JAIST Repository
https://dspace.jaist.ac.jp/

Title	FPGA向けのオフチップメモリ参照を削減したブロックベースCNNアクセラレータに関する研究
Author(s)	陳, 炎
Citation	
Issue Date	2025-03
Type	Thesis or Dissertation
Text version	ETD
URL	http://hdl.handle.net/10119/19924
Rights	
Description	Supervisor: 田中 清史, 先端科学技術研究科, 博士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Doctoral Dissertation

A Study on Block-based CNN Accelerators with Reduced Off-chip Memory References for FPGAs

CHEN Yan

Supervisor: TANAKA Kiyofumi

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
[Information Science]

March 2025

Abstract

The rapid advancement of convolutional neural networks (CNNs) has revolutionized various fields since AlexNet's success in 2012. While GPUs excel at training CNNs, hardware accelerators are vital for inference, leveraging 8-bit or lower quantized data for energy efficiency. However, challenges remain with memory access and computational efficiency. This research introduces a novel CNN accelerator to address these issues.

CNN accelerators generally fall into Overlay and Dataflow categories. Overlay designs sequentially process layers with a unified Processing Element (PE) Array, offering flexibility but heavily relying on off-chip memory, which limits efficiency for lightweight models like MobileNetV2. Dataflow designs dedicate hardware to each layer, minimizing memory access but requiring extensive on-chip memory to store weights, reducing flexibility.

The proposed block-based architecture leverages the strengths of existing designs while addressing their limitations, enabling tailored optimization for different network structures. It incorporates multiple PE Arrays and supports flexible runtime interconnection modes: serial execution, which accelerates entire blocks, and parallel execution, which processes individual layers.

Experiments on 7Z010, ZU3, ZU7, and VU13P FPGAs show significant performance gains, achieving up to 11,821 FPS on VU13P with 8-bit quantized MobileNetV2. Despite having a minimal area requirement comparable to typical Overlay designs, it achieves significantly higher throughput per unit area than state-of-the-art accelerators. Compared to typical Overlay designs, our design reduces overall off-chip memory transfer volume by 93%; compared to typical Dataflow designs, our design reduces the on-chip weight storage requirement by 88%, offering a scalable, high-performance solution for modern CNNs.

The minor research optimizes the accelerators by balancing PE Array size, memory, and DSP allocation. Simulated Annealing (SA) and Genetic Algorithm (GA) are explored, with SA providing more stable results. Experiments on various FPGAs demonstrate significant throughput and area efficiency improvements over manual configurations, highlighting SA's practical utility.

Keywords: FPGA, Hardware Accelerators, Convolutional Neural Networks, MobileNetV2, YOLOv3.

Acknowledgment

Firstly, I would like to express my sincere gratitude to my supervisor, Professor Tanaka Kiyofumi, for his invaluable guidance and support throughout my research. His insights and encouragement have been instrumental in the completion of this dissertation.

Secondly, I would like to extend my appreciation to my second supervisor, Professor Inoguchi Yasushi, for providing numerous valuable suggestions and feedback that greatly contributed to improving my research.

I am also deeply grateful to Professor Kaneko Mineo, who, despite having retired, guided me in completing my minor research. His mentorship has been a significant part of my academic journey.

Furthermore, I would like to thank my graduate school, JAIST, for providing computational resources necessary for running training programs and synthesis tools. I am also grateful for the Doctoral Research Fellowship (DRF) from JAIST, which supported my studies.

Lastly, I would like to express my heartfelt gratitude to my family, who have supported me both physically and mentally throughout this journey.

Contents

Abstract **I**

Acknowledgment **II**

Contents **III**

List of Figures **VII**

List of Tables **X**

Chapter 1 Introduction **1**
1.1 Background 1
1.1.1 Overlay Architecture Accelerators 2
1.1.2 Dataflow Architecture Accelerators 3
1.1.3 Background Works 4
1.2 Objective 6

Chapter 2 Preliminaries **12**
2.1 Blocks 12
2.2 MobileNetV2 14
2.3 Quantization 15
2.4 FINN 16
2.5 DSP Packing 18
2.6 FPGA Basic Knowledge 18
2.6.1 LUT 19
2.6.2 Carry Logic 20
2.6.3 Flip-Flop (FF) 20
2.6.4 Digital Signal Processor (DSP) 21
2.6.5 BRAM 21
2.6.6 Phase-Locked Loop (PLL) and Buffer 22

2.6.7 IO pad and Serial Transceiver 23

Chapter 3 Proposed Architecture 24
3.1 Overview 24
3.1.1 Memory Access Analysis 24
3.1.2 Throughput Upper Limit Model 24
3.1.3 Serial Execution 27
3.1.4 Bus Between Modules 30
3.1.5 Inter-block Data 33
3.1.6 Batch-based Execution 35
3.1.7 Weight Memory Structure 36
3.1.8 Parallel Execution Mode 36
3.1.9 Pixels Processed in Parallel 37
3.1.10 Reduced Control Set 37
3.1.11 Theoretical Upper Limit 38
3.2 ADATA 39
3.2.1 Sub-modules 39
3.2.2 URAM-used and URAM-free Configuration 39
3.2.3 SDRAM version 40
3.2.4 URAM version 40
3.2.5 Running Time 41
3.3 PE Array 0/1 41
3.3.1 DSP Packing 42
3.3.2 DSP Cascading 43
3.3.3 Multi-cycle Path 45
3.3.4 Processing of Input Data 45
3.3.5 Input Shift of Cascaded DSPs 46
3.3.6 Weight Memory Binding 47
3.3.7 Accumulator 48
3.3.8 Quantization 49
3.3.9 Running Time 50
3.4 Combine and Broadcast 50
3.5 PEDW Array 51
3.6 SWU 53
3.6.1 Standard Convolution 55
3.6.2 Depthwise Convolution 56
3.6.3 Running Time 58

3.7 GAP . . . 60
3.8 ADD . . . 60
3.9 Weight . . . 60
3.10 Controller . . . 62
3.11 Improvement on Scalability . . . 63
3.11.1 Increasing Batch Size Result in Negative Improvement 65
3.11.2 Increasing PE Parallelism Result in Negative Improvement . . . 67
3.11.3 Implementation of `concat` . . . 69
3.11.4 Utilize Wider AXI Port to Improve SDRAM Bandwidth Utilization . . . 71

Chapter 4 Experiments **74**
4.1 Quantized Models . . . 74
4.2 Performance Estimation . . . 75
4.2.1 Configurations . . . 76
4.2.2 Throughput Calculation . . . 77
4.2.3 Simulation . . . 79
4.2.4 Simulation of the Design with Improvements . . . 79
4.3 Implementation . . . 87
4.3.1 Implementation on Edge Computing Device . . . 89
4.3.2 Implementation on High-end Device . . . 94
4.3.3 Fully Utilizing VU13P . . . 100
4.3.4 Implementation for Improved Performance . . . 104
4.3.5 Utilizing Improvements for Previous Implementations . 108
4.3.6 Implementation on Ultra-Small Devices . . . 116

Chapter 5 Analysis **120**
5.1 Bandwidth Savings . . . 120
5.2 Achieving the Throughput Limit . . . 124
5.3 Effective DSP Utilization Ratio . . . 133

Chapter 6 Comparison **136**
6.1 Running Other Networks . . . 140
6.2 Object Detection . . . 143

Chapter 7 Optimization **148**
7.1 Utilizing URAM to Store Weights . . . 148

7.2 Optimization Problem Description . . . 148
7.3 Optimization Goal . . . 151
7.3.1 Area . . . 151
7.3.2 PE Array . . . 152
7.3.3 PEDW Array . . . 155
7.3.4 Weight and Quantization Parameter Storage . . . 157
7.3.5 Miscellaneous Modules . . . 162
7.4 Cost Function . . . 162
7.5 Optimization Algorithm . . . 164
7.5.1 Simulated Annealing . . . 164
7.5.2 Genetic Algorithm . . . 166
7.6 Experiment . . . 168
7.7 Conclusion . . . 194

Chapter 8 Conclusion **196**
8.1 Conclusion . . . 196
8.2 Future Works . . . 199

Bibliography **203**

Publications **211**

List of Figures

1.1 Two CNN accelerator architectures. 2
2.1 The typical structure of CNN models. 12
2.2 Blocks used in CNNs. The trapezoid's base length represents the variation in the number of I/O channels, and the color indicates the computational loads. 13
2.3 Processing a convolution layer in different orders. C is channel, W is width (horizontal direction) and H is height (vertical direction) 16
3.1 Throughput upper limits. 25
3.2 An example configuration of our CNN accelerator architecture that runs Inverted Residual Blocks efficiently, and the most used data path is highlighted in red. 26
3.3 The architecture of [1] 29
3.4 Two main interconnection methods target different workloads. 36
3.5 The data path of the parallel execution mode is highlighted, where the common path is in red, the path for PE Array 0 is in green and the path for PE Array 1 is in blue. 37
3.6 Throughput upper limits of our design. 38
3.7 Performing simultaneous multiplication of two unsigned or signed numbers using DSP48E2. 42
3.8 Clock frequencies used for each component. 45
3.9 Example of adder shifting for weight memories. 47
3.10 DSP48 for accumulator. 49
3.11 Writing data to the $K + 1$ SWU memories, whereas data in the red frame are inputted and written at the same time. . . . 54
3.12 Reading data through shift registers, whereas data in red frame from columns 0-2 are for pixel 0 and data with light blue background from columns 1-3 are for pixel 1. 57

3.13 Bus bit-width changing in PEDW Array related path. 58
3.14 The running time of SWU under different conditions 59
3.15 The optimal working state when executing multiple blocks, where the red area represents the stalling time. 63
3.16 Burst transfer to off-chip SDRAM. 67
3.17 The implementation of `concat`. 70
3.18 Multiple accelerator instances sharing the top-level port. . . . 72

4.1 Difference between quantization schemes. 74
4.2 The trend in throughput variation as batch size increases. . . . 82
4.3 The trend in throughput variation as parallelism of PE increases. 85
4.4 The trend of configuration with both improvement on batch size and PE parallelism. 88
4.5 Port allocation on ZYNQ UltraScale+ platform. 90
4.6 The floorplan for single instance occupying an SLR. 97
4.7 The floorplan for multiple instances. 101
4.8 The floorplan for multiple instances. 107
4.9 Port allocation on ZYNQ UltraScale+ platform for improved design. 108
4.10 The Fusion of two AXI slave ports on ZYNQ UltraScale+ platform. 110
4.11 The floorplan for single instances crossing two SLRs. 115

5.1 The amount of data transfer required for inferring a single image. 122
5.2 The amount of data transfer required for inferring a single image with imporved design. 123
5.3 Average off-chip memory bandwidth requirements in GiB/s for each block/layer when running MobileNetV2 with ZU7 SDRAM4. 125
5.4 Average off-chip memory bandwidth requirements in GiB/s for each block/layer when running MobileNetV2 with SDRAM4x2BG on VU13P. 130
5.5 Average off-chip memory bandwidth requirements in GiB/s for each block/layer when running MobileNetV2 with SDRAM2x4BG on VU13P. 131
5.6 Average off-chip memory bandwidth requirements in GiB/s for each block/layer when running MobileNetV2 with SDRAM4x4BG on VU13P. 132

6.1 The configuration for ResNet-50. 141

7.1 Memories with MUX structure. 157
7.2 Simulated Annealing. 165
7.3 Genetic Algorithm. 168
7.4 Cost function distribution 177
7.5 Efficiency distribution 185
7.6 Throughput distribution 192

List of Tables

3.1 Available Paths on Example Configuration 31
3.2 Block/Layer Path Mapping 32
3.3 Depth requirements per block/layer for inter-block memory of ADATA 34
3.4 FPS Matrix of SDRAM Version when Running MobileNetV2 . 64
3.5 FPS per DSP Matrix of SDRAM Version when Running MobileNetV2 64
3.6 FPS Matrix of URAM Version when Running MobileNetV2 . 65
3.7 FPS per DSP Matrix of URAM Version when Running MobileNetV2 66
3.8 FPS Matrix of SDRAM Version when Running EfficientNetV1 68

4.1 FPGA Devices 75
4.2 Estimated, Simulated, and Actual Performance 78
4.3 FPS Matrix of SDRAM Version when Running MobileNetV2 with Improvement 3.11.1 80
4.4 FPS per DSP Matrix of SDRAM Version when Running MobileNetV2 with Improvement 3.11.1 80
4.5 FPS Matrix of URAM Version when Running MobileNetV2 with Improvement 3.11.1 81
4.6 FPS per DSP Matrix of URAM Version when Running MobileNetV2 with Improvement 3.11.1 81
4.7 FPS Matrix of SDRAM Version when Running MobileNetV2 with Improvement 3.11.2 83
4.8 FPS per DSP Matrix of SDRAM Version when Running MobileNetV2 with Improvement 3.11.2 83
4.9 FPS Matrix of URAM Version when Running MobileNetV2 with Improvement 3.11.2 84
4.10 FPS per DSP Matrix of URAM Version when Running MobileNetV2 with Improvement 3.11.2 84

4.11 FPS Matrix of SDRAM Version when Running MobileNetV2 with Both Improvements 85
4.12 FPS per DSP Matrix of SDRAM Version when Running MobileNetV2 with Both Improvements 86
4.13 FPS Matrix of URAM Version when Running MobileNetV2 with Both Improvements 86
4.14 FPS per DSP Matrix of URAM Version when Running MobileNetV2 with Both Improvements 87
4.15 Implementation Result on ZYNQ UltraScale+ Platform . . . 92
4.16 Implement Result of URAM8BG on VU13P 96
4.17 Implement Result of URAM8 under Different Conditions on VU13P 99
4.18 Implementation Result of 7 URAM3BG Instances on VU13P 103
4.19 Implementation Result of 3 URAM3x2BG Instances and 1 URAM3x1BG Instance on VU13P 106
4.20 Implementation Result with Improvements on ZYNQ UltraSCALE+ Platform 109
4.21 Implementation Result of a URAM2x2BG Instances on ZU7 . 111
4.22 Implement Result of URAM2x4BG on VU13P 112
4.23 Implement Result of URAM4x4BG on VU13P 117
4.24 Implement Result of SDRAM1BG on 7Z010 and 7Z007S . . . 119

5.1 Off-chip SDRAM Access 121
5.2 Ratio of required bandwidth and available bandwidth. 128
5.3 Effective Working Time Ratio of DSP 134
5.4 Effective Working Time Ratio of DSP for Each Block/Layer . 135

6.1 Comparison on Mid-range Devices 137
6.2 Comparison on Cost-optimized Devices 138
6.3 Evaluation of Running Other Networks 142
6.4 FPS Matrix of URAM Version when Running YOLOv3 144
6.5 FPS Matrix of SDRAM Version when Running YOLOv3 . . . 145
6.6 FPS Matrix of URAM Version when Running SSD-MobileNetV2145
6.7 FPS Matrix of SDRAM Version when Running SSD-MobileNetV2146
6.8 FPS Matrix of URAM Version when Running SSDlite-MobileNetV2146
6.9 FPS Matrix of SDRAM Version when Running SSDlite-MobileNetV2 147

