	image_1_input_to_3_output	7	6
image distributor (input-stationary architecture)	SRAM_controller	55	99
	image_4_input_to_16_output	79	116
	image_8_input_to_4_output	48	71
	image_distributor	35	40
	image_input	116	174
	image_to_2D_conv	7	6
weights distributor (weight-stationary architecture)	weight_1_input_to_4_output	46	62
	weight_to_MAC	42	57
	weight_distributor	57	92
weights distributor (input-stationary architecture)	weight_1_input_to_8_output	50	78
	weight_distributor	69	104
2D convolution module	weight_to_MAC	39	49
	MAC_0	69	134
	MAC_1	69	134
	MAC_2	72	134
	conv_adder_final_stage	61	64
	conv_adder_first_stage	58	63
3D convolution module	channel_adder_final_stage	64	64
	channel_adder_first_stage	61	65
	channel_adder_middle_stage	56	60
output buffer	output_buffer	90	124

Table 5.2: Summary of the code structure for all the code level blocks

	# Cores	# SRAMs
Image Distributor (Weight-stationary Architecture)	4	1
Image Distributor (Input-stationary Architecture)	31	8
Weights Distributor (Weight-stationary Architecture)	6	1
Weights Distributor (Input-stationary Architecture)	5	1
2D Convolution Module (Inside 3D Convolution Module)	6	N/A
3D Convolution Module	(n×6)+(n-1)	N/A
Output Buffer	1	1

Table 5.3: Core counts for all the basic modules. n equals to the number of 2D conv modules inside one 3D conv module.

Figure 5.18 and 5.19 show the actual error-free mapping of the weight-stationary architecture on KiloCore 2, produced by the mapper of Project Manager, the KiloCore 2 simulator/Compiler.

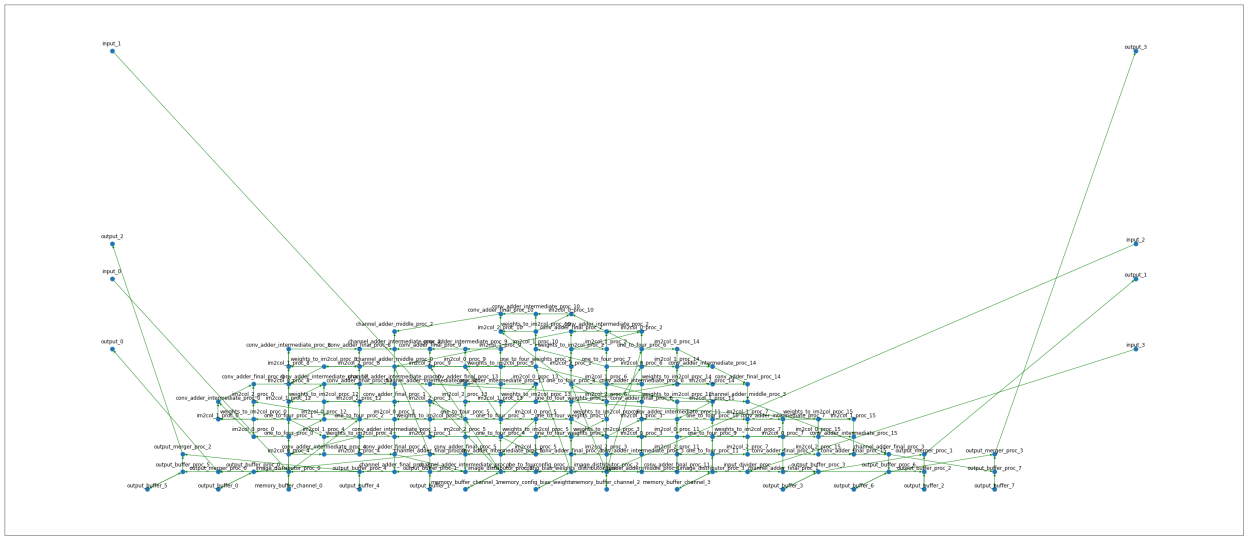

Figure 5.18: Fly line diagram of the weight-stationary architecture, each dot is a processor core, and the green lines indicate inter-core connection.

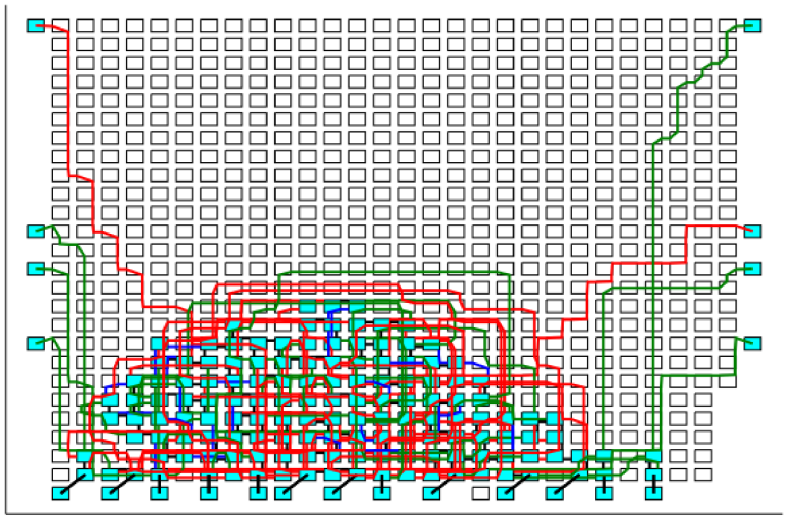

Figure 5.19: The complete error-free weight-stationary architecture mapping to the manycore processor array. The various link colors signify different types of communication links: nearest neighbor, long distance, or packet network.

Similar to the previous Figures, Figure 5.20 and 5.21 show the actual error-free mapping of the input-stationary architecture on KiloCore 2, produced by the mapper of Project Manager, the KiloCore 2 simulator/Compiler.

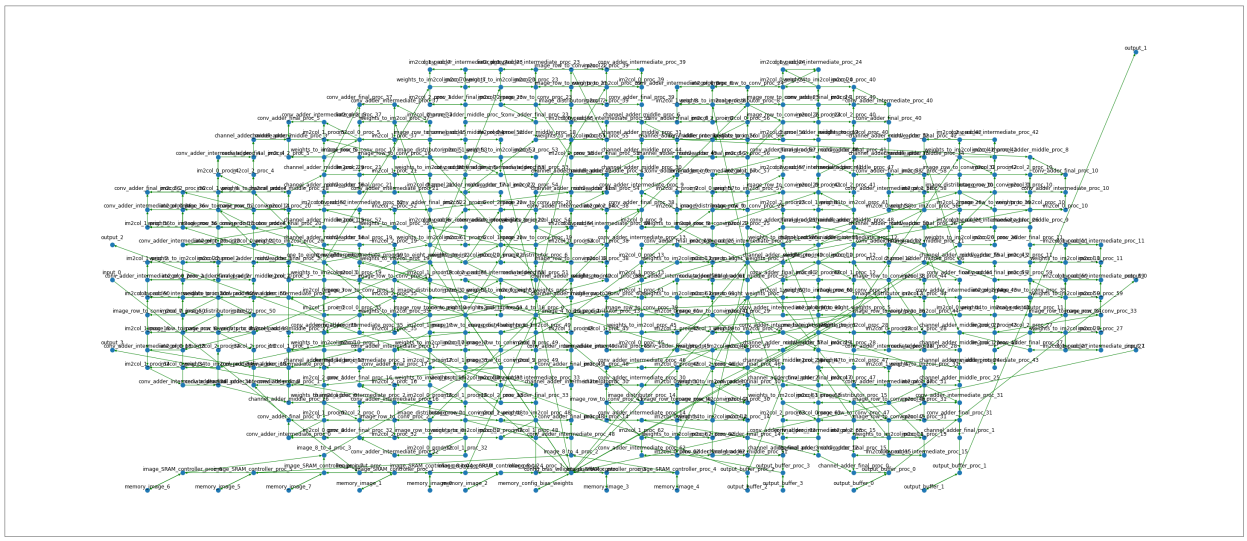

Figure 5.20: Fly line diagram of the input-stationary architecture, each dot is a processor core, and the green lines indicate inter-core connection.