7.1 Parameters 149
7.2 $RAMB36_d()$ 152
7.3 Resource Requirements of PE Array 153
7.4 Resource Requirements of PEDW Array 156
7.5 Resource Requirements of Miscellaneous Modules 162
7.6 Utilization limits 163
7.7 Scenarios of Genetic Algorithm 169
7.8 FPGA List 170
7.9 Solutions Obtained by Simulated Annealing 194

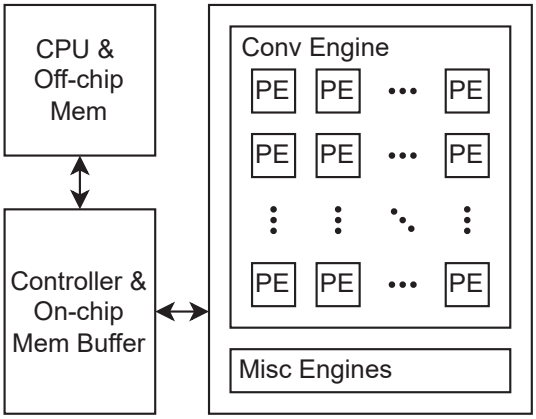
Chapter 1

Introduction

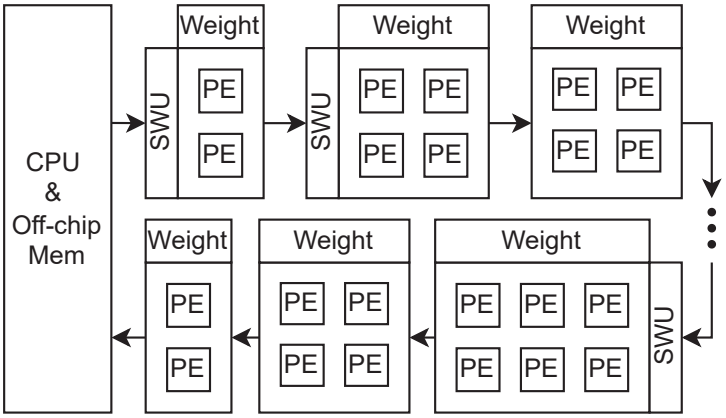
1.1 Background

As Alex et al. [2] won the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [3] with their design, AlexNet, Convolutional Neural Network (CNN) technology rapidly developed and found widespread use across various fields. Modern CNNs contain many layers, and the increase in layers leads to a greater computational load, placing higher demands on computational devices. Training network models typically requires high-precision computing equipment, such as Graphics Processing Units (GPUs). However, quantizing trained network models to 8-bit or lower can still maintain comparable inference accuracy, meaning that using high-precision devices like GPUs for inference results in wasted compute resources and energy. Specialized 8-bit precision accelerators have been developed, and today, such dedicated accelerators are widely integrated into Central Processing Units (CPUs), GPUs, and various embedded systems.

The main component of these accelerators is typically a systolic array or other cascaded structure composed of numerous 8×8 multipliers. Systolic arrays can efficiently perform matrix multiplication and conduct layer-by-layer inference. These accelerators execute different layers' computations on the same multiplier array by overlaying, hence the name "Overlay architecture accelerators [1, 4–8]," and its structure is shown in Figure. 1.1a. In contrast, another approach designs separate dedicated accelerator modules for each layer and connects them in series, with a physical structure that mirrors the network model. This type of accelerator is called a "Dataflow architecture accelerator [9–14]," where the data payload (the activation values of feature maps) flows between modules. Its structure is shown in Figure.1.1b. These are the two primary architectures in accelerators today [15].



(a) Overlay architecture accelerator.



(b) Dataflow architecture accelerator.

Figure 1.1: Two CNN accelerator architectures.

1.1.1 Overlay Architecture Accelerators

As previously mentioned, the theoretical peak performance of Overlay architecture accelerators is very high, and the required minimum area does not vary with the size of the target network model. Overlay architecture accelerators can serve as hard accelerators attached to System on Chips (SoCs), MicroController Units (MCUs), GPUs, etc., and can also be deployed on reprogrammable Field-Programmable Gate Arrays (FPGAs). The layer-by-layer, operator-by-operator execution characteristic allows it to easily handle complex models, particularly those that need to repeatedly read the outputs of previous layers. In non-reprogrammable systems, Overlay architecture accelerators can rely on general-purpose computing devices, such

as CPUs, to handle unsupported operators, showcasing their flexibility. On reprogrammable devices, operators can be flexibly added or removed to adapt to new network models or improve the throughput per unit area for specific target network models.

However, the layer-by-layer execution feature requires temporary storage of inter-layer activation data, which are often large in volume. Designing high-capacity Static Random-Access Memory (SRAM) for on-chip storage of these data incurs significant area. Thus, it is more common to transfer data to high-capacity off-chip Synchronous Dynamic Random-Access Memory (SDRAM) and perform loop nest optimizations (LNO) to improve data reuse. This approach, however, limits performance based on off-chip memory bandwidth. Additionally, a single large computing array may waste computational capacity when executing layers with lower computational requirements.

This design can provide high overall performance for classic networks with substantial computational demands. However, lightweight networks that use depthwise convolutional layers have recently achieved satisfactory accuracy with reduced computational and parameter requirements. Overlay architecture accelerators demonstrate lower efficiency when running these networks, due to memory bandwidth limitations and substantial wastage of computational capability when executing depthwise convolution layers.

1.1.2 Dataflow Architecture Accelerators

In contrast, Dataflow architecture accelerators exhibit consistently high performance across various networks because their submodules and interconnects are fully optimized for the target network. This one-to-one correspondence between the network model and the accelerator makes Dataflow architecture accelerators primarily suited for programmable devices like FPGAs, where they can be reprogrammed to accelerate different network models.

Compared to Overlay architecture accelerators, which frequently access off-chip memory during execution, each layer's module in a Dataflow architecture accelerator has its own dedicated memory module. Weights can be preset or written before startup, so only input image data is needed to produce results directly during execution. The modules are connected via a stream bus, requiring only minor First-In-First-Outs (FIFOs) for throughput matching without any need for temporary data storage, thus completely

avoiding additional memory access and demanding minimal off-chip memory bandwidth.

However, this architecture requires sufficient on-chip memory to fully store the weights of the target network. Additional on-chip memory is also needed for row buffers for the Sliding Window Unit (SWU) performing the `im2col` (also called `unfold`) operation required by layers with convolution kernels size larger than 1, as well as for FIFOs to prevent deadlocks caused by throughput differences between modules. Therefore, the more complex and larger the target network model, the greater the minimum area required to deploy a Dataflow architecture accelerator, which imposes significant constraints on the device's capacity and the number of parameters (the model size) it can support.

Additionally, modern networks widely use operations like shortcut connections and concatenation (`concat`) operators that require repeated reading of previous layer outputs, which are relatively challenging to implement on Dataflow architecture accelerators. For early shortcut connections using 1×1 convolution layers, the throughput of these layers can be adjusted to match the timing difference between payload generation and consumption, and FIFOs can be added for the further timing adjustment. For the currently popular shortcut connections and `concat` operations without any intermediate layers, the usual approach is to add deep FIFOs with depths exceeding the clock cycle count of the time gap between payload generation and consumption to prevent deadlock. This further increases the demand for on-chip memory capacity.

1.1.3 Background Works

Lucian Petrica et al. [15] deployed ResNet50, targeting the ImageNet dataset, on the Alveo U250 data center accelerator card using FINN [10, 11], a framework for designing Dataflow architecture accelerators. The U250 card has on-chip resources identical to the XCVU13P (VU13P) and boasts the highest capacity of Block RAM (BRAM) and Ultra RAM (URAM) among the UltraScale+ series. They quantized weights to 1 bit to reduce the on-chip memory required for weight memory and used a heuristic algorithm to pack weights from multiple layers together. [16] They also introduced asynchronous design elements, allowing a single memory module to be accessed by multiple compute modules simultaneously, increasing the capacity utilization

of individual BRAMs and reducing waste. Despite these optimizations, they still used 1,935 BRAMs to store convolutional layer weights and 109 URAMs for the final fully connected layer weights. [17] For activations, they opted for a higher 2-bit quantization to mitigate accuracy loss from lower-bit quantization, represented as $w1a2$ (1-bit weights, 2-bit activations). They achieved a throughput of 2,703 Frames Per Second (FPS). Even fitting 1-bit quantized weights into the on-chip memory of the UltraScale+ device with the highest capacity is already challenging, let alone for networks using the more common 8-bit quantization scheme.

Di Wu et al. [1] designed an Overlay architecture accelerator with a dedicated execution unit for depthwise convolution, allowing a regular convolution layer to be followed immediately by a depthwise convolution layer. The depthwise convolution module accesses the output buffer of the main computation array, reads its output, performs on-the-fly `im2col` operations and depthwise convolution, and then writes the results back to the buffer. The controller writes data processed by the depthwise convolution module back to the main off-chip memory. They also proposed a technique called Channel Augmentation, which rearranges data to address the issue of insufficient utilization of computing resources in the first layer due to the limited number of input channels (typically RGB 3 channels). Their experimental results showed that the independent depthwise convolution engine significantly reduced off-chip memory access, and the Channel Augmentation technique greatly improved computing efficiency in the first layer, achieving throughput far surpassing other Overlay architecture accelerators without a dedicated depthwise convolution unit. However, their implementation on the XCZU9 showed lower area efficiency in terms of FPS per Digital Signal Processor (DSP) and FPS per Look-Up Table (LUT) compared to the implementation on the XCZU2, suggesting poor scalability and inefficiency in utilizing the larger area of high-end FPGA devices. Our estimates suggest that their implementation on the XCZU9, like many Dataflow architecture accelerators, likely encountered memory bottlenecks.

Manoj Alwani et al. [18] attempted to merge computations from multiple layers by changing the computation order, aiming to reduce the need for storing intermediate data to off-chip memory. If a re-computation mode is used instead of storing and reusing intermediate data, their paper indicated that this approach significantly increases computational load in certain cases. For example, when fusing the first two convolutional layers in AlexNet, the

extra re-computation involved 678 million additional MAC operations, which is 8.6 times the total computation. Compared to re-computation, storing intermediate results has a smaller cost. The paper noted that by storing and reusing intermediate results to avoid re-computation, only about 55.86KB of on-chip storage is needed to save the same amount of off-chip data transfer when fusing AlexNet's first two layers. In experiments with the VGGNet-E network, fusing all 19 convolution and pooling layers required 470 billion additional MAC operations in the re-computation model, while the reuse model only required 1.4MB of storage. However, this study is quite dated, raising doubts about the feasibility of depthwise convolutions, which are commonly used in modern networks.

Weixiong Jiang et al. [14] designed a specialized Dataflow architecture accelerator for the widely-used MobileNetV2. They used deep FIFOs to implement residual connections (i.e., shortcut connections) to avoid deadlock, designed dedicated pointwise and depthwise convolution engines, and proposed a resource allocation strategy to balance the computation time needed for different layers in the Dataflow architecture accelerator. They also introduced a technique called Tunable Activation Weight Imbalance Transfer to improve quantization accuracy, achieving a Top-1 classification accuracy of 72.98%, close to that of the floating-point model. The accelerator was deployed on the evaluation board with a XCZU9EG FPGA, achieving an impressive 1,910 fps throughput. However, the accelerator used 75% of the large FPGA's BRAM, totaling 691 BRAMs, or approximately 24.3Mb of memory. The requirement for such large on-chip memory capacity makes it challenging to deploy on smaller devices, limiting its flexibility.

The studies mentioned above illustrate explorations to improve existing accelerator architectures, such as adding a dedicated depthwise convolution module to conventional Overlay accelerators, using heuristic algorithms to package weights, making it feasible to deploy Dataflow architecture accelerators on current devices, merging multiple layers to reduce off-chip memory access, and mapping entire models onto Dataflow architecture accelerators.

1.2 Objective

Although GPUs used for training models can also be used for inference, recent years have seen the addition of specialized computational units, such as Tensor Cores, to enhance performance for specific workloads by accel-

erating low-precision matrix multiply-accumulate operations. As general-purpose computing units, GPUs rely on CUDA cores to execute most common computations. In the early days of model training on GPUs, calculations were primarily carried out using these CUDA cores. However, with the development of low-precision training in recent years, Tensor Cores have gradually replaced CUDA cores for multiply-accumulate operations in training. Despite these advancements, GPUs, as general-purpose units, still allocate a significant portion of their die area to functions that are not utilized in neural network acceleration. This results in relatively high costs and power consumption, prompting the development of specialized accelerators designed specifically for neural network acceleration to reduce these overheads. Modern GPUs primarily achieve higher performance through extremely high computational power (hundreds to thousands of Tera Operations per second (TOP/s)) and off-chip memory bandwidth TB/s) to run neural network models in an Overlay fashion. Even with low memory access efficiency, they can achieve very high throughput. For instance, the 80GB memory version of the A100 GPU [19, 20] features an off-chip memory bandwidth of 1.94TB/s and contains 432 third-generation Tensor Cores, each capable of performing 1,024 INT8 operations per clock cycle, delivering an INT8 computational power of 624 TOP/s (312 Tera Multiply-Accumulation per second (TMAC/s)). As of now (excluding the unreleased Blackwell architecture), the latest H100 GPU [21] SXM5 version provides an off-chip memory bandwidth of 3,352GB/s. It is equipped with 528 fourth-generation Tensor Cores, each capable of performing 2,048 INT8 operations per clock cycle, achieving an INT8 computational power of 1,978.9 TOP/s (989.45 TMAC/s). Although higher computational power and off-chip memory bandwidth can improve throughput, the energy efficiency issues brought about by more off-chip memory access cannot be ignored.

FPGA is an excellent validation platform. High-capacity FPGAs can closely emulate ASIC performance but still fall short in terms of scale and frequency. In this study, we use the VU13P, which features an impressive 12,288 DSPs, a leading number among FPGA platforms. By employing DSP packing techniques, a single DSP can perform two 8×8 multiplications, allowing for a theoretical maximum of 24,576 8×8 multipliers. At the maximum frequency of 775 MHz for a speed grade of -2, the theoretical peak INT8 performance is 19.0 TMAC/s. The FPGA with the highest DSP count currently is AMD's Versal Premium Series VP1802, which has 14,352 DSPs.

Each DSP can be split into three 8×9 small signed multipliers, resulting in an equivalent of 43,056 8×9 multipliers. At a maximum frequency of 1,070 MHz for a speed grade of -2, its theoretical peak INT8 performance is 46.1 TMAC/s. Among competitors, Altera's Agilex 5 has up to 3,680 18×19 multipliers (A5D 064), which can each be split into three 8×8 multipliers, yielding an equivalent of 11,040 8×8 multipliers. With a maximum frequency of 655 MHz for a -2V speed grade, its theoretical peak INT8 performance is 7.23 TMAC/s. Agilex 7, with up to 17,056 18×19 multipliers (AGF 027), can split each multiplier into two 8×8 multipliers, resulting in an equivalent of 34,112 8×8 multipliers. At a maximum frequency of 771 MHz for a -2V speed grade, its theoretical peak INT8 performance is 26.3 TMAC/s.