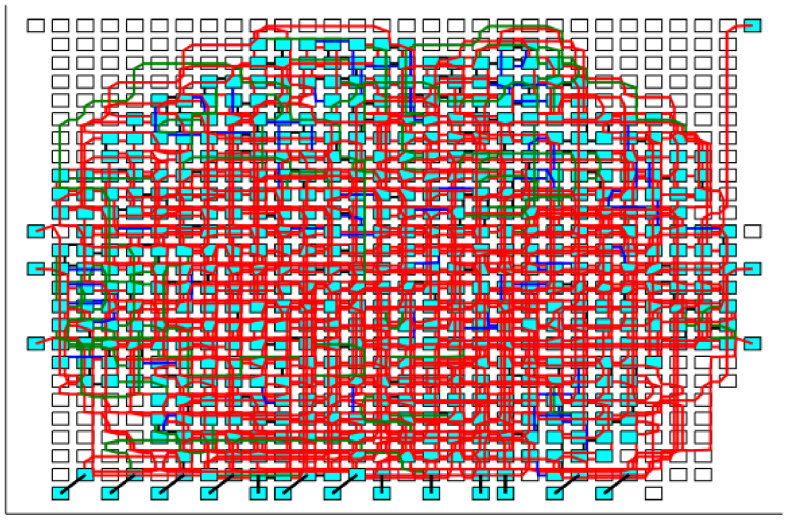

Figure 5.21: The complete error-free input-stationary architecture mapping to the manycore processor array. The various link colors signify different types of communication links: nearest neighbor, long distance, or packet network.

Chapter 6

Inference Evaluation

6.1 Overview

The functionality of the two chip level inference architectures on KiloCore 2 has been verified by using random inputs generated by MATLAB. However, in order to show that the optimized YOLOv3-Tiny implementation in this work is still able to accurately perform Object Detection, a real-world image is used as an input to test its performance, as shown in Figure 6.1. The image is quantized to 16-bit fixed-point and the final prediction output is compared with the original YOLOv3-Tiny implementation working in full 32-bit floating-point accuracy.

Figure 6.1: Test image used in this section with three detections expected: dog, person, and horse.

All simulation data are obtained with Project Manager, a cycle-accurate C++ simulator for the KiloCore 2 platform. The simulator also generates accurate power and throughput measurements, which can be used to compare with other hardware platforms like GPU and general-purpose CPU.

For this section, all the fixed-to-float (or float-to-fixed) conversions and data visualizations are made with code written in MATLAB.

6.2 Inference Result

A MATLAB implementation of the original YOLOv3-Tiny system is used as the Golden Reference for comparison. The MATLAB implementation closely follows the version written in C by the original author and uses the unaltered 32-bit floating-point parameters, this guarantees that any potential errors in the KiloCore 2 implementation won't be repeated in the Golden Reference. Outputs are gathered and compared layer by layer, table 6.1 shows the mean error and max error for each layer, calculated with equation 6.1 and 6.2.

$$mean\ error = \texttt{mean}(|image_{\ Golden\ Reference} - image_{\ KiloCore\ 2\ output}|) \quad \text{(element-wise)} \tag{6.1}$$

$$max\ error = \texttt{max}(|image_{\ Golden\ Reference} - image_{\ KiloCore\ 2\ output}|) \quad \text{(element-wise)} \tag{6.2}$$

Layer	1	2	3	4	5	6	7	8	9	10	11	12	13
mean	0.0071	0.0282	0.0346	0.0504	0.0969	0.0552	0.25	0.0947	0.3262	1.0032	0.2609	0.236	1.2051
max	0.1263	0.2215	0.3247	0.4628	0.8898	4.8365	19.07	3.1055	2.9977	8.464	5.9827	4.1391	10.1982

Table 6.1: Mean and max error for each layer compared to the Golden Reference

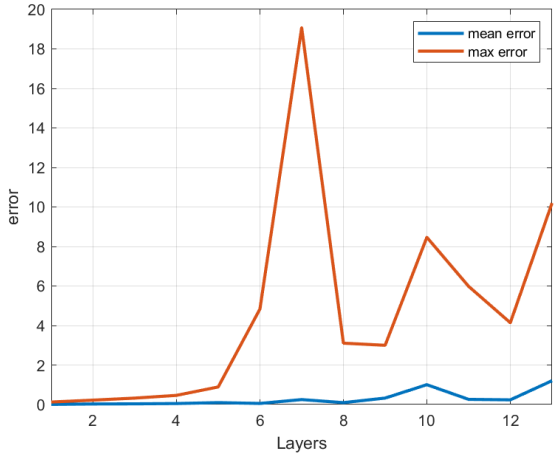

Figure 6.2: Mean and max error plot for each layer.

Figure 6.3 gives a visual representation of the outputs of each layer. The output images are in greyscale with the positive number showing in light pixel and the negative number showing in dark pixel. Intuitively, a light pixel means that the convolution filter thinks this pixel contains important information, and it is being extracted by the activation function [35]. As the output image on layer 10 and layer 13 shows, three clusters of filters are heavily activated, which correspond to the three predictions: horse, person, and dog.

The final detection output is converted to human-readable bounding boxes shown in Figure 6.4 along with the golden reference detection output. The detections produced by this thesis show that all the predicted categories are correct, and the confidence errors are within 0.01 compared to the golden reference. The box dimensions are slightly affected due to the lower precision of the weights used, but the predicted centers of the objects are still accurate. In conclusion, the result proves that this work has implemented the CNN calculations correctly, and all the optimization techniques used including quantization and BN folding did not degrade the accuracy and performance of the YOLOv3-Tiny system.