Compared to GPUs, which achieve hundreds of TMAC/s, FPGA performance is an order of magnitude lower for two main reasons: 1. Lower operating frequency: GPU frequencies generally range from 1.5 GHz to 2 GHz; 2. Fewer multipliers: For example, the A100 has an equivalent of $432 \times 512 = 221{,}184$ INT8 multipliers, while the H100 has an equivalent of $528 \times 1{,}024 = 540{,}672$ INT8 multipliers. In practical FPGA usage, the number of usable DSPs may be much lower than the theoretical count due to space occupation by non-computational components and routing limitations (e.g., congestion and distance). Additionally, frequency is often constrained and cannot reach the theoretical maximum. Off-chip memory bandwidth is another limitation. While GPUs with multi-channel High Bandwidth Memory (HBM) can achieve several TB/s, FPGA platforms typically offer only tens of GB/s (DDR-based platforms) to several hundred GB/s (HBM-based platforms). In summary, Overlay architecture accelerators mainly rely on computational power and off-chip memory bandwidth, resulting in almost no advantage of implementing Overlay architecture accelerators on FPGAs over GPUs.

An alternative is the Dataflow architecture accelerator, primarily implemented on FPGAs and designed for a single network structure. It requires no memory access during operation, resulting in extremely low off-chip memory bandwidth demands. However, its single-network design makes it unsuitable for ASIC implementation. In early stages, Due to the extremely high utilization of available computing resources and on-chip memory bandwidth by Dataflow architecture accelerators, they even outperformed contemporaneous GPUs, which have no Tensor Cores inside, in throughput. However, the introduction of Tensor Cores significantly boosted GPU computational

power, and advancements in memory technology have enabled GPUs to achieve off-chip memory bandwidths of several TB/s, leading to substantial throughput improvements. State-of-the-art GPUs use the most advanced process technologies, such as the H100 built on TSMC's 5nm process. In contrast, FPGAs, due to their flexibility and long lifecycle, typically always use the latest process technology available at their launch and do not undergo process updates during their lifecycle. For example, the widely used 7 series is based on a 28 nm process and UltraScale+ is based on a 16/14 nm process, while the latest Versal series uses a 7 nm process. Consequently, FPGA on-chip computation energy efficiency lags behind GPUs. In summary, the advantages of Dataflow accelerators implemented on FPGAs have diminished significantly.

The objective is to reduce CNN accelerators' dependency on off-chip memory to lower costs. Furthermore, off-chip data transfer consumes more energy than on-chip computation, meaning communication and off-chip memory access account for a significant portion of power consumption. Reducing bandwidth dependency can indirectly lower power consumption. Currently, GPUs, Tensor Processing Units (TPUs), Neural Processing Units (NPUs) embedded in SoCs, and accelerators implemented on most FPGAs all use Overlay architectures to run network models. These architectures allocate computation tasks layer by layer to available resources, with significant inter-layer data typically stored off-chip. This increases off-chip memory access demands, raises bandwidth requirements, and increases data movement energy consumption.

In this dissertation, we propose a new accelerator architecture that effectively addresses the shortcomings of prior research. Noting that modern advanced network models are often composed of multiple blocks with similar structures, we leverage this by designing a specialized accelerator tailored for a specific type of block, with each layer within the block having its dedicated execution unit. This approach maximizes resource utilization by covering at this granularity, minimizing idle hardware resources.

Our design combines the advantages of both Dataflow and Overlay architectures—low memory access demand and low area requirements—while avoiding their downsides, such as large area and high memory access demands. Since each block's weight size is much smaller than that of an entire model, we avoid the massive on-chip memory requirements typical of Dataflow architecture accelerators needed to store the full model weights.

For most commonly used lightweight models, two computation arrays with similar parallelism are needed to execute the expansion and reduction convolutional layers, respectively. For isolated layers within the network model, we designed a structure that temporarily merges the execution units of two layers to dynamically increase parallelism, allowing for flexible adjustment of computation resource allocation based on workload.

As the accelerator targets entire blocks, the computation arrays of each layer within a block are connected using a handshake-based bus, eliminating the need to temporarily store data between layers. On-chip memory is primarily used for weight storage and row buffers required for sampling depthwise convolution inputs, so the required on-chip memory is minimal, making deployment feasible even on resource-constrained devices.

For high-end devices with larger on-chip memory, we designed an optional on-chip cache to store inter-block data, further reducing off-chip memory bandwidth requirements and improving throughput. Ultimately, our accelerators, which are designed for inverted residual blocks, achieved a throughput of 586 FPS on the cost-optimized ZU3 FPGA when running MobileNetV2, 2,350 FPS on the mid-range ZU7 FPGA, and 11,821 FPS on the high-end VU13P FPGA, where the inter-block caching enabled for the designs targeting mid-range and high-end FPGAs.

The computational demand and inter-layer data volume of CNN models are directly proportional to the input image size. Although specialized accelerators can achieve high throughput for basic image classification tasks, throughput decreases as image resolution increases. In real-world image classification, resolutions like 224×224 are common. For object detection, typical resolutions are 300×300 and 416×416. When the resolution increases to 300×300, the computation load of the convolution layers rises to 1.79 times the original, reducing throughput for these parts to 55.75% of the original at a fixed computational limit. At a resolution of 416×416, the computation load increases to 3.45 times, with throughput dropping to 28.99%.

Furthermore, in data centers, higher throughput and better energy efficiency are critical. This drives GPU- and TPU-based accelerators to relentlessly scale up computational power by adopting more advanced process technologies to gain an edge. Our accelerators have extremely high throughput and are also suitable for data center usage scenarios. In addition, since our accelerator architecture is mainly targeted at blocks and can be executed in fallback mode like a normal Overlay, it has certain ASIC implementation

value to achieve higher throughput and energy efficiency.

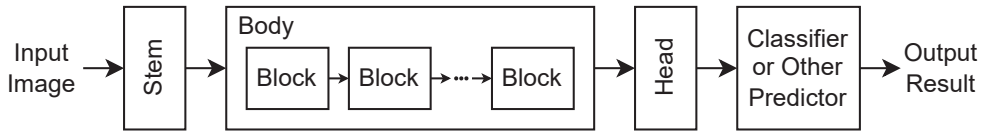


Figure 2.1: The typical structure of CNN models.

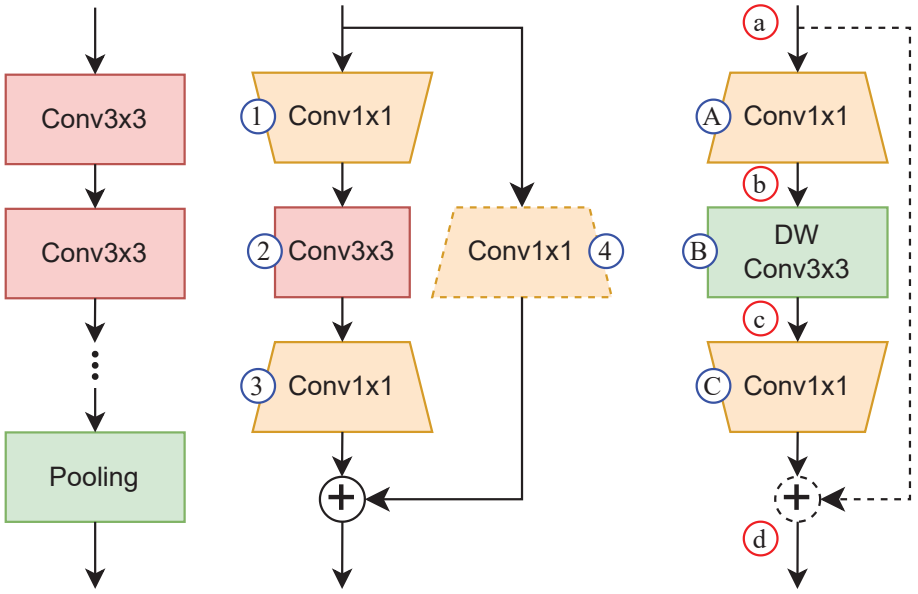
Chapter 2

Preliminaries

2.1 Blocks

As shown in Figure. 2.1,the structure of modern CNNs generally consists of a Stem convolution layer, a Body section made up of a backbone network, followed by a Head convolution layer and a predictor. The backbone network typically consists of many consecutive, similar blocks. The predictor varies depending on the task; for example, in classification tasks, it is often a fully connected layer. Early blocks were simple, such as VGG [22], which consists of several consecutive 3×3 convolution layers followed by pooling (Figure. 2.2a). Later, ResNet [23] introduced the residual block with residual connections, where each block's final output is the sum of the current block's output and its input(Figure. 2.2b). This design effectively addresses the degradation problem that arises when CNNs become very deep. Most blocks consist of three convolution layers: a 1×1 convolution layer for dimensionality reduction (① in Figure. 2.2b), a 3×3 convolution layer for feature extraction ②, and another 1×1 convolution layer ③ for dimensionality expansion. When the input and output dimensions differ, an additional 1×1 convolution layer ④ is added on the shortcut path for dimension adjustment. While residual connections help address neural network degradation, as mentioned earlier, they also pose challenges for accelerator design.

ResNet can achieve high classification accuracy but involves a large



(a) The block in VGG.

(b) The Residual Block.

(c) The Inverted Residual Block.

Figure 2.2: Blocks used in CNNs. The trapezoid's base length represents the variation in the number of I/O channels, and the color indicates the computational loads.

computational workload. Therefore, the inverted residual block with depthwise convolution were proposed in MonbileNetV2 [24]. Inverted residual block and its variations are widely used [25–27]. Unlike the conventional residual block, it expands dimensions internally rather than reducing them and uses less computationally intensive depthwise convolutions for feature extraction(Figure. 2.2c). Networks using inverted residual blocks significantly reduce computation and parameter counts while maintaining comparable classification accuracy to conventional residual blocks, leading to their widespread use today. A typical inverted residual block also consists of three layers: a 1×1 expansion convolution layer(Ⓐ in Figure. 2.2c), a 3×3 depthwise convolution layer Ⓑ for feature extraction, and a 1×1 projection convolution layer Ⓒ for dimensionality reduction. Generally, the expansion convolution layer increases the number of input feature map channels by a factor of six. When the input and output feature maps have the same shape, they are added together via residual connections. While models

using inverted residual blocks reduce weight size and computation, they do not reduce inter-layer activation transfer. Thus, unlike networks using conventional 3×3 convolutions, which are bottlenecked by computation, networks with inverted residual blocks, using 3×3 depthwise convolutions, require high data transfer in less time, making data transfer the primary bottleneck.

2.2 MobileNetV2

As mentioned earlier, MobileNetV2 mainly consists of its proposed inverted residual blocks, while other components are almost identical to conventional CNN models (Figure 2.1). In standard MobileNetV2, the input is a 224×224 3-channel image. It first passes through an input convolution layer with a kernel size of 3×3 and a stride of 2, producing 32 output channels. This is followed by a batch normalization (BN) [28] layer and an activation function, resulting in a feature map size of 32×112×112 (in "NCHW" format with "N" omitted, where the first number represents channels, and the other two represent height and width).

The output is then fed into the first block "blk0". This block is slightly different from others, as it lacks the upsampling convolution layer Ⓐ. Instead, it directly processes the input through a depthwise convolution layer Ⓑ and a downsampling convolution layer Ⓒ, ultimately outputting a 16×112×112 feature map.

The output then progresses through "blk1" and "blk2". These two blocks have identical parameters and form a stage. The depthwise convolution in "blk1", being the first block of the stage, has a stride of 2, while other blocks ("blk2") in the stage have a stride of 1. All blocks in this stage output feature maps of 24×56×56. In "blk1", the channel count is increased to 6 times its input channels (16), resulting in 96 channels, a standard practice in MobileNetV2. Similarly, in "blk2", the channels increase from 24 to 144. Since the input and output feature map sizes of "blk2" are identical, it has a residual connection.

The subsequent blocks ("blk3" to "blk5") form another stage, identical to the previous stage, with output feature maps of size 32×28×28. Blocks "blk6" to "blk9" form a stage that outputs feature maps of size 64×14×14. Blocks "blk10" to "blk12" form a stage that outputs feature maps of size 96×14×14, without further reducing feature map dimensions. Blocks "blk13"

to "blk15" form a stage that outputs feature maps of size 160×7×7.

Finally, "blk16" forms a standalone stage, producing feature maps of size 320×7×7. This is followed by an upsampling convolution layer that increases the feature map size to 1280×7×7, and global max pooling compresses it to 1280×1×1. This can be viewed as the output of a fully connected layer with 1,280 nodes. Compared to earlier models that extensively used flattening and fully connected layers for feature extraction, this approach significantly reduces computation.

The final step is a fully connected classification predictor, which identifies 1,000 ImageNet classes from these 1,280 features.

2.3 Quantization

When deploying CNNs, quantization is commonly applied to improve performance on CPU or NPU platforms. There are various quantization schemes. Most schemes can be represented as

$$\begin{cases} y = \lfloor \text{clamp}(\Gamma \cdot x + B, Q_n, Q_p) \rceil & \text{For Activations} \\ w_Q = \lfloor \text{clamp}(\gamma_{QW} \cdot w, Q_n, Q_p) \rceil & \text{For Weights,} \end{cases} \tag{2.1}$$

where Q_n and Q_p represent the minimum and maximum values for the `clamp` function, $\lfloor \cdot \rceil$ denotes rounding, and Γ and B are the scaling factor and bias with BN folding. They are calculated as:

$$\Gamma = \frac{\gamma_{QW} \cdot \gamma_{QI}}{\gamma_{QO}} \cdot \frac{\gamma_{BN}}{\sqrt{\sigma_{BN}^2 + \epsilon_{BN}}} \tag{2.2}$$

$$B = \frac{1}{\gamma_{QO}} \cdot \left(\beta_{BN} - \frac{\mu_{BN} \cdot \gamma_{BN}}{\sqrt{\sigma_{BN}^2 + \epsilon_{BN}}} \right), \tag{2.3}$$

where γ_{QW} is the weight scaling factor, γ_{QI} is the input feature map scaling factor, and γ_{QO} is the output feature map scaling factor. BN parameters are taken from the BN layer.

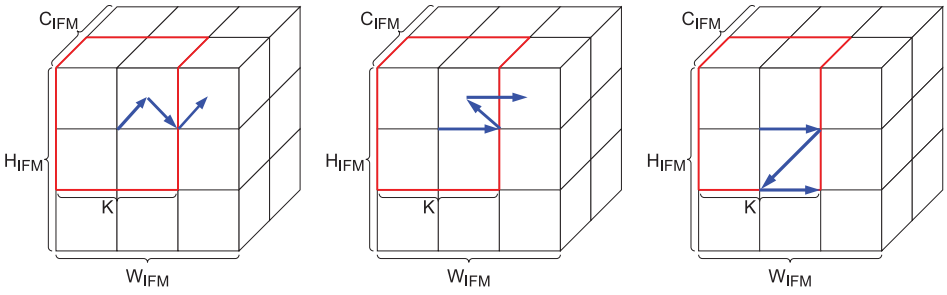
Quantization schemes are divided into two types: post-training quantization (PTQ) and quantization-aware training (QAT). In PTQ, quantization parameters are computed after training based on the distribution of weights and activations. In QAT, fake quantization modules are inserted into the

CNN during training without changing the data distribution. The fake quantization module is computed as

$$y = \gamma_{QO} \cdot \left\lfloor \text{clamp}(\frac{x}{\gamma_{QO}}, Q_n, Q_p) \right\rceil . \tag{2.4}$$

After training, the fake quantization modules are removed, and BN layers are folded by freezing. QAT generally achieves higher accuracy than PTQ. In quantized models for CPU or GPU, feature map data between blocks is not quantized; instead, it is quantized at the entry of the next block to maintain accuracy. In contrast, all data must be quantized, as deploying floating-point numbers on FPGA devices is challenging.

2.4 FINN



(a) Channel ⇒ Row ⇒ Column (b) Row ⇒ Channel ⇒ Column (c) Row ⇒ Column ⇒ Channel

Figure 2.3: Processing a convolution layer in different orders. C is channel, W is width (horizontal direction) and H is height (vertical direction)

FINN [10, 11] is a framework developed by Xilinx for deploying CNNs on FPGAs. It uses a Matrix-Vector-Threshold Unit (MVTU) module to map layers, and these MVTUs are connected sequentially, forming a Dataflow accelerator as shown in Figure. 1.1b.

There are multiple ways to execute layers in the accelerator. The method shown in Figure. 2.3a ensures that the output feature map data of one layer is in the same order as the input feature map data for the next layer, allowing FIFO connections between the modules of two layers. This property

makes it suitable for Dataflow architecture accelerators. To make another method shown in Figure. 2.3b suitable for Dataflow architecture accelerators, additional buffers are required between layers to rearrange data order. For Overlay architecture accelerators, this method strikes a balance between feature map and weight reuse. The method in Figure. 2.3c is less suitable for Dataflow accelerators, but it allows complete weight reuse when memory capacity for weights is limited. FINN uses the method shown in Figure. 2.3a.

In the MVTU, multiple Processing Elements (PEs) process multiple output channels (neurons) in parallel. Each PE also processes multiple input channels (synapses) in parallel. In a single PE, FINN leverages multiple SIMD lanes to process multiple input channels independently, thereby enhancing computational parallelism. The number of these SIMD lanes is represented as S_{PE}. The parallelism of the MVTU is represented by P_{PE} (equal to the number of PEs), which determines the neuron and synapse folding. Neuron folding (Neu) is calculated as:

$$Neu = \left\lceil \frac{C_{OFM}}{P_{PE}} \right\rceil, \tag{2.5}$$

where C_{OFM} represents the number of channels in the output feature map. Synapse folding (Syn) is determined by:

$$Syn = \left\lceil \frac{C_{IFM} \times K \times K}{S_{PE}} \right\rceil, \tag{2.6}$$

where K is the convolutional kernel size and C_{IFM} is the number of channels in the input feature map. S_{PE} represents the number of input channels processed in parallel. The minimum clock cycles (CC) required to execute a layer are given by:

$$CC = H_{OFM} \times W_{OFM} \times Neu \times Syn, \tag{2.7}$$

where H_{OFM} and W_{OFM} represent the height and width of the output feature map, respectively. When running layers with a convolution kernel size $K > 1$, additional SWUs are required to act the `im2col` operation. The SWU contains $K+1$ memories, K for sampling outputs and one extra for inputting the next row simultaneously.

2.5 DSP Packing

The input ports of a single DSP differ based on the FPGA series. For instance, the DSP48E1 [29] used in the Virtex6 and 7 series across all product lines supports the multiplication of two signed numbers of 25-bit and 18-bit widths. The DSP48E2 [30] in the UltraScale/UltraScale+ series supports the multiplication of two signed numbers of 27-bit and 18-bit widths. The latest Versal series DSP58 [31] supports the multiplication of signed numbers of 27-bit and 24-bit widths and additionally supports a dot product mode, which splits into three groups of 9-bit and 8-bit signed multiplications. Since the initial DSPs, Altera has supported the ability to split into multiple 9×9 bit calculation modes (except for the Stratix10 and Arria10 series, which lack this split mode). These DSPs with 9×8 or 9×9 calculation modes can efficiently compute convolutions. However, DSPs without split modes would waste high bit-width capability when performing only 8×8 multiplications, so methods for utilizing a single high bit-width DSP to execute multiple low bit-width multiplications have been proposed.

DSP packing techniques [32–35], enable a single DSP48 to perform multiple multiplications within a single clock cycle. When performing 8×8 bit multiplications, two activations or two weights are packed into the 27-bit input port. A few guard bits are allocated between the two 8-bit inputs. An approximate product is then obtained from the output port. Packing two activations results in $a1 \cdot w$ and $a2 \cdot w$, effectively processing two pixels in parallel. Packing two weights yields $w1 \cdot a$ and $w2 \cdot a$, effectively processing two output channels in parallel. Additional circuitry can be added to the output port for more accurate values. [33]

2.6 FPGA Basic Knowledge

FPGA is a flexible, programmable hardware containing a large number of programmable logic units. By modifying the attributes of these units, FPGAs can simulate any digital circuit. The main programmable units include LUTs, carry logic, Flip-Flops (FFs, also known as registers), DSPs, block memory, Phase-Locked Loops (PLLs), buffers, I/O pins, transceivers, and other routing resources. Additionally, FPGAs include a small number of hard IP cores to implement specific functions, such as PCIe.

2.6.1 LUT

LUTs are used to implement combinational logic, such as $O = \overline{A} \cdot B + C \cdot \overline{D}$. The 4-input LUT (LUT4) is often used as a baseline for comparing the capacity of different FPGA architectures. This metric is referred to as Logic Cell (LC) or Logic Element (LE).

In a LUT4, its four input pins can connect to four different input signals, producing an output signal based on these inputs. As shown in the earlier example, A,B,C,D are the four inputs, O is the output, and their logical relationship can be implemented by a LUT. Essentially, a LUT is an asynchronous SRAM that typically only allows its internal data to be modified during programming. This SRAM has a port width of 1 bit and a depth of 16, sufficient to store the output (truth table) for all possible combinations of four inputs. The four input pins act as read addresses, retrieving the corresponding output for a given combination.

Modern FPGA LUT designs are much more complex than this baseline. For instance, in Xilinx's 7-Series and UltraScale(+) devices, the LUT6 typically has six inputs, two outputs, equivalent two 1-bit-wide, 32-bit-deep SRAMs. One SRAM output connects to output port O5, and it also connects to output port O6 through a multiplexer (MUX) with the other SRAM. This MUX uses one input pin to select which SRAM output is used. The remaining five inputs are shared as address lines for both SRAMs. Consequently, the LUT can function as one LUT6, two LUT5s (sharing inputs but implementing different logic), or a combined LUT5+LUT6 with partial logic sharing.

Some LUTs can dynamically modify their stored data during runtime, a feature known as distributed memory or LUTRAM. These LUTs, which can be used as RAM, can also function as multi-stage shift registers.

In the latest Versal architecture, LUT6 is composed of four LUT4s, with additional output ports and more complex input-output sharing rules. Another FPGA manufacturer, Altera, uses Adaptive Logic Modules (ALM) in its mid-to-high-end devices and LUT4s in entry-level devices. ALM designs are more complex, incorporating two LUT4s and four LUT3s. While its equivalent capacity is 64 bits, the same as LUT6, it provides more input and output ports, allowing for better utilization of logical resources.

2.6.2 Carry Logic

Carry logic is primarily used to mitigate potential issues such as reduced frequency in adders or comparators caused by overly long carry chains. Adjacent carry logic units typically have dedicated cascading resources to chain multiple units into longer carry logic structures. In Xilinx FPGAs, the carry logic design varies across different families. In current-generation devices, the 7 series uses CARRY4, which, as the name suggests, employs a ripple-carry structure to handle four carry inputs. The UltraScale series features CARRY8, while the Versal series introduces LOOKAHEAD8, which uses a more advanced lookahead carry structure to process eight carry inputs in parallel. This parallelism provides significant latency advantages over traditional ripple-carry structures in wide-width computations.

In Altera's ALMs, dedicated ripple-carry structures are available for adders. Unlike Xilinx FPGAs, where an additional 2-input LUT is needed before the carry chain for constructing adders, Altera's design eliminates this requirement, enabling more efficient use of logic resources for adders.

2.6.3 Flip-Flop (FF)

FFs are units used to temporarily store one bit of data, synchronized by clock or control signals. Various types of FFs exist, such as SR-FF, JK-FF, T-FF, and D-FF. In FPGAs, the most commonly used is the D-FF with set/reset and enable features. Its functionality is to capture the input pin's signal level at a clock edge and store it internally for output. Using FFs and clock signals, sequential logic can be implemented.

More FFs are often used to create deeper pipelines, which can improve the operating frequency of a design. However, the design's frequency is ultimately determined by the slowest (longest delay) path in the circuit. Thus, only a well-designed circuit without long paths can utilize more flip-flops to increase the frequency.

Adding more pipeline stages may also increase latency, as results take longer time to propagate. In designs where inputs depend on outputs (e.g., CPU designs with data hazards), deeper pipelines can introduce more stalls if instruction reordering (via compiler optimization or out-of-order execution) is not applied. In such cases, the frequency may increase, but the performance (IPC) could decrease due to inefficiencies.

2.6.4 Digital Signal Processor (DSP)

DSPs are units designed for multiplication operations. Their predecessors were simple multipliers, but recent designs incorporate additional modules and registers to efficiently handle more complex computational tasks with less reliance on general-purpose logic (e.g., LUTs). These tasks include FIR filtering, systolic arrays, and floating-point calculations.

Modern DSPs typically feature pre-adders and post-adders/accumulators around the multipliers. They also include one or more stages of pipeline registers (D-FFs) at the input, output, and interconnections between multipliers, adders, and accumulators to achieve higher operating frequencies. Cascading resources enable multiple DSPs to form wider multipliers or parallel multiply-accumulate units.

In Xilinx FPGAs, DSP types include DSP48E1, DSP48E2, and DSP58, depending on the series. The DSP48E1 features a 25×18-bit signed multiplier (unsigned multiplication requires fixing the MSB to 0) and a pre-adder for 25-bit signed addition or subtraction. Its complex post-accumulator, also referred to as an Arithmetic Logic Unit (ALU), supports a wide range of functions, including basic arithmetic, bitwise logic operations, XOR-reduction for error checking, and data comparison. The ALU operates at a 48-bit width but can be split into two 24-bit or four 12-bit ALUs for smaller-width operations.

The DSP48E2 extends the pre-adder and one multiplier input to 27 bits and introduces more computation modes. The DSP58, based on the DSP48E2, further extends another multiplier input to 24 bits and allows splitting into three 9×8-bit multipliers.

In Altera FPGAs, DSP capabilities vary across series. Recent FPGA generations include 27×27-bit multipliers that can be divided into two 18×18-bit or 19×18-bit multipliers. Some even support finer splits, such as three 9×9 multipliers (Stratix V), four 9×9 multipliers (Agilex 7), or six 9×9 multipliers (Agilex 5).

2.6.5 BRAM

BRAM refers to on-chip synchronous SRAM with relatively large capacity (tens of kilobits) used to temporarily store information required by circuits. While LUTRAM can also store data, its capacity is limited, and its low bit width necessitates combining a large number of LUTs to form large memory

blocks. This increases timing closure complexity and routing congestion. Therefore, selecting the appropriate memory type based on the required capacity and port bit width is critical.

Xilinx FPGAs feature two types of block memory. One is the 36Kb BRAM, with adjustable port widths ranging from 1 to 72 bits (9 to 72 bits in Versal). When the port width of a single BRAM is ≤ 36 bits, it can form single-port memory (one port for read or write), simple dual-port memory (namely pseudo dual-port, one port for read, the other for write), or true dual-port memory (two independent ports for read/write). For port widths > 36 bits, a single BRAM cannot create true dual-port memory. In simple and true dual-port modes, the two ports can operate on different clocks, enabling the creation of asynchronous FIFOs for data buffering and clock domain crossing (hardware FIFOs were removed in Versal).

The second type is the 288Kb UltraRAM (URAM). Compared to BRAM, URAM in UltraScale series FPGAs restricts port width to 72 bits, making it less flexible. In true dual-port mode, URAM simulates two ports through time-division multiplexing using a single physical port. Thus, simultaneous read and write to the same address through different ports may yield inconsistent results. Additionally, URAM does not support dual-clock operation. The latest Versal URAM enhances port width flexibility, allowing adjustments from 9 to 72 bits.

Altera's BRAM primarily includes M20K with a capacity of 20Kb in newer architectures and M10K (10Kb) in older ones.

2.6.6 Phase-Locked Loop (PLL) and Buffer

PLLs and buffers generate required clock signals and manage signal fan-out, respectively. Typical FPGA boards have fixed-frequency oscillators (e.g., 25MHz, 33MHz, 50MHz, 100MHz), providing input clock signals to the FPGA. For specific applications like DDR memory or QSFP ports, custom oscillators may directly provide specialized frequencies, often dedicated to specific IPs and unavailable for general use.

For custom applications requiring non-standard frequencies, PLLs generate the desired clock signals. A PLL increases the input frequency by a multiplication factor, which can be fractional on advanced platforms. The resulting frequency, termed the VCO frequency, must remain within a specific range (typically 500MHz to 1GHz) to ensure stable operation. The VCO

frequency is then divided to obtain the target frequency, with integer division ratios. PLLs can also adjust clock phase. In addition, PLL can ensure that multiple clocks with different frequencies but proportional relationships are from the same source (i.e. synchronized clocks) to create multi-cycle paths.

Xilinx devices provide two frequency adjustment modules: PLLs and Mixed-Mode Clock Managers (MMCMs). MMCMs operate similarly to PLLs but offer additional features for broader applications.

When signals must connect to numerous target pins (e.g., clock or control signals like enable, reset, and set), high fan-out occurs. Since every output pin has a specific drive strength, weak drive strength can cause longer delays in level transitions. Designing all output pins with high drive strength is challenging due to physical and cost constraints. Therefore, a limited number of high-drive cells, called buffers, are used. Buffers enhance the driving capability of signals, though additional delay from the wires and buffers must be considered. Buffers come in various types, depending on their purpose and scope.