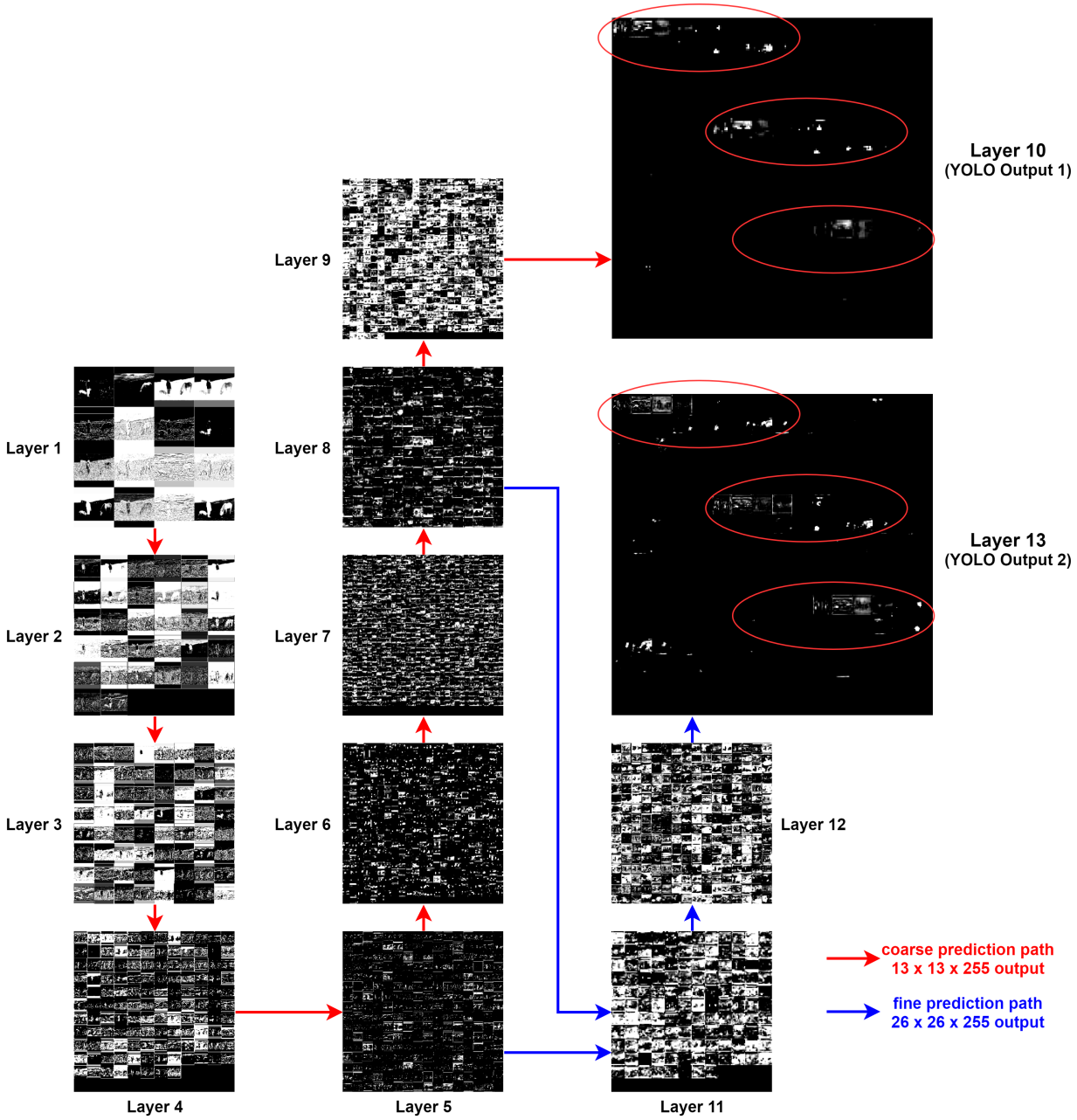

Figure 6.3: Output image of each layer. Red arrow indicates data flow for generating the coarse prediction in 13 × 13, Blue arrow indicates data flow for generating the fine prediction in 26 × 26

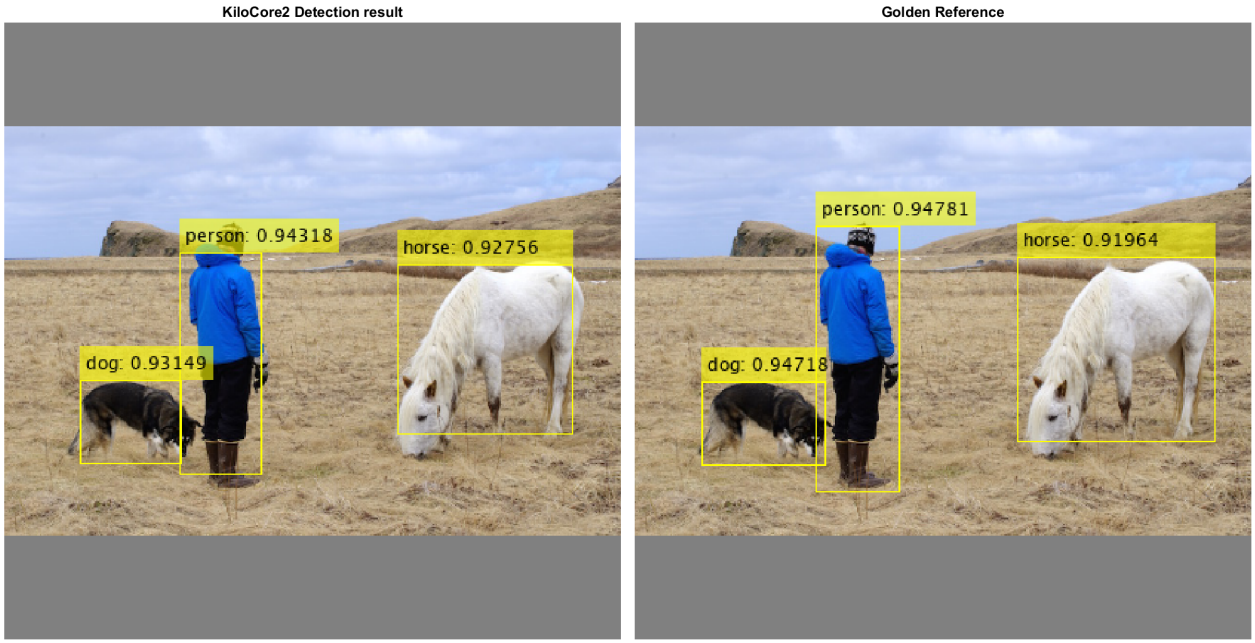

Figure 6.4: Final detection output. Left side is the result generated by KiloCore 2, right side is the Golden Reference output produced by the original YOLOv3-Tiny system.

6.3 Simulation Measurements

The simulation environment is set to limit available resources to one KiloCore 2 chip, so the result can be reproduced on the real KiloCore 2 Test System board. The maximum available core count on one KiloCore 2 chip is 697, with 14 SRAM modules, and 4 complex 16-bit input/output ports. Energy measurements from the 32 nm PD-SOI CMOS fabricated chip are used as inputs to the simulator to obtain energy data. Area usage is physically measured from the fabricated 32 nm PD-SOI CMOS chip, where each processor occupies 265 μm $\times$ 274.5 μm of area and each SRAM memory occupies 356.2 μm $\times$ 475.41 μm of area. The total area used is calculated by equation 6.3, where $nProc$ and $nMem$ are the maximum number of processors and memory modules used. For this work, $nProc$ is 536 and $nMem$ is 13.

$$Area(mm^2) = nProc \times 0.0727(mm^2) + nMem \times 0.169(mm^2) \tag{6.3}$$

Throughput is calculated as the reciprocal of the total latency to process 1 image for the CNN implementation, using equation 6.4. This property is also called Frame rate, or frames per second (FPS). Latency is calculated as the sum of the processing time of each layer, using equation

6.5.

$$Throughput(FPS) = \frac{1}{total\ latency(s)} \tag{6.4}$$

$$Total\ Latency(s) = (\sum_{i=1}^{13} LastOutputTime_i\ (ps)) \times 10^{12} \tag{6.5}$$

Power is calculated as in equation 6.6, where *Energy* is the sum of the total energy consumed by each layer, using equation 6.7.