2.6.7 IO pad and Serial Transceiver

IO pads and serial transceivers are components used by FPGAs to communicate with external devices. IO pads are generally divided into two categories: GPIO and high-speed serial transceivers. Some devices feature GPIO equipped with lower-speed serial transceivers (typically below 3Gbps) and SDR/DDR conversion modules. However, compared to high-speed serial transceivers, these GPIO-based transceivers are simpler in functionality and lack features like hardwired DC-balanced encoding schemes (e.g., 8b/10b, 128b/130b).

High-speed serial transceivers cannot typically be used as general-purpose I/O (GPIO) and are instead dedicated to high-speed serial communication protocols such as PCIe, SATA, and QSFP. The maximum data rate of high-speed serial transceivers varies depending on the device series and its intended application, ranging from 3Gbps to 112Gbps. The availability of built-in DC-balanced encoders also differs: most devices support 8b/10b encoding, newer platforms support 128b/130b encoding, and some devices support 64b/66b, 64b/67b, and PAM-4 encoding.

Chapter 3

Proposed Architecture

3.1 Overview

As mentioned earlier, the acceleration bottleneck in lightweight networks with depthwise convolution layers typically lies in inter-layer data transfer. Since MobileNetV2 is widely used and has ample related research for comparison, we use it as a paradigm for bandwidth performance analysis and accelerator design. The relevant content of this section has been presented in [36], along with the specific implementation details provided in the following sections

3.1.1 Memory Access Analysis

When the input and output feature maps have identical dimensions and there is a 6× upsampling within the block, assuming the input data size is one unit, an Overlay architecture accelerator that executes layer-by-layer needs to read one unit of data for the main data path and shortcut data path at different times at ⓐ in Figure. 2.2c, due to a significant time difference in data consumption between the two paths. At ⓑ, six units of data are written and read, and at ⓒ, six units are also written and read. Since summing the values from the two data paths incurs negligible area cost and delay, we assume they can be executed continuously, so it requires writing one unit of data at ⓓ, totaling 27 units of data read and written.

3.1.2 Throughput Upper Limit Model

We constructed an off-chip memory bandwidth throughput upper limit model based on actual network structure and assuming that channel counts in all layers are rounded up to multiples of 8 for efficient transfer. The memory controller and off-chip memory are assumed to support full-speed, cost-free,

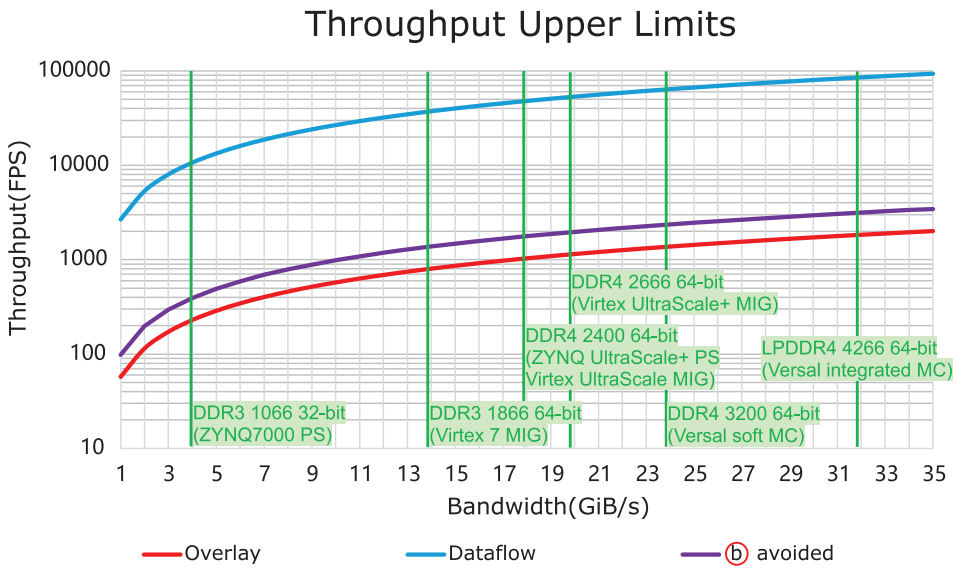


Figure 3.1: Throughput upper limits.

efficient random access. Additionally, we assume the Overlay architecture accelerator has ample on-chip memory and has been optimized through loop unrolling, requiring minimal data reads—each layer's weights only need to be read once, and inter-layer data only need to be written and read once. As mentioned earlier, [1] added an independent depthwise convolution engine to the output buffer at ⓑ for continuous execution to reduce feature map transfer by up to 44.4%, which we also consider in this model. The model's prediction results are shown in Figure. 3.1, which also includes peak off-chip memory bandwidth for common FPGA platforms. Note that actual performance is limited by bandwidth allocation to Processing System (PS)/Programmable Logic (PL) on ZYNQ platforms and the limited random access capability of soft memory controllers MIG on normal FPGA platforms, which often do not reach the peak bandwidth.

We created a System Performance Modeling Project [37] using Advanced eXtensible Interface (AXI) Traffic Generator [38] and AXI Performance Monitor [39] to test the maximum bandwidth allocated to the PL side by default on ZYNQ platforms. On a ZYNQ UltraScale+ platform with 64-bit 2,400MT/s DDR4 SDRAM, the maximum PL bandwidth is 14.05 GiB/s for reads, 12.54 GiB/s for writes, and 11.82 GiB/s for mixed reads/writes. On a

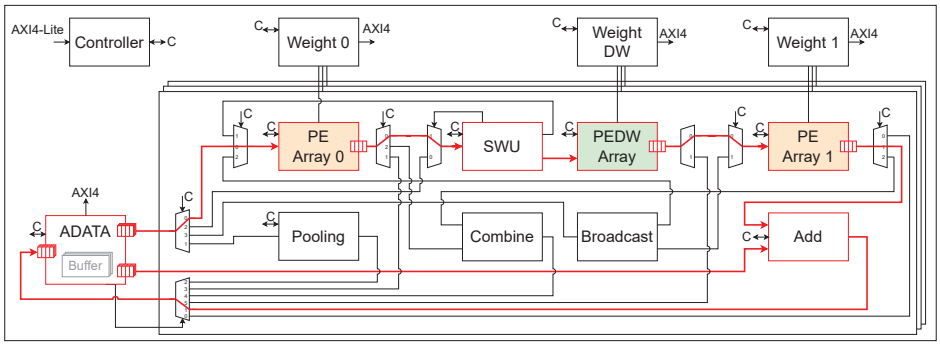


Figure 3.2: An example configuration of our CNN accelerator architecture that runs Inverted Residual Blocks efficiently, and the most used data path is highlighted in red.

ZYNQ7000 platform with 32-bit 1,066MT/s DDR3 SDRAM, the maximum PL bandwidth is 3.71 GiB/s for reads, 3.76 GiB/s for writes, and 3.21 GiB/s for mixed reads/writes.

From Figure. 3.1, we observe that even with advanced platforms boasting ultra-high memory bandwidth, the theoretical maximum throughput of an Overlay architecture accelerator dependent on off-chip memory reads/writes is only around 2,000 FPS. In contrast, a Dataflow architecture accelerator, which does not need to read weights from memory for each image and involves no off-chip reads/writes of feature maps during operation, achieves a theoretical maximum throughput exceeding 100,000 FPS. The calculated values for [1] are also marked in the figure; it uses a ZCU102 platform where the PS-side memory is SO-DIMM with a maximum bandwidth of 2,133 Mb/s according to [40]. Therefore, it is around 11 GiB/s, with an intersection near 1,000 FPS. Their achieved throughput is 809.8 FPS, which is expected given the actual random memory access by Overlay architecture accelerators. The approach in [1] does indeed achieve a noticeable improvement in throughput limit over typical Overlay architecture accelerators by avoiding memory access at (b), yet it remains far from the performance of Dataflow architecture accelerators. Reducing off-chip memory access further is key to improving throughput.

3.1.3 Serial Execution

To further reduce off-chip memory access, ⓐ, ⓒ, and ⓓ can be optimized. For ⓒ, we can avoid access by executing an entire block continuously; for this, we designed an accelerator architecture that executes by block. Figure. 3.2 shows an example accelerator structure capable of efficiently executing inverted residual blocks.

Running a CNN model requires numerous modules to handle various operators. Common operators in CNNs include `matmul`(matrix multiplication), `im2col`, `pool`, `mul` (scale), `add` (bias), `relu`, and others. In accelerators, data are often quantized to reduce computational bit-width and the complexity of floating-point operations. This introduces additional operators like `round` and `sign`.

Accelerators typically merge multiple consecutive identical operators to reduce computational complexity and data transfer. For example, batch normalization and quantization's multiplication and addition (scale and bias), as well as the bias in convolution or fully connected layers, can be combined into a single `mul` and a single `add` operation. Besides, the `sign`, `round` and `relu` operations are integrated into the quantization modules of PE Arrays.

Our accelerator architecture includes various modules for different operators. The main part is the core operator of convolution and fully connected layers, `matmul` operation. The `matmul` operation in convolution layers comprises multiple smaller calculations, while the `matmul` in fully connected layers follows the standard form, which can be viewed as multiple smaller computations. The `madd` and `acc` parts of our PE Array 0/1 can perform `matmul` calculations and support configurable loop counts for step-by-step execution of smaller computations.

The core operation of depthwise convolution layers is also `matmul`, but it differs slightly from regular convolution layers. Its loop structure lacks the input channel loop, resulting in significantly reduced computational workload. Our custom depthwise convolution computation engine, the PEDW Array, efficiently handles this part.

Next is `im2col`, which transforms input feature maps into the format required for convolution operations. For example, if the input feature map is of size 4×7×7 (height and width of 7, and 4 channels in NCHW format) with a 3×3 kernel, same padding, and stride of 1, the result of `im2col` would be

196 groups of sequences with 9 elements each. The 196 groups correspond to sampling from the 4×7×7 input feature map, and 9 refers to the 3×3 data elements sampled each time. The SWU module in our accelerator performs this operation and can adjust the output order and parallel approach of the 196 groups as needed.

The `pool` compresses feature maps with larger sizes into smaller ones. Common pooling methods include max pooling and average pooling, both of which have minimal computational complexity. A special form of pooling, global pooling, compresses an entire feature map into a size of C×1×1 for use in conjunction with fully connected layers. In recent CNN models, conventional pooling has become less common, with convolution layers or depthwise convolution layers with a stride of 2 increasingly used to progressively reduce feature map sizes. However, global pooling, often replacing flattening operations, has become more prevalent. Our accelerator includes a module for global pooling.

Next we will introduce how to design the accelerator

First, we take a convolution engine and split it into two clusters, namely PE Array 0 and 1. These two clusters handle the computation of 1×1 expansion convolution layer Ⓐ and 1×1 projection convolution layer Ⓑ in the inverted residual block, respectively. Since the computational load of these two layers is generally the same in most inverted residual blocks, they are set with identical parallelism. When processing layers with slight discrepancies in computation, such as blocks with stride 2 that halve the output feature map size in width and height, this setup may lead to some computational resource waste.

Next, we design a specialized computation engine, the PEDW (Processing Element for Depth-Wise convolution) Array, specifically for the 3×3 depthwise convolution layer Ⓑ between the two conventional convolution layers. Due to the relatively low computation requirements of this layer, the number of multipliers used is also minimal, meaning that even during independent layer execution, idle resources will not significantly impact computational efficiency.

Since the depthwise convolution kernel is usually not 1×1, an additional engine performs the `im2col` operation to unfold the input feature map on-the-fly. We utilize a Dataflow structure within the block, and the SWU in FINN meets our requirements. Because the Dataflow architecture requires consistent input ordering between each module, our computation modules

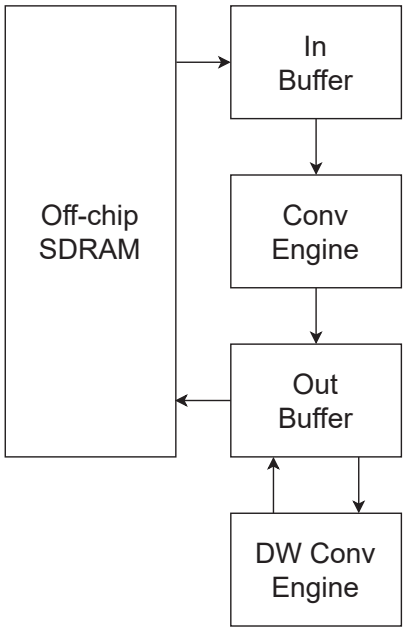


Figure 3.3: The architecture of [1]

and various processing modules are designed to process or transfer data in a channel-row-column sequence to ensure alignment. By connectiong these modules according to the actual structure of an inverted residual block, we create a Dataflow architecture accelerator capable of executing the entire block.

Processing each block sequentially allows the backbone network, composed of inverted residual blocks, to be computed. The Dataflow connection at ⓑ and ⓒ avoids off-chip data exchange, resulting in a total off-chip memory access reduction of 88.9%.

Currently, there is no Overlay research at the intra-block level using Dataflow, where Overlay is applied at the block level. The closest work is [1], as mentioned earlier, which designs two computation engines targeting conventional convolution layers and depthwise convolution layers, respectively. Its structure is shown in Figure. 3.3.

First, feature map data are read from off-chip memory and cached in an on-chip input buffer. The conventional convolution engine reads data from this buffer for computation, with results written to another output buffer. The depthwise convolution engine can access this buffer, read data, and

compute results. Since depthwise convolution does not change the channel count, it only reduces the feature map size when the stride is 2.

Originally, during conventional convolution engine execution, the buffer results were written back to off-chip memory. In this design, if there are consecutive executions of conventional and depthwise convolution layers, only the results of the depthwise convolution are written back to off-chip memory. These buffers are used for on-the-fly `im2col` operations, so the two on-chip buffers consume memory equivalent to that required by two SWUs.

In our design, only one SWU is needed to handle both conventional and depthwise convolutions. Splitting the conventional computation engine into two consecutive computations does not incur additional off-chip memory capacity overhead but has a significant impact on off-chip memory bandwidth savings.

3.1.4 Bus Between Modules

The bus width between modules is generally $B \times BW_A \times DC \times 2 \times PP$, where B is the batch size, BW_A is the bit-width of the quantized activation value, $DC \times 2$ includes DC, which is the height of the DSP cascade, and 2 is a multi-cycle correction factor that will be explained in later sections. PP represents the parallel number of processed pixels (2 in our design), enabling almost all modules in the accelerator to process two pixels in parallel. Modules are interconnected with handshake buses, and most module outputs (before DEMUX, to ensure that routing switches do not cause incorrect transmission targets) have handshake registers to resolve issues with prolonged ready signals.

Data paths can be switched via MUX and DEMUX, and possible paths in Figure. 3.2 are listed in Table.3.1, where paths 2, 3, 4, 6, 7, 8, and 9 are needed for executing MobileNetV2.

Path 8 is intended to split the inverted residual block into two parts when there is a substantial imbalance in computation load between the two conventional convolution layers, as seen in the final block. This enables separate execution by merging the two PE Arrays to process this high-load, high-parameter layer, thereby reducing PE idling and lowering the minimum required on-chip memory for storing weights.