$$Power(W) = \frac{Energy(J)}{total\ latency(s)} \tag{6.6}$$

$$Energy(J) = (\sum_{i=1}^{13} TotalEnergy_i\ (nJ)) \times 10^{9} \tag{6.7}$$

Throughput per area is calculated as the throughput (FPS) divided by the die area in mm^2 and is given by equation 6.8.

$$Throuput\ per\ Area(FPS/mm^2) = \frac{Throughput(FPS)}{Area(mm^2)} \tag{6.8}$$

Throughput per watt is calculated as the throughput (FPS) divided by power (W) and is given by equation 6.9.

$$Throuput\ per\ Watt(FPS/W) = \frac{Throughput(FPS)}{Power(W)} \tag{6.9}$$

The Energy-Delay Product (EDP) is calculated as the product between latency and the energy consumed per image as shown in equation 6.10. A lower Energy-Delay Product means the system is more efficient.

$$EDP = Energy(J) \times Latency(s) \tag{6.10}$$

Total memory requirement is calculated by summing the run-time SRAM space and the off-chip DRAM space. The DRAM needs to save the input image with all the weights and biases, and also have enough space to buffer the largest layer's output, which is 208×208×16 at layer 1.

$$
\begin{aligned}
&Memory\ Requirement(MB) \\
&\quad = (SRAM\ requirement) + (DRAM\ requirement) \\
&\quad = (SRAM\ used \times 64KB) + (input\ image + Weights/Biases + layer\ buffer) \\
&\quad = (13 \times 64KB) + (1014KB + 17.72MB + 1352KB) \\
&\quad = 19.53MB
\end{aligned}
\tag{6.11}
$$

Area (mm^2)	Latency (s)	Throughput (FPS)	Power (W)	Energy (J)	Throughput/Area (FPS/mm^2)	Throughput/Watt (FPS/W)	EDP
41.16	2.424	0.4126	0.9209	2.232	0.01002	0.4481	5.409

Table 6.2: Performance data of this work.

Layer	Active Energy (nJ)	Total Energy (nJ)	First output time (ps)	Last output time (ps)	Cores used
1	51,293,292	64,949,252	64,676,619	96,302,349,479	102
2	136,491,464	160,473,595	21,842,347,065	182,888,582,793	143
3	140,250,749	159,881,252	9,185,674,260	149,702,903,210	143
4	141,079,995	148,579,141	4,939,071,081	153,652,724,316	536
5	141,902,225	149,588,950	2,763,278,413	157,496,095,519	536
6	146,450,814	154,564,407	1,541,100,813	166,241,530,945	536
7	581,363,490	613,790,917	3,135,402,837	664,417,462,499	536
8	58,353,187	68,652,293	2,311,330,098	94,370,481,702	248
9	146,346,534	154,458,706	1,540,379,435	166,213,257,259	536
10	29,267,257	34,415,456	1,135,646,393	47,172,814,157	248
11	7,359,007	8,656,389	547,650,852	11,887,837,962	248
12	423,260,703	444,564,230	7,996,207,737	436,495,761,828	536
13	58,718,848	69,279,358	2,229,646,118	96,765,724,888	248
Total	-	2,231,853,946	-	2,423,607,526,557	-

Table 6.3: Detailed simulation measurements of each layer.

Chapter 7

Performance Comparison with Other Hardware Platforms

7.1 Overview

The metrics used for comparison are throughput per area, throughput per watt, work per energy, and run-time memory requirement, they are chosen to show performance, power-efficiency, and memory-efficiency of the manycore implementation on KiloCore 2, respectively. GPUs, specialized CNN ASICs, and general-purpose CPUs are included for an exhaustive comparison across all platforms.

7.2 Comparison of YOLOv3-Tiny Manycore Implementation with Other Platforms

Because different fabrication technologies can significantly affect performance and power, all the measurements used for comparison are scaled to 32nm to match KiloCore 2's manufacturing process. The scaling method uses predictive polynomial models with table-based coefficients, which produces an accurate scaling factor of CMOS device performance between different technology nodes [5] [36].

$$DelayFactor = a_{d3}V^3 + a_{d2}V^2 + a_{d1}V + a_{d0} \tag{7.1}$$

$$EnergyFactor = a_{e2}V^2 + a_{e1}V + a_{e0} \tag{7.2}$$

The scaling factors are calculated using equation 7.1 and 7.2 with the coefficients provided in table 7.1, 7.2 from [5].

process	a_{d3}	a_{d2}	a_{d1}	a_{d0}	delay factor
32nm LP @ 0.9V	-325.9	1374	-1922	913.2	58.7589
20nm HP @ 1V	0	34.63	-66.37	41.15	9.41
16nm HP @ 1V	0	24.8	-47.52	28.87	6.15
14nm HP @ 1V	-40.66	109.2	-100.6	35.92	3.86
10nm HP @ 1V	-34.95	93.65	-85.99	30.4	3.11

Table 7.1: The polynomial coefficient values and delay factors calculated with equation 7.1. [5]

process	a_{e2}	a_{e1}	a_{e0}	energy factor
32nm LP @ 0.9V	0.9559	-0.7823	0.471	0.541209
20nm HP @ 1V	0.373	-0.1582	0.04104	0.25584
16nm HP @ 1V	0.2958	-0.1241	0.03024	0.20194
14nm HP @ 1V	0.2363	-0.09675	0.02239	0.16194
10nm HP @ 1V	0.2068	-0.09311	0.02375	0.13744

Table 7.2: The polynomial coefficient values and energy factors calculated with equation 7.2. [5]

The scaled data is calculated using equations 7.3, 7.4, 7.5, and 7.6. For area scaling, the factors are given in table 7.3.

$$Area_x = AreaFactor_y \times Area_y \tag{7.3}$$

$$Delay_x = \frac{DelayFactor_x}{DelayFactor_y} \times Delay_y \tag{7.4}$$

$$Energy_x = \frac{EnergyFactor_x}{EnergyFactor_y} \times Energy_y \tag{7.5}$$

$$Power_x = \frac{EnergyFactor_x \cdot DelayFactor_y}{EnergyFactor_y \cdot DelayFactor_x} \times Power_y \tag{7.6}$$

process	scale factor
32nm	1
20nm	2.2
16nm	2.4
14nm	2.7
10nm	4.5

Table 7.3: Factors used for area scaling [5].

Table 7.4 gives the performance measurements across all hardware platforms, scaled to 32nm.

	KiloCore 2	Jetson NANO	Jetson TX2	Core i3-8145U	Core i5-8365U	Core i7-8665U	Core i9-9900K
Area (mm^2)	**41.16**	259.6	N/A*	553.5	N/A*	N/A*	486
Latency per image (ms)	2424	367.3	212.3	944.7	608.9	585.5	**205.7**
Throughput (FPS)	0.4126	2.722	4.710	1.059	1.642	1.708	**4.861**
Power per image (W)	**0.9209**	55.70	178.5	1042	381.6	381.6	2417
Energy per image (J)	**2.232**	20.46	37.89	984.0	232.3	223.4	497.1
Throughtput per Area (FPS/cm^2)	1.002	**1.049**	N/A*	0.1912	N/A*	N/A*	1.000
Throughtput per Watt (FPS/W)	**0.4481**	0.04887	0.02639	0.001016	0.004304	0.004476	0.002012
energy×delay (J*ms)	**5.409**	7.515	8.045	929.5	141.5	130.8	102.3

Table 7.4: Performance data for all hardware platforms [6] [7] [8] [9] [10]. CPU Power is assumed to be one half of the TDP. Memory area for the GPUs and CPUs is not included in this data because it is publically unavailable, however memory area is included for the KiloCore 2 implementation.
*No public data available

7.2.1 Performance

Table 7.5 shows the performance (throughput per area) data of various platforms, comparing with KiloCore 2. A visualization is shown in Figure 7.1.

	KiloCore 2	Jetson NANO	Core i3-8145U	Core i9-9900K
Throughtput/Area(FPS/cm^2)	1.002	**1.049**	0.1912	1.000
Normalized Throughtput/Area	5.24	**5.49**	1	5.23

Table 7.5: Throughput per area comparison. Data taken from table 7.4. Memory area for the GPUs and CPUs is not included in this data because it is publically unavailable, however memory area is included for the KiloCore 2 implementation.

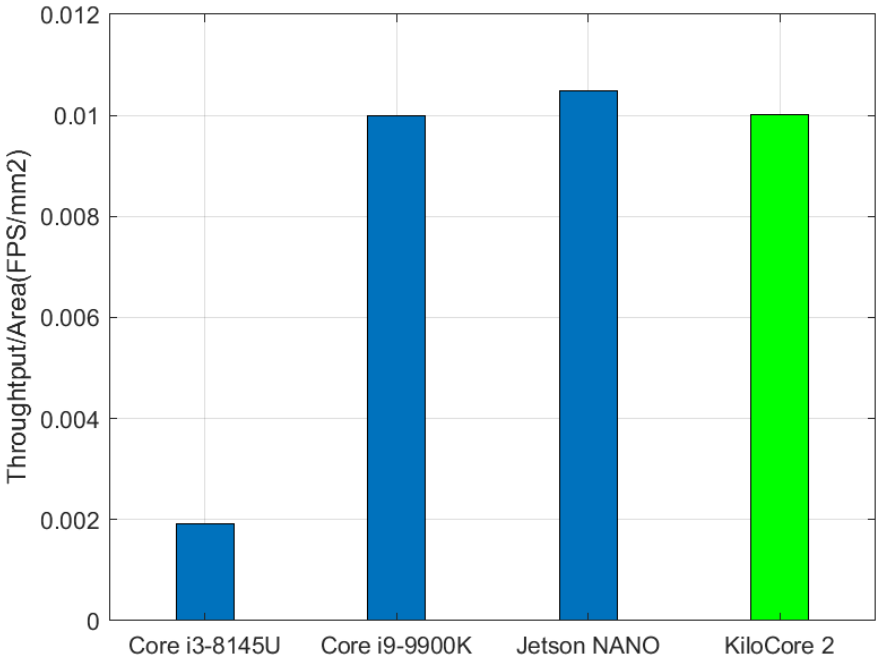

Figure 7.1: Throughput per area comparison. Memory area for the GPUs and CPUs is not included in this data because it is publically unavailable, however memory area is included for the KiloCore 2 implementation.

Comparing with the GPUs, this implementation can achieve the same high-performance level in terms of throughput per area. If comparing with the general-purpose CPU, KiloCore 2 can offer a 5.24× performance improvement. This shows that the manycore structure of KiloCore 2 is suitable for processing Convolutional Neural Networks. Similar to GPUs, the large pool of cores on KiloCore 2 can provide massive parallelism for accelerating matrix-related tasks.

7.2.2 Power/Energy-Efficiency

Table 7.6 shows the throughput per watt and Energy Delay Product (EDP) of various platforms, comparing with KiloCore 2. A visualization of throughtput per watt is shown in Figure 7.2.

	KiloCore 2	Jetson NANO	Jetson TX2	Core i3-8145U	Core i5-8365U	Core i7-8665U	Core i9-9900K
Throughtput /Watt(FPS/W)	**0.4481**	0.04887	0.02639	0.001016	0.004304	0.004476	0.002012
Normalized Throughput/Watt	**441**	48.1	26.0	1	4.24	4.41	1.98
energy×delay (J*ms)	**5.409**	7.515	8.045	929.5	141.5	130.8	102.3
Normalized energy×delay	**1**	1.39	1.49	172	26.2	24.2	18.9

Table 7.6: Throughput per watt and EDP comparison. Data taken from table 7.4

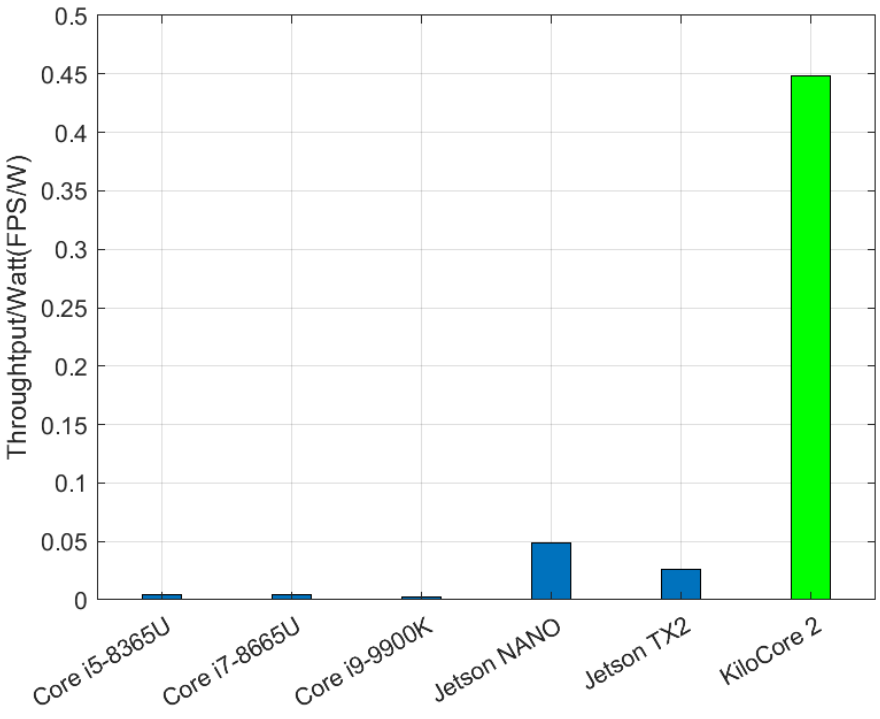

Figure 7.2: Throughput per watt comparison. Core i3-8145U not included due to value too small.

As the throughput per watt comparison shows, at the same wattage rating, this work can offer an impressive 9.17× to 16.98× performance boost compared with the mainstream low-power GPU accelerators. Among the different level of consumer CPUs, KiloCore 2 beats the highest-performance Core i9 by 222.73×, demonstrating remarkable power-efficiency.

Figure 7.3 gives an illustration of the EDP comparison of the various platforms. Since EDP is combining energy consumption and latency together, it becomes an ideal metric to show the energy-efficient of a system. EDP can become very large for a fast system if the system is consuming a lot of energy to increase its processing speed. Conversely, a system can reduce the energy consumption by lowering its clock speed, but this will cause the latency to increase, again making the EDP grow. However, as the data shows, this work demonstrates a moderate 1.39× to 1.49× EDP advantage compared with the GPUs, and again beating the CPUs by a large margin, from 18.90× to 171.84×.