The control signals of the most MUX and DEMUX connect to the controller, which manages the operation of each module and switches paths

Table 3.1: Available Paths on Example Configuration

#	PATH	Block/Layer
1	ADATA ⇒ PE Array 0 ⇒ ADATA	Fully Connected / Conv 1x1
2	ADATA ⇒ Broadcast ⇒ PE Array 0 & PE Array 1 ⇒ Combine ⇒ ADATA	Fully Connected / Conv 1x1 (parallel execution)
3	ADATA ⇒ SWU ⇒ PE Array 0 ⇒ ADATA	Conv k×k
4	ADATA ⇒ SWU ⇒ PEDW Array ⇒ PE Array 1 ⇒ ADATA	(Inverted) Residual Block w/o channel expansion w/o residual connection
5	ADATA ⇒ SWU ⇒ PEDW Array ⇒ PE Array 1 ⇒ ADD ⇒ ADATA	(Inverted) Residual Block w/o channel expansion w/ residual connection
6	ADATA ⇒ PE Array 0 ⇒ SWU ⇒ PEDW Array ⇒ PE Array 1 ⇒ ADATA	(Inverted) Residual Block w/ channel expansion w/o residual connection
7	ADATA ⇒ PE Array 0 ⇒ SWU ⇒ PEDW Array ⇒ PE Array 1 ⇒ ADD ⇒ ADATA	(Inverted) Residual Block w/ channel expansion w/ residual connection
8	ADATA ⇒ PE Array 0 ⇒ SWU ⇒ PEDW Array ⇒ ADATA	half block w/ channel expansion w/o residual connection
9	ADATA ⇒ GAP ⇒ ADATA	Global Average Pooling

when relevant modules are idle. A few DEMUX control signals are directly connected to the certain modules; these modules do not synchronize input-output switching with the currently running block or layer but instead use a state machine on their input side to switch states.

The path mapping of our accelerator running MobileNetV2 is listed in Table 3.2

Table 3.2: Block/Layer Path Mapping

Block/Layer	Path
stem	3
blk0	4
blk1	6
blk2	7
blk3	6
blk4	7
blk5	7
blk6	6
blk7	7
blk8	7
blk9	7
blk10	6
blk11	7
blk12	7
blk13	6
blk14	7
blk15	7
blk16_1st	8
blk16_2nd	2
head	2
gap	9
fc(1/4)	2
fc(2/4)	2
fc(3/4)	2
fc(4/4)	2

3.1.5 Inter-block Data

The ADATA module in Figure. 3.2 handles inter-block data with two options: storing them on-chip or transferring them off-chip. When stored on-chip, it further avoids off-chip data exchanges at ⓐ and ⓓ, so only the input image is needed to obtain the final output without additional feature map transfer, and only weights need to be transferred during operation.

Due to block-based execution, our accelerator requires on-chip memory for weight storage, with the minimum capacity sufficient to store the weights of a single block. If the on-chip memory capacity for weights is increased to accommodate the sum of the largest capacities of two consecutive blocks, block-switching time for initializing the next block's weights can be reduced, resulting in a slight throughput increase. The row buffer capacity needed for SWU and other modules is also minimal.

Modern mid-range and high-end FPGA devices typically have ample on-chip memory, which often leaves significant memory unused. In newer AMD devices, this memory is usually URAM, which offers 8× the capacity of traditional BRAM but lacks features such as variable port width and dual clocks, making it more restricted in use. The inverted residual block expands dimensions within the block and reduces them between blocks, so the inter-block feature map size is significantly smaller than that of intra-block, requiring only minimal capacity to store them.

Thus, we can use these URAMs to form an inter-block feature map buffer, reducing the remaining 11.1% off-chip memory transfer at ⓐ and ⓓ. This URAM buffer can also prefetch the next image to be processed during runtime. We refer to the design using URAM for inter-block buffer as the URAM version, while the design that transfers inter-block feature maps to off-chip SDRAM is referred to as the SDRAM version.

The required on-chip memory capacity is determined by the largest inter-block feature map data volume. For a configuration with $DC = 4$ running standard 8-bit quantized MobileNetV2, the bit-width of the inter-block memory per image is $BW_A \times DC \times 2 \times PP = 128$, and the maximum required depth is calculated as follows: for the input image, it is $\frac{224\times224\times8\times8}{128} = 25088$ (where the 3 channels are padded to 8 channels), and for the stem layer's output feature map, it is $\frac{112\times112\times32\times8}{128} = 25088$ (in the example configuration, the output channel count of "blk0" may need to be padded to 32, matching the stem layer's output feature map size). Therefore, we need an on-chip

Table 3.3: Depth requirements per block/layer for inter-block memory of ADATA

Block/Layer	Memory depth requirements
input	25088
stem	25088
blk0	25088
blk1	6272
blk2	6272
blk3	1568
blk4	1568
blk5	1568
blk6	784
blk7	784
blk8	784
blk9	784
blk10	1176
blk11	1176
blk12	1176
blk13	560
blk14	560
blk15	560
blk16_dw	3360
blk16_prj	1120
head	4480
gap	160

memory with a bit-width of 128 and a depth of 25,088 for inter-block feature maps, which translates to 6.125×2 URAMs. We round this up to $8 \times 2 = 16$ URAMs, providing a depth of 32,768.

In the example configuration, the memory depth requirements for the output feature maps of each block and individual layer are shown in Table. 3.3. It can be observed that the data volume for later blocks is much smaller. As long as the depth needed by the current block and each subsequent block (including input and output, taking the maximum) plus the input image depth is less than 32,768, prefetching can start while processing the block. From the table, it is evident that prefetching can start as early as when processing "blk2".

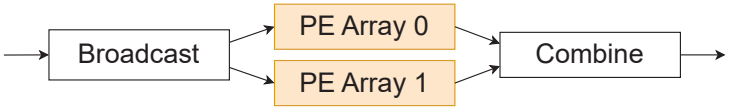
3.1.6 Batch-based Execution

Additionally, from Figure. 3.2, we can observe that the main execution part is duplicated. This is because, as the parallelism of a single PE Array increases, it does not always lead to a linear improvement in throughput. Therefore, we opted to process multiple images in parallel, similar to GPUs, to mitigate these diminishing returns.

Furthermore, we can see that in Figure. 3.2, the top modules related to weight memory, the Controller module, and the ADATA module on the left (which handles inter-block feature map data) are not duplicated. The reason for not duplicating the memory and controller modules is that by fully synchronizing the modules processing multiple images in parallel, they can share the memory modules, the control signals from the controller, and runtime parameters. The state signals from each module are referenced only from submodule 0, avoiding the need for duplicating. Sharing the weight memory when processing multiple images also reduces the average weight memory transfer requirement per image.

The ADATA module is not duplicated due to the fixed number of AXI ports, which limits duplicating. On ZYNQ UltraScale+ platforms, the maximum width of AXI port form/to PS is 128, and parallelizing the data from multiple images onto the AXI port does not yield performance gains. Therefore, we use time-division multiplexing for these AXI ports. Additionally, when ADATA is configured to use URAM for inter-block buffer, these URAM buffer are duplicated.

(a) Serial execution mode.



(b) Parallel execution mode.

Figure 3.4: Two main interconnection methods target different workloads.

3.1.7 Weight Memory Structure

Although the weight memory appears as a unified block in Figure. 3.2, internally, it is organized like a Dataflow architecture accelerator, where each PE in the PE Array has its own dedicated memory to maximize the ultra-high on-chip memory bandwidth of the FPGA. Separate weight memories help reduce routing distance and decrease fan-out signal loads.

3.1.8 Parallel Execution Mode

As shown in Figure. 2.1, aside from the main part composed of blocks, the CNN model network includes other independent layers, especially the head and predictor, which have a large number of output channels and thus significant computational load. It is clear that using just one of the two PE Arrays to execute these independent layers would cause severe computational resource wastage, as the other PE Array would be entirely idle. Additionally, the large disparity in computational load between the two 1×1 convolution layers in the last block would lead to excessive idling of the less-loaded PE Array.

We designed modules that can temporarily combine two PE Arrays into a single large computing array, as shown in Figures. 3.2 and 3.5, labeled as Broadcast and Combine. These modules enable dynamic switching between the serial and parallel execution modes during runtime. The configuration that connects these modules in a block-wise, serial calculation mode is referred to as the serial execution mode (Figure. 3.4a and highlighted paths in Figure. 3.2). The mode that connects two PE Arrays in parallel, using Broadcast and Combine, is called parallel execution mode (Figure. 3.4b and highlighted paths in Figure. 3.5). Parallel execution mode not only reduces

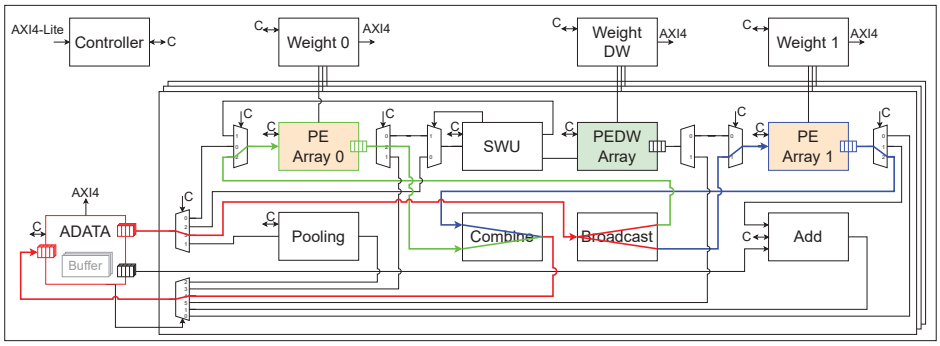


Figure 3.5: The data path of the parallel execution mode is highlighted, where the common path is in red, the path for PE Array 0 is in green and the path for PE Array 1 is in blue.

PE idling when executing a single layer, but also treats the two PE Arrays as one, effectively halving the on-chip weight memory requirement when executing layers with a high number of weights.

3.1.9 Pixels Processed in Parallel

The maximum parallelism achievable through handling multiple input and output channels simultaneously is limited. To further increase parallelism, we utilize the dual-port nature of on-chip BRAM/URAM, allowing the SWU's row buffer to sample activation values from two pixels simultaneously. This capability enables processing of multiple adjacent pixels in a row, enhancing parallelism across additional dimensions.

3.1.10 Reduced Control Set

Many modules have an output FIFO, especially those with longer pipelines, higher register usage, and larger area requirements. The FIFO allows these modules to continue executing already dispatched tasks and temporarily store results when Dataflow blocking occurs, avoiding the need for global clock enable (CE) which could increase routing difficulty and reduce frequency.

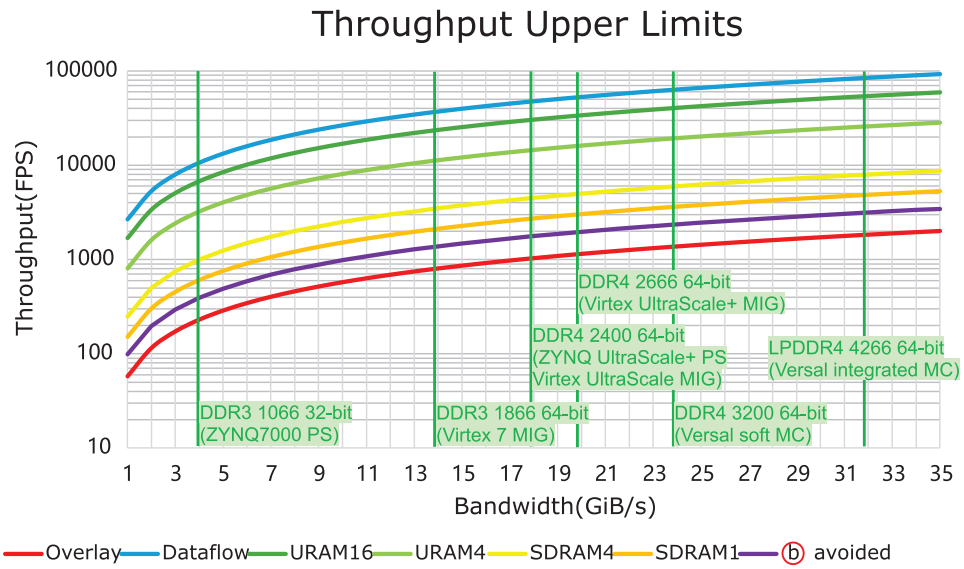


Figure 3.6: Throughput upper limits of our design.

3.1.11 Theoretical Upper Limit

The theoretical upper limit of our accelerator's throughput is illustrated in Figure. 3.6. Here, SDRAM represents the version without on-chip inter-block memory buffering, followed by numbers indicating batch size. Since the URAM version is primarily intended for medium to large FPGAs, the URAM1 configuration has minimal value, and we opted for the high-parallelism URAM16 configuration instead. As batch size increases, the performance gap between the URAM version and the Dataflow architecture accelerator gradually narrows. This is due to the decreasing impact of additional weight transfer bandwidth as batch size grows. The throughput of the SDRAM version is significantly lower than the URAM version but still substantially surpasses that of typical Overlay architecture accelerators.

Next, we will detail each module in the proposed architecture.

3.2 ADATA

3.2.1 Sub-modules

As mentioned earlier, ADATA is responsible for handling inter-block data, and it contains several sub-modules to manage different parts: the Main output port (MAIN module), Shortcut output port (SC module), Write input port (W module), and an initialization module for writing image data into the internal inter-block buffer (INIT module). A control module coordinates the operations of these sub-modules, allowing it to accept commands for the next block/layer while all four sub-modules are still operational.

When the read modules (MAIN and SC) complete their current tasks, the control module can immediately activate these read modules, as it has already received the instructions for the next block. However, this could potentially lead to a situation where the read modules access old data before the W module completes its writing task, resulting in incorrect computation. To prevent this, we have added a safeguard: a flag signal is triggered when there is a mismatch between the current blocks of the read and write modules. When the flag is active, an additional check is performed on the write count (using the write completion count from the B channel in AXI bus) and the read count; data reading is allowed only if the write count exceeds the read count.

3.2.2 URAM-used and URAM-free Configuration

As previously mentioned, ADATA offers two configurations for handling inter-block data: a URAM-based design for on-chip storage and an SDRAM-based design for off-chip storage. The URAM version is suited for mid-to-high-end devices with ample on-chip memory and sufficient resources to support high-parallelism (batch size) configurations. The SDRAM version is intended for smaller devices where both on-chip memory and computational resources are limited, making high-parallelism configurations impractical. In these devices, time-division multiplexing of AXI ports does not lead to excessive waiting times, so the lack of inter-block data buffering does not impact performance. Since the URAM version requires more resources than the SDRAM version, devices capable of accommodating the URAM design can also support the corresponding SDRAM design.

Similar to ADATA's two configurations, its sub-modules also have two operating modes: URAM mode and SDRAM mode. In URAM mode, inter-block data are read from or written to URAM directly. In SDRAM mode, data are read from or written to external SDRAM via time-division multiplexed AXI ports. The SDRAM version omits URAM-related logic and the INIT module.