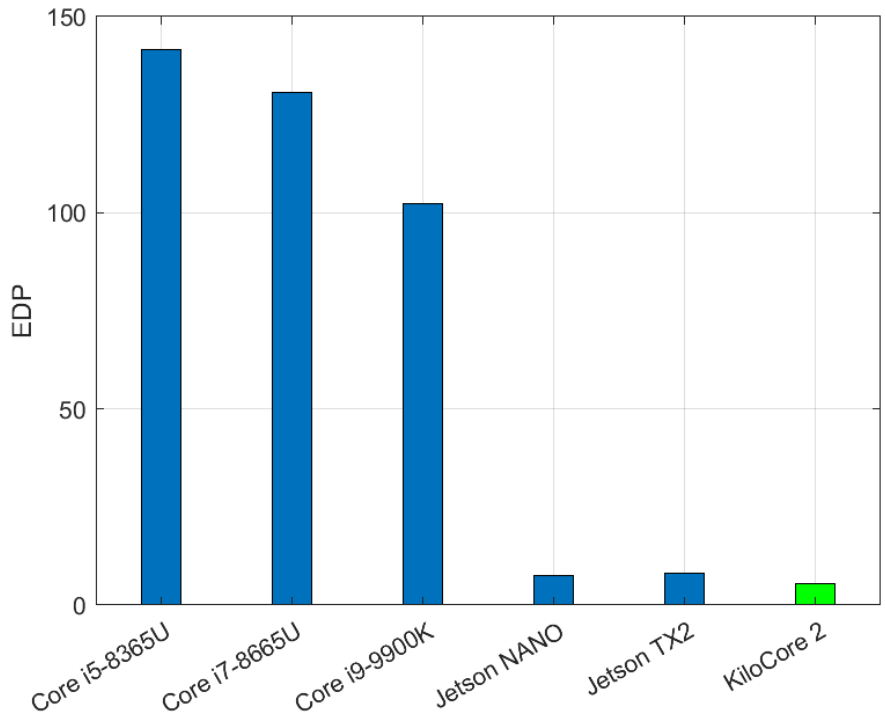

Figure 7.3: EDP comparison. Core i3-8145U not included due to value too large.

7.2.3 Memory-Efficiency

Table 7.7 shows the run-time memory requirement data of various platforms, comparing with KiloCore 2. A visualization of the same data is shown in Figure 7.4.

	KiloCore 2	Jetson Nano	Jetson TX2	Jetson Xavier NX	Raspberry pi 3 with Intel Neural Compute Stick 2	Google Edge TPU Dev Board
Memory requirement (MB)	**19.53**	966	1044	1265	DNR*	DNR*
Normalized memory requirement	**1**	49.5	53.5	64.8	N/A	N/A

Table 7.7: Memory requirement comparison. Data taken from table 7.4, [8], and [11].
*DNR = did not run, caused by limited memory capacity, unsupported network layers, or hardware/software limitations [11].

Modern hardware and software implementations for Covolutional Neural Network inference relies heavily on run-time memory capacity and bandwidth. Not only that a large memory chip will increase power consumption, but it also adds more cost to the whole system, either due to more area on the PCB, or designing specialized high-bandwidth connection to the processor. As table 7.7 shows, the Intel Neural Compute Stick 2 and Google Edge TPU Dev Board are all popular low-power Machine Learning accelerator ASICs, but they all have DNR result for YOLOv3-Tiny. Although they can not even run the YOLOv3-Tiny Neural Network, they are included here to show the importance of memory capacity in an inference system.

For this work, the run-time memory requirement is reduced by 49.46× to 64.77×, comparing with the three GPU accelerators. So for the same performance in terms of throughput per area, KiloCore 2 is able to use significantly less memory area, which reduces system complexity and saves even more energy.

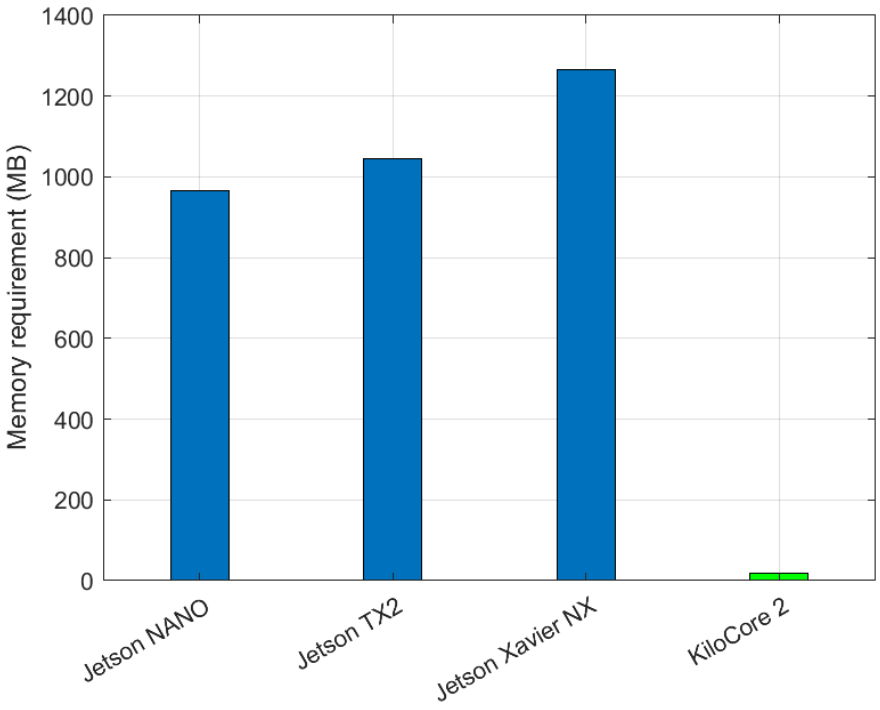

Figure 7.4: Memory requirement comparison.

7.3 Summary

First, as the throughput comparison shows, the YOLOv3-Tiny inference architectures implemented in this thesis can achieve a performance level that is on par with the modern state-of-the-art GPU accelerator and high-performance CPU. Second, this implementation working on the KiloCore 2 chip displays an impressive power efficiency that is 9.17× to 440.87× more efficient compared with the other hardware platforms in terms of throughput per watt. Finally, as the memory requirement suggests, mainstream CNN inference architectures depend heavily on large memories to offer data-level parallelism, which consumes more area on the whole PCB. In contrast, this thesis provides a manycore implementation that uses 49.46× to 64.77× less memory compared with the GPUs, while still being capable of running a large and deep Convolutional Neural Networks with the lowest EDP.

Chapter 8

Future Work

8.1 8-bit Quantization

More aggressive quantization techniques should be investigated to further reduce memory usage and increase performance. Compressing all the weights and biases from 32-bit to 8-bit will offer a 4× reduction in memory requirements, from 35.45MB down to 8.86MB. This also means that it will be possible to save the input image and all the weights together for one layer in the on-chip SRAM modules, which will increase the performance of the inference architectures.

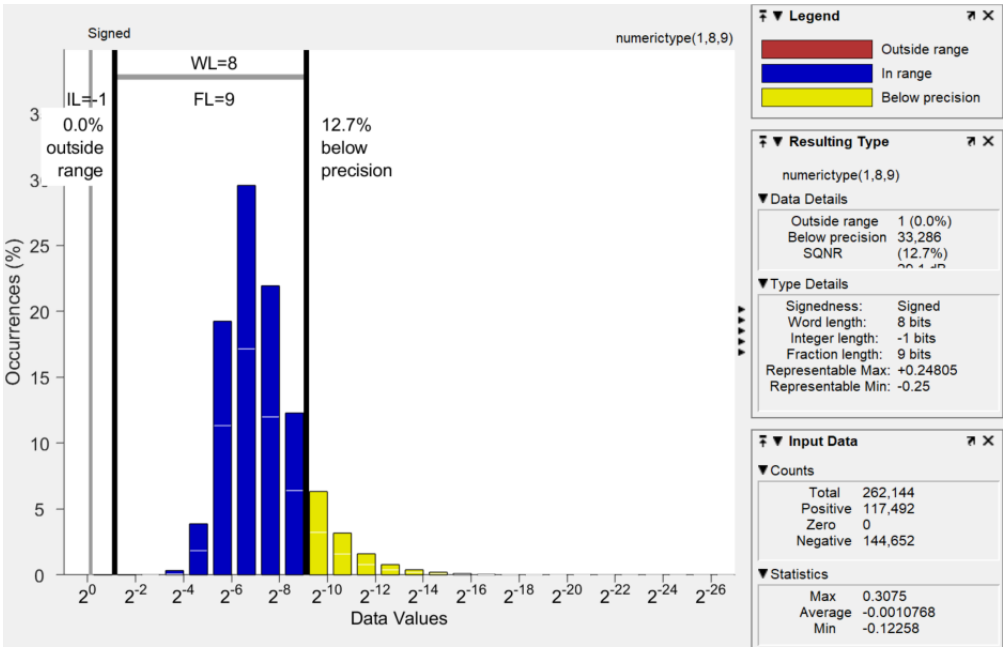

Figure 8.1: 8-bit weights of layer 8, the SQNR is 12.7% with all the bits used as fraction bits.

Previous research has shown that CNNs can still work accurately if they are quantized to extremely low precision like 8-bit [37] [38], and there exists software implementation of YOLOv3 in INT8 precision like [39] which proves that YOLOv3's functionality will not be affected by 8-bit quantization. A preliminary analysis applying 8-bit word length for layer 8 in YOLOv3-Tiny is shown in Figure 8.1. As the data suggests, precision loss caused by 8-bit quantization is still manageable and can be limited under 15%.

8.2 Alternative Mappings of YOLOv3-Tiny

For this thesis, the available hardware is assumed to be just one KiloCore 2 chip with 697 cores and 14 SRAM modules. However, since KiloCore 2 is based on the AsAP scalable manycore platform, more flexible mapping of the inference architectures should be explored. One direction to go is to consider a bigger chip with more cores, SRAM modules, and Input/Output ports. As illustrated in Figure 8.2, this design is aimed to improve the latency per image as more 3D convolution modules are used in parallel. No algorithmic changes are needed for this mapping strategy, and it does not consume more SRAM modules compared to the architectures implemented in the previous chapters.

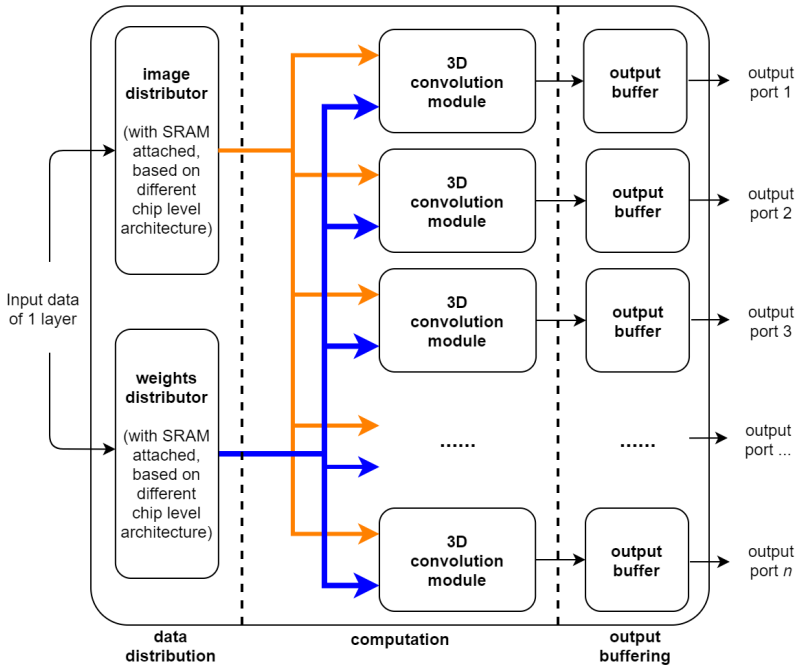

Figure 8.2: Bigger chip configuration using n output ports.

Another direction is to increase throughput by using multiple KiloCore 2 chips. Figure 8.3 shows a pipelined connection of 13 KiloCore 2 chips, with the critical path highlighted in red going through the two most computationally intensive layers: layer 7 and layer 12. The critical path is the slowest path in the implementation, which sets the throughput of the whole pipelined system. Using the simulation data in chapter 6, table 6.3, the throughput of this pipelined mapping is assumed to be 1.506 FPS (0.664 s), which is 3.65× better than the single chip implementation.

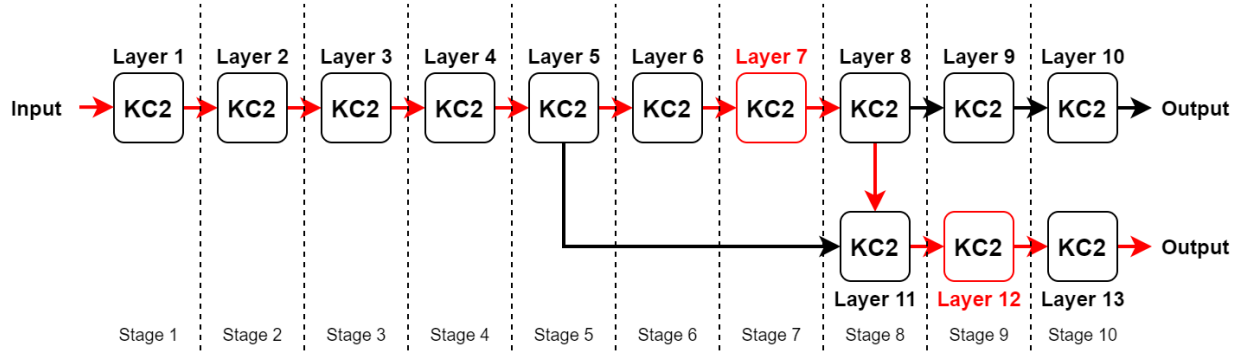

Figure 8.3: Pipelined connection of 13 KiloCore 2 chips. The critical path along with the two slowest layers are marked in red.

8.3 Quantization Aware Training

Quantization aware training is a technique used to improve the accuracy and efficiency of a quantized CNN during inference. The main idea is to introduce quantization during training so the CNN can optimize the weights for lower precision data [40]. For 16-bit quantization implemented in this thesis, more simulation with different input images should be performed in order to evaluate the overall accuracy. If using 8-bit quantization, this training strategy is likely required to make sure that the final detection result of YOLOv3-Tiny can preserve as much accuracy as possible.

Chapter 9

Thesis Summary

This thesis presents a high-performance, memory-efficient, and power-efficient Convolutional Neural Network inference implementation for the YOLOv3-Tiny Object Detection System on the KiloCore 2 manycore platform.