3.2.3 SDRAM version

The MAIN and W modules share one AXI master port, MAXI0, with the MAIN module using the read-related channels (AR, R) and the W module using the write-related channels (AW, W, B). The SC module has a dedicated AXI master port, MAXI1, using only the read-related channels. The INIT module also has a dedicated AXI master port, MAXI2, using only the read-related channels. The URAM design does not include the SC module's MAXI1 port, while the SDRAM design does not include the INIT module's MAXI2 port.

When transmitting data from parallel-processed images through time-division multiplexed AXI ports, data for each image are polled by ticks. Since all modules for parallel-processed images operate in perfect synchronization, any delay in one image's module will cause delays in the modules of all other images. While adding FIFOs to each parallel image's I/O ports can mitigate this issue, polling by tick not only reduces the required FIFO depth but also prevents scenarios where data for all parallel images are not simultaneously ready due to memory controller or AXI interconnect delays.

3.2.4 URAM version

When using URAM as inter-block buffers, as mentioned earlier, the memory for each parallel image is independent, allowing concurrent read/write operations to meet the high bandwidth and bit-width demands of ultra-high parallel image processing. Due to the URAM port width limitation, the data of two pixels processed in parallel typically have separate memory banks. These URAMs operate in true dual-port mode, ensuring that the two ports do not simultaneously read/write from/to the same address.

The usage strategy for the two ports of URAM is as follows: Port A is primarily used for reading data to the Main output port, while Port B

is mainly used for reading data to the Shortcut output port. For writing the data from the Write input port, data are written through Port A if it is idle; otherwise, it is written through Port B, which has a higher priority than reading the data to the Shortcut output port. Additionally, when prefetching the next image to be recognized during execution, the INIT module has the lowest priority for writing. It only writes when the port is entirely idle, with a preference for Port A when available.

3.2.5 Running Time

The runtime of the ADATA module depends on the port throughput. When running in SDRAM mode, the AXI port throughput may become a bottleneck, as it requires time-division multiplexing of AXI ports to access data from external SDRAM. To ensure address alignment for read/write operations, we round the actual number of images being transferred (batch size) up to the nearest power of 2, so each read/write command aligns perfectly. This may add overhead when the number of parallel images is not a power of 2.

In this configuration, the runtime for the read submodule is represented by:

$$CC_{AREAD} = H_{IFM} \times \left\lceil \frac{W_{IFM}}{PP} \right\rceil \times \frac{C'_{IFM}}{DC \times 2} \times B', \tag{3.1}$$

where H_{IFM} is the height of the input feature map or image, W_{IFM} is the width, C_{IFM} is the number of input channels, PP is the number of pixels processed in parallel, DC is the DSP cascade height, and B is the batch size. Here, C'_{IFM} represents the padded number of input channels, and B' is the rounded-up batch size. The time required for writing is similar, with IFM replaced by OFM to reflect output feature map data.

In URAM mode, no bottleneck arises since each fully parallelized image has a dedicated internal module, and the read time is given by:

$$CC_{AREAD} = H_{IFM} \times \left\lceil \frac{W_{IFM}}{PP} \right\rceil \times \frac{C'_{IFM}}{DC \times 2}. \tag{3.2}$$

3.3 PE Array 0/1

The PE Arrays are composed of many PEs that can perform multiply-accumulate operations, enabling them to handle standard convolutional

layers. Fully-connected layers, treated as 1×1 convolution layers, can also be processed. As our design theoretically requires minimal bandwidth, we have implemented advanced methods to maximize the computational capabilities of the PEs in PE Arrays 0 and 1, pushing the accelerator closer to its throughput limits.

3.3.1 DSP Packing

We use DSP packing techniques to increase the computational density of each DSP. DSPs are designed for signed fixed-point calculations, meaning their most significant bit (MSB) is used as a sign bit. When unsigned data are computed, the range is limited. Generally, weight values are signed, which poses no issues. However, the sign of activation values depends on the previous layer's activation function, e.g., ReLU, which outputs only positive values. Thus, when activation values are unsigned, the MSB needs to remain 0 and cannot be used.

Our design allows activation values to be either signed or unsigned, preserving the original range of the neural network. We perform parallel calculations for two adjacent output pixels, so two activation values are packed together in each operation, where $a1$ and $a2$ come from filters 1 and 2, respectively. Depending on the layer, they may be signed or unsigned, and to ensure compatibility, the MSB is left unused.

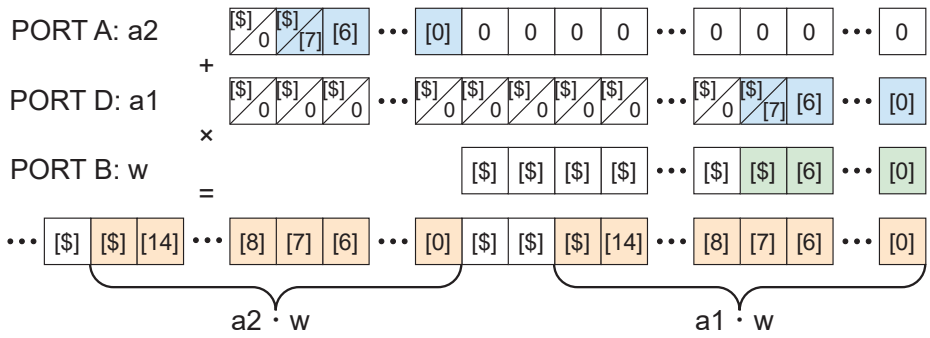


Figure 3.7: Performing simultaneous multiplication of two unsigned or signed numbers using DSP48E2.

As shown in Figure. 3.7, when using DSP48E2, $a1$ occupies bits 0 to 7 of the DSP's A port, with its sign bit always extended to bits 8 to 26. $a2$

occupies bits 18 to 26 of the D port, with its sign bit extended to bit 26.

On the other hand, weight $w1$ occupies bits 0 to 7 of the B port, with its sign extended to bits 8 to 17. The calculation is then $(A+D)\times C$, where bits 0 to 15 of the result correspond to $a1w1$, bits 16 and 17 are sign extensions, and bits 18 to 33 is the approximate result of $a2w1$. According to [33], $a1w1$'s sign bit must be added to restore the exact value.

Since the sign extension of $a1w1$ is only two bits, cascading four DSPs would result in $a1w1$ occupying up to bit 17. Further increasing the number of cascaded DSPs would overflow into the lower bits of $a2w1$. To avoid overflow, the DSP cascade depth in our design is limited to four.

When using older FPGA architectures, as the DSP48E1 pre-adder is only 25 bits, it cannot accommodate guard bits between the two results for cascading. Cascading is therefore disabled, and the built-in ALU cannot be used for addition. Instead, extra LUTs are used to form adders, resulting in lower performance per unit area compared to designs using more advanced DSPs.

3.3.2 DSP Cascading

We leverage DSP modules for multiply-accumulate operations. In FPGAs, DSP columns are typically equipped with dedicated routing resources, allowing cascaded DSPs to use these dedicated routes, reducing the usage of general routing resources. This alleviates routing congestion and increases the final clock frequency, a widely accepted practice for designing efficient FPGA-based computing systems.

In Overlay accelerators, it is common to cascade DSPs to accumulate and pass the partial sum values, then pass weights horizontally through multiple cascaded DSP columns, forming a systolic array interwoven in both vertical and horizontal directions. However, a systolic array requires downstream DSP columns to reuse upstream weights, meaning multiple DSP columns share the same weight input but compute with different activation values. The best way to meet this requirement is to input activations from multiple channels of several filters in parallel, along with weights for a single output channel, and output the activations for one channel of multiple pixels.

The on-chip memory in FPGA platforms is dual-ported. Reading or writing such a large volume of feature maps requires creating multiple copies of the buffer to increase the number of ports, which is undoubtedly a waste

of precious on-chip resources. Moreover, it requires buffering all channels of multiple filters' activations for repeated reads to compute all output channels. Additionally, when using DSP packing to compute $w1a1$ and $w2a1$, where $w1$ and $w2$ are signed, the maximum cascading depth is eight, severely limiting the scale of the systolic array. Therefore, a design based on multiple systolic arrays is not suitable for our target accelerator.

After careful consideration, we adopted a PE structure similar to MVTU in FINN, with loop nesting for processing, incorporating DSP cascading. Compared to MVTU, our PE can handle higher bit-width data, and we have added redundancy in memory modules used for the filter buffer and counters for operational states, allowing parameter flexibility for different layers.

DSPs in the cascade process multiple input channels, effectively computing multiple synapses in parallel. Different channels are input sequentially until all channels have been computed (input channel loop), then the entire convolution kernel is computed by rows and columns (kernel row and column loop). Each DSP column in the cascade acts as one PE, calculating one output channel (one neuron). Each PE uses weights from different output channels (shared among multiple PE Arrays for parallel image processing), enabling parallel computation of multiple output channels, i.e., multiple neurons. The loop inputs the convolution kernel into these PEs, each time using weights for subsequent output channels until all output channels are processed (output channel loop), and then processes the convolution kernel corresponding to the next pixel (feature map row and column loop).

This cascading scheme not only efficiently utilizes the dedicated routing resources within the DSP columns, but also, by staggering the processing times of the same batch of input data, allows the DSP's internal ALU to perform addition on the current products and partial sums from the previous DSP stage. This approach avoids the need for additional summing structures built from LUTs. Unlike a systolic array that requires parallel input of multiple activation values, our approach may require parallel input of more weight data, which can be obtained easily from distributed weight memory modules.

As mentioned before, we leverage the DSP packing to compute both $w1a1$ and $w1a2$, corresponding to two output pixels. In this way, only the activation values of two input filters (typically representing two pixels) need to be buffered to compute all output channels. The input consists of multiple channels of two filters, and the output comprises multiple channels

of two pixels, making the data order and per-clock data volume completely consistent and forming an ideal Dataflow structure.

3.3.3 Multi-cycle Path

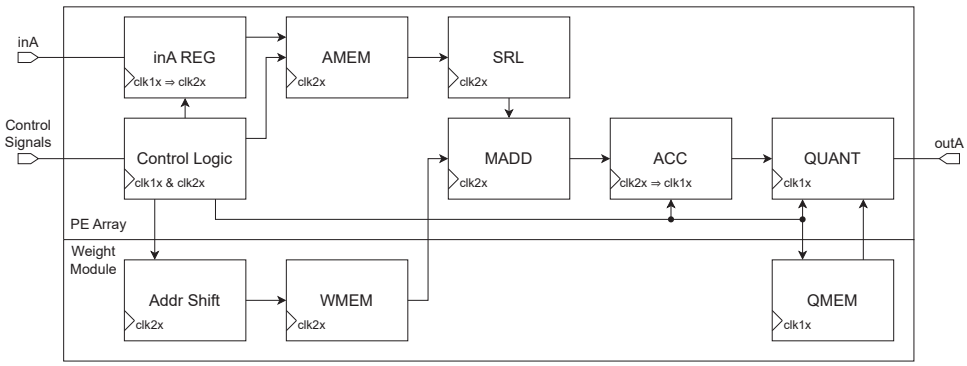


Figure 3.8: Clock frequencies used for each component.

The DSPs' maximum operating frequency is very high, and to enable the control logic to operate at such high frequencies, we need to minimize the logic levels, with a maximum of three levels to match the DSP's performance. Achieving this is challenging, so we run these DSP modules at double the clock frequency, $clk2x$, using multi-cycle paths. Thus, when the cascade depth is DC, the actual number of input channels processed per cycle of the main clock $clk1x$ is $DC \times 2$, leading to the formula $S_{PE} = DC \times 2$. Figure. 3.8 shows the clock frequencies used for each component. The input activation values undergo a clock domain conversion to $clk2x$ immediately. The centralized control state machine counter runs on $clk1x$ and converts control signals to the compute path via multi-cycle paths.

3.3.4 Processing of Input Data

To avoid severe timing violations that would result from fanning out the output of the configuration register indicating whether the activation values are signed to all DSP input ports, we choose to expand the activation values to 9-bit signed integers at an earlier stage. Hence, 8-bit of data and 1-bit of sign.

We expand the sign of the input activation values right after clock domain conversion, before the data are stored in the filter buffers (AMEM in Figure. 3.8). When running MobileNetV2 with $DC = 4$, the on-chip memory required for buffering the activations of the two input filters is minimal. Since these buffers operate at $clk2x$, each filter requires only a 36-bit width to handle activations, where four 1-bit data are allocated as sign-extension bits for four 8-bit input activations. A depth of 320 is sufficient to store the activations of an entire filter. The maximum BRAM width of 36 bits can be used for this, with a depth of 512, though this results in slight memory wastage; alternatively, LUTRAM can be used without capacity or area waste, although maximum frequency might be lower.

3.3.5 Input Shift of Cascaded DSPs

Since there is a one-cycle delay between DSPs in different positions within the cascade when processing the same input filter data, downstream DSPs are one clock cycle behind upstream DSPs. To adjust the timing of data entering the cascaded DSPs, shift registers are inserted at the input. As mentioned, the input activation values are shared across PEs, making the data volume small, so this can be achieved with a simple Shift Register Logic (SRL).

However, for weight data that are not shared among PEs, using SRL would require a significant amount of logic resources, which is inefficient and would increase routing pressure. Pre-shifting the data before storing them in weight memory can address this issue, but this method introduces a delay of $DC - 1$ clock cycles whenever switching to the next two pixels after processing all output channels for two filters. This would result in significant overhead in layers with fewer input and output channels in the early stages of processing.