Chapter 1 gives the motivation of this project, which states that a manycore platform is best suited for running CNN inference and achieving a good power-performance balance. Chapter 2 and Chapter 3 introduce the software and hardware background of this work. Specifically, Chapter 2 explains why YOLOv3-Tiny is used, how it works, and how the CNN is constructed. Then Chapter 3 gives the hardware information of the KiloCore 2 chip, describes the architecture of its processors, memories, and programming workflow. The YOLOv3-Tiny CNN is pre-processed in Chapter 4, using BN folding and quantization to make it run more efficiently on the KiloCore 2 chip. All the implementation details, from the top-level dataflow to the core level algorithms, are given in Chapter 5. Finally, Chapter 6 evaluates the performance of the implementation, and Chapter 7 compares this work against a wide variety of hardware platforms. To improve this implementation in the future, some practical optimization techniques are introduced in Chapter 8.

This work provides 9.17× to 440.87× improvement over the other hardware platforms in terms of throughput per watt, at the same time offers the lowest EDP and still maintaining the same performance comparable to other high-power CPUs and specialized GPU accelerators in terms of throughput per area. The manycore implementation also consumes the least amount of memory among all platforms, which significantly reduces the overall PCB area of the hardware system, makes it suitable for mobile devices and edge-computing use cases.

Glossary

AsAP *Asynchronous Array of simple Processors.* A manycore platform designed by the VLSI Computation Lab at UC Davis.

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

BN Batch Normalization. A normalization layer between convolution layers to stabilize the activations during training.

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

CMOS Complementary Metal–Oxide–Semiconductor.

CNN Convolutional Neural Network, a Deep Learning algorithm that can extract useful information from a multi-dimensional input matrix, typically an image.

CPU Central Processing Unit. General-purpose processor that can handle various tasks.

DRAM Dynamic Random Access Memory. A type of volatile memory that is typically used as the "main memory" outside the processor. Cheap but slower than SRAM.

DVFS Dynamic Voltage and Frequency Scaling. A technique to reduce power consumption where the voltage and frequency of a processor is dynamically adjusted based on its workload.

EDP Energy Delay Product. A metric to measure how energy-efficient and low-latency of a system is.

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

GALS Globally Asynchronous Locally Synchronous. A circuit design method that breaks the global clock network into modular and decentralized clock domains.

GPU Graphic Processing Unit. A kind of processor that is optimized for processing image related tasks.

im2col *image to column*, or *image to column vector*. An algorithm to efficiently implement convolution.

KiloCore 2 The 4th generation of *Asynchronous Array of simple Processors* design inspired by the AsAP 2.0 platform.

Leaky ReLU Leaky Rectified Linear Unit activation function. Allows a positive input to pass without modification, and applies a small, positive gradient when the input is not positive.

ML Machine Learning. Using supervised or unsupervised algorithms to perform complex analysis on large data sets. Example applications: Image Classification, Natural Language Processing, Autonomous Driving, etc.

PD-SOI Partially Depleted Silicon-on-Insulator.

ReLU Rectified Linear Unit activation function. Allows a positive input to pass without modification, and removes every non-positive input.

RISC Reduced Instruction Set Computer. A computer (processor) that uses a short and optimized Instruction Set.

SIMO Single Input Multiple output.

SQNR Signal to Quantization Noise Ratio. A measurement showing the quality and accuracy of the quantization.

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.

TDP Thermal Design Power. Refers to the power consumption under the maximum theoretical load.

YOLO You Only Look Once. An Object Detection System that can output predictions in one Neural Network evaluation. The network is trained to predict categoriness, location, and bounding boxes all at the same time, the data is encoded inside the output 3D tensor, so no Softmax layer or Fully-connected layer is used.

Bibliography

[1] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016.

[2] Brent Bohnenstiehl. *Design and Programming of the KiloCore Processor Arrays*. PhD thesis, University of California, Davis, Davis, CA, USA, March 2020. http://vcl.ece.ucdavis.edu/pubs/theses/2020-1.bbohnenstiehl/.

[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] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

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

[6] Nvidia jetson nano gpu specs — techpowerup gpu database. https://www.techpowerup.com/gpu-specs/jetson-nano-gpu.c3643. Accessed: 8/6/21.

[7] Nvidia jetson tx2 gpu specs — techpowerup gpu database. https://www.techpowerup.com/gpu-specs/jetson-tx2-gpu.c3231. Accessed: 8/6/21.

[8] Byung-Gil Han, Joon-Goo Lee, Kil-Taek Lim, and Doo-Hyun Choi. Design of a scalable and fast yolo for edge-computing devices. *Sensors*, 20(23), 2020.

[9] Intel core i3-8145u specs — techpowerup cpu database. https://www.techpowerup.com/cpu-specs/core-i3-8145u.c2101. Accessed: 8/6/21.

[10] Intel core i9-9900k specs — techpowerup cpu database. https://www.techpowerup.com/cpu-specs/core-i9-9900k.c2098. Accessed: 8/6/21.

[11] Jetson nano: Deep learning inference benchmarks. https://developer.nvidia.com/embedded/jetson-nano-dl-inference-benchmarks. Accessed: 8/6/21.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, May 2017.

[13] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision, 2015.

[14] Hugo Touvron, Andrea Vedaldi, Matthijs Douze, and Hervé Jégou. Fixing the train-test resolution discrepancy, 2020.

[15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge, 2015.

[16] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit, 2017.

[17] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[18] Zhewen Yu and Christos-Savvas Bouganis. A parameterisable fpga-tailored architecture for yolov3-tiny. In Fernando Rincón, Jesús Barba, Hayden K. H. So, Pedro Diniz, and Julián Caba, editors, *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, pages 330–344, Cham, 2020. Springer International Publishing.

[19] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement, 2018.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] D. N. Truong, W. H. Cheng, T. Mohsenin, Z. Yu, A. T. Jacobson, G. Landge, M. J. Meeuwsen, A. T. Tran, Z. Xiao, E. W. Work, J. W. Webb, P. Mejia, and B. M. Baas. A 167-processor computational platform in 65 nm CMOS. *IEEE Journal of Solid-State Circuits (JSSC)*, 44(4):1130–1144, April 2009.

[28] 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.

[29] Tinoosh Mohsenin and Bevan M. Baas. Split-row: A reduced complexity, high throughput LDPC decoder architecture. In *IEEE International Conference of Computer Design (ICCD)*, October 2006.

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

[31] 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, March 2017.

[32] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.

[33] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High Performance Convolutional Neural Networks for Document Processing. In Guy Lorette, editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France), October 2006. Université de Rennes 1, Suvisoft. http://www.suvisoft.com.

[34] Yangqing Jia. Learning semantic image representations at a large scale, 2014.

[35] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks, 2013.

[36] 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. http://www.ece.ucdavis.edu/cerl/techreports/2011-4/.

[37] Norbert Mitschke, Michael Heizmann, Klaus-Henning Noffz, and Ralf Wittmann. A fixed-point quantization technique for convolutional neural networks based on weight scaling. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 3836–3840, 2019.

[38] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[39] Light version of convolutional neural network yolo v3 & v2 for objects detection with a minimum of dependencies (int8-inference, bit1-xnor-inference). https://github.com/AlexeyAB/yolo2_light. Accessed: 8/6/21.

[40] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference, 2017.