Therefore, instead of trying to shift the output, we chose to shift the read address. This approach requires each DSP in the cascade to have its own independent weight memory. Figure. 3.9 shows an example where the DSP cascade depth is 2; here, every two PEs and two weight memories are grouped together. Weight memory 0 stores the weights for the first-stage DSP in the two PEs, and weight memory 1 stores the weights for the second-stage DSP. The circuit in the figure offsets the address inputs of the two memories, allowing the address signal to fan out to a larger area. We added shifted reset circuits shown in Figure. 3.9 to the downstream pipeline registers so

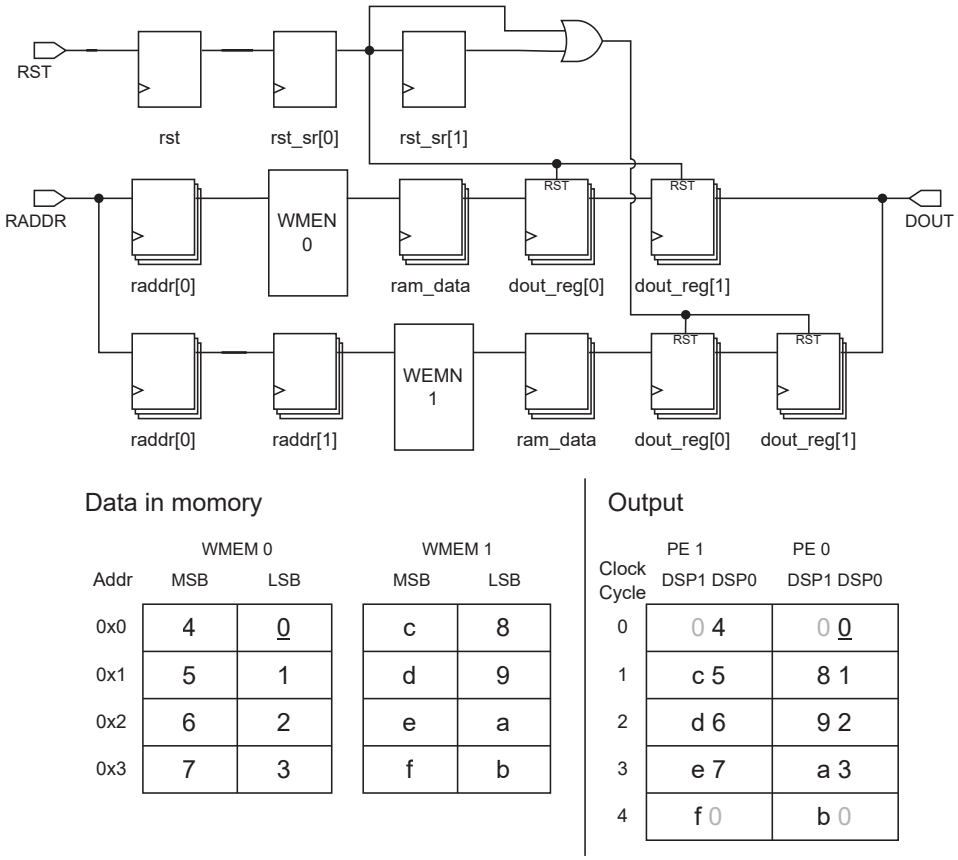


Figure 3.9: Example of adder shifting for weight memories.

they initialize to zero correctly at the start.

3.3.6 Weight Memory Binding

When the BRAM used for weight memory is configured with an 8-bit port, its depth is up to 4,096, which is far more than we need, leading to resource waste. Since the port width of URAM cannot be changed, this approach cannot use URAM as a substitute for BRAM in devices with limited BRAM but abundant URAM. Therefore, we package DC PEs into a group, group number is marked as G, so the total parallelism of the PE Array is represented as $P_{PE} = G \times DC$. Each group uses $DC \times DC$ DSPs for MAC operations, and DSPs in the same position in the cascade within the group share a BRAM, using different 8-bit words on one of the BRAM's ports.

For URAM, which has a fixed 64-bit port width (excluding ECC), lowering the port width by masking address bits to increase depth complicates the design and reduces frequency. When $DC = 4$, a group has four PEs, and four DSPs in the same position in the cascade will share this 64-bit width, but they can only use 32 bits. Since the number of PEs needs to be a multiple of $DC \times 2$, the group count must also be even. Thus, when using URAM, two groups of PEs can share the weight memory's port width.

For smaller devices, DC may be less than 4, reducing the number of PEs in a group

, and each PE would require greater weight memory depth, resulting in the need for more BRAMs. This allocation strategy, therefore, covers all device configurations.

These weight memories are configured in simple dual-port mode, with one port for the weight reads described above, operating at $clk2x$, and the other port for preloading the weights for the next layer, operating at $clk1x$. When using URAM as weight memory, it doesn't support dual-clock operation, so we use interleaved clock enable signals to create multi-cycle paths, reducing the routing difficulty of the initialization signals.

3.3.7 Accumulator

The accumulator can be implemented with either LUTs or the DSP48's SIMD24 mode, allowing simultaneous calculation of values for two output pixels.

When using DSP48, as shown in Figure. 3.10, the values sent to the accumulator are input into both 48-bit ports of the DSP, with different input register stages to ensure that the downstream accumulator register can fetch different inputs from both ports at the same time. In SIMD mode, the DSP's A and B ports are combined into a 48-bit input with a single-stage input register, while the C port uses a two-stage input register. We control the clock enable of the P register to update alternately at $clk2x$, ensuring data on odd clock cycles are added to data on even cycles. This also forms a multi-cycle path, with output converted to $clk1x$. When one output pixel is calculated, the input accumulator value's Z MUX is switched to 0 to clear the accumulated value, and the P register is updated to hold the partial sum of the new pixel, avoiding unnecessary stalls.

When using LUTs to construct the accumulator, the calculation process

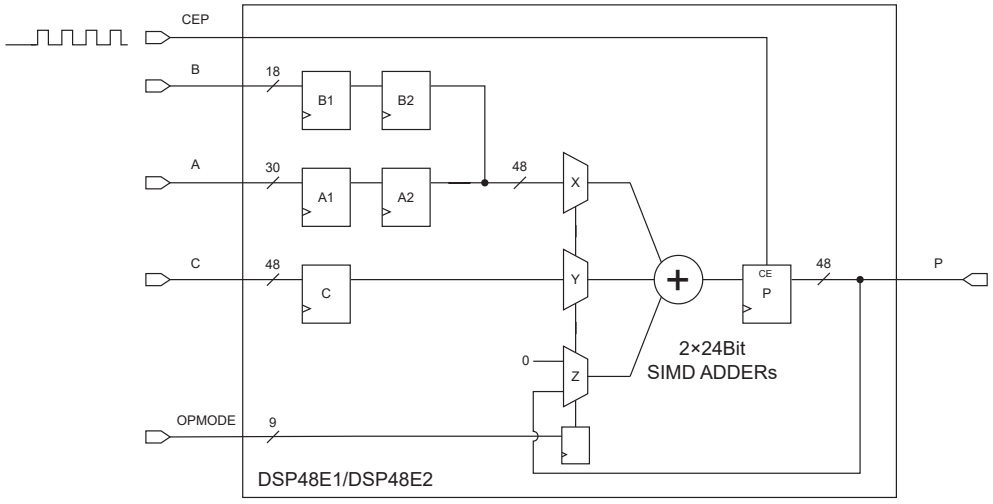


Figure 3.10: DSP48 for accumulator.

is the same as when using DSPs, but the placement of certain registers has been adjusted to reduce logic levels and maintain high frequency.

3.3.8 Quantization

The interconnect bus between modules in our accelerator that processes a single image also uses a bit width of $DC \times 2 \times PP$ for activations. Thus, each PE Array includes $DC \times 2$ quantization modules, each capable of processing PP activations in parallel, ensuring the PE Array can output as many activations as the input bus. These quantization modules are shared by all PEs, so each PE takes turns accessing them, and the number of PEs needs to be a multiple of $DC \times 2$. Therefore, if the output data rate of the PEs is too high (i.e., when Syn $< \frac{P_{PE}}{DC \times 2}$), waiting will occur.

In the example design, the quantization module follows the typical `madd` and `clamp` scheme, with BN folded into the `madd` computation. This quantization scheme can be replaced by more advanced ones. The `clamp` can output either signed results $[-127, 127]$ or unsigned results $[0, 255]$ depending on the activation function.

Before `clamp`, `round` is needed. There are various ways to handle `round` at 0.5; in our implementation, we chose floor$(x + 0.5)$ to simplify hardware, which rounds up for both positive and negative values at 0.5.

The quantization parameter memory module consists of LUTRAMs, also in simple dual-port mode, with the beta value pre-added by 0.5, allowing `round` by simply taking the integer part.

3.3.9 Running Time

The number of $clk1x$ clock cycles required for the PE Array module to execute a regular convolution layer is

$$CC_{PE} = Neu \times \max\left(Syn, \frac{P_{PE}}{DC \times 2}\right) \times H_{OFM} \times \left\lceil \frac{W_{OFM}}{PP} \right\rceil, \tag{3.3}$$

where Neu is the folding factor for neurons (output channels) and is equal to $\frac{C'_{\mathrm{OFM}}}{P_{PE}}$. P_{PE} is the parallelism of the PE Array, i.e., the number of PEs. C'_{OFM} is the output feature map channel count, rounded up to the nearest multiple of P_{PE}, with weights for any additional channels filled with zeros. Syn is the folding factor for synapses (usually input channels) and is equal to $\frac{K \times K \times C'_{\mathrm{IFM}}}{DC \times 2}$, where K is the kernel size, which is 1 for all but the first layer. Since the input feature map is typically the output from other computing modules, C'_{IFM} represents the actual channel count of data output by the previous module. After the first layer, it's also rounded up to the nearest multiple of P_{PE}; for the first layer, it's rounded up to the nearest multiple of $DC \times 2$, matching the channel count of the interconnect bus. The second term in the max() function represents the time required for a full polling cycle when multiple PEs share the quantization module. If Syn is smaller than this value, even though the PE Array can quickly process all synapses, the overall processing rate is reduced due to the time required for quantization.

3.4 Combine and Broadcast

The Combine and Broadcast modules are used to temporarily group multiple PE Arrays into a single large PE Array. The Broadcast module broadcasts the input data stream to multiple output ports, only processing the next payload when all output ports have successfully output a payload. The Combine module polls data from multiple input ports and sends the received data to the output port.

In the example accelerator shown in Figure. 3.2, the output feature map channels are evenly distributed across the PE Arrays, with a granularity of

$DC \times 2$ channels. PE Array 0 processes the first $DC \times 2$ channels, PE Array 1 processes the next $DC \times 2$ channels, alternating until all channels are allocated. As a result, the two PE Arrays continuously output $DC \times 2 \times 2$ channels. The Combine module merges such data in an interleaved manner, making the data structure and sequence equivalent to that of a single large PE Array.

Since C_{OFM} is allocated to two PE Arrays, each PE Array processes $\frac{C_{\mathrm{OFM}}}{2}$ channels and must still follow the padding rules of the PE Array. The value of $\frac{C_{\mathrm{OFM}}}{2}$ must be rounded up to the nearest multiple of P_{PE}, so C'_{OFM} must be rounded up to the nearest multiple of $2 \times P_{PE}$.

Due to the polling mechanism used by Combine, when using the configuration in Figure. 3.2 (with two input ports and the same parallelism for both upstream PE Arrays), the utilization of the two input ports and their buses is limited to 50%. Combining multiple PE Arrays usually occurs in the deeper layers, where the number of input synapses and output neurons is typically large, so the bus utilization limit and rounding to a larger value usually don't add extra overhead.

Due to the characteristics of the Dataflow architecture, when processing the same layer data sequentially, the two PE Arrays have a time difference. Downstream modules need to wait for the upstream module to output part of the data before starting. Therefore, switching from the serial to the parallel execution mode requires additional wait time, allowing downstream modules to clear the current load. The time required is approximately $\frac{K}{H}$ of the running time of the last serial-executed block, where K is the kernel size in the depthwise convolution layer, and H is the height of the output feature map.

3.5 PEDW Array

The PEDW (Processing Element for Depth-Wise convolution) Array is designed to compute depthwise convolution layers. Unlike regular convolution, depthwise convolution does not require summing across different channels, making the PEs in PE Array 0/1 very inefficient for this purpose.

We designed a fully pipelined PE with a initiation interval of 1, capable of completing the channel-wise computations for two filters per clock cycle and outputting the activation values for two pixels. When the kernel size $K = 3$,

each filter performs nine multiplications in parallel, summing them through an adder tree to produce the results for two pixels, which are then sent to the quantization module. Like the regular PE Arrays, which can perform multiple parallel operations across multiple output channels, the PEDW Array can also operate across multiple output channels simultaneously to improve parallelism.

Due to the significantly lower computational load of depthwise convolution layers compared to regular convolution layers, we use LUTs to implement these 8×8 multipliers. These LUT-based multipliers are interleaved among the DSPs used by PE Array 0/1. To avoid interfering with the high-frequency routing of these DSPs, these LUT-based multipliers operate on the $clk1x$ clock to ease their placement and routing requirements.

Each clock cycle, the PEDW Array requires a substantial number of input weights and feature map data. The control logic generates the access addresses for the weight memory, and it typically takes several clock cycles to read weights from memory. If the feature map data are received in the initial pipeline stages, multiple stages of pipeline registers are required to propagate this large amount of data. While this approach could effectively use an endpoint FIFO to reduce control set overhead, the large area required would be impractical. Another approach is to first read the weights from memory and then receive the feature map data from the input port when needed in the pipeline stage. However, if the input feature map data are not always ready, a global CE signal would be needed to pause the entire pipeline and wait for the input. After careful consideration, we decided to use a local CE signal before the feature map input stage, and then use an endpoint FIFO to buffer the payload, with only two pipeline stages requiring CE signal to solve this issue.

Since the PEDW Array outputs results every clock cycle, sharing quantization modules among multiple PEs, as in the PE Array, would lead to polling delays for all layers. Therefore, each PE has its own dedicated quantization module. When the number of PEs in the PEDW Array exceeds $DC \times 2$, the data bus would require polling, causing efficiency loss. Thus, the maximum parallelism is limited to $DC \times 2$, and it can be adjusted to a factor of $DC \times 2$. Because ReLU always follows the depthwise convolution layer, the quantization module disables signed output to save resources.

The weight memory for the PEDW Array typically requires a very large output bit width to support multiple parallel multiplications across several

PEs. However, the memory depth required is quite shallow, and since LUTs implement the multipliers, they are widely spread out. Therefore, using BRAM for this memory is not ideal, and we use simple dual-port LUTRAM instead. The computation order of the PEDW Array is relatively complex and will be explained in detail in later sections.

The time required for the PEDW Array to execute a depthwise convolution is

$$CC_{PEDW} = \left\lceil \frac{C'_{IFM}}{P_{PEDW}} \right\rceil \times \left\lceil \frac{W_{OFM}}{PP} \right\rceil \times H_{OFM}, \tag{3.4}$$

where PP represents the effective pixel count processed in parallel, which will be explained in the next section.

3.6 SWU

The SWU performs the `im2col` operation (also known as `unfold`) on-the-fly. It requires at least $K + 1$ true dual-port memory modules internally, with K modules used to output activations for K rows, and the remaining one for writing the input data while sampling. Once a row of filter output is completed and the input memory module has finished writing a full row of activation data, the roles of these $K + 1$ memory modules switch. The module that was used for writing switches to output, and the one that initially output sampled data for the smallest row number, which is no longer needed, switches to input mode to write the next row of input data. This process is repeated to complete sampling across the entire feature map.

The input and output submodules operate independently, controlled by a central state machine. They monitor each other's current row status and pause if one progresses faster than the other to synchronize. The input module can also start collecting data for the next layer before the output module finishes sampling the current layer. Near the boundary, when some rows are padded with zeros, fewer than K memory modules are used for output. At this point, memory modules that are no longer needed for output are used to writing input data, significantly reducing the waiting time required to switch to the next block when the accelerator runs continuously over two blocks.

Since each activation column needs to be individually addressable, even though the input and output buses handle two activations per clock cycle, SWU must separate them internally for sampling, where the data from odd

columns are stored into odd addresses and the even ones are stored into even addresses. The two write ports of each BRAM or URAM are used to store activations for odd and even columns, respectively. (Figure. 3.11)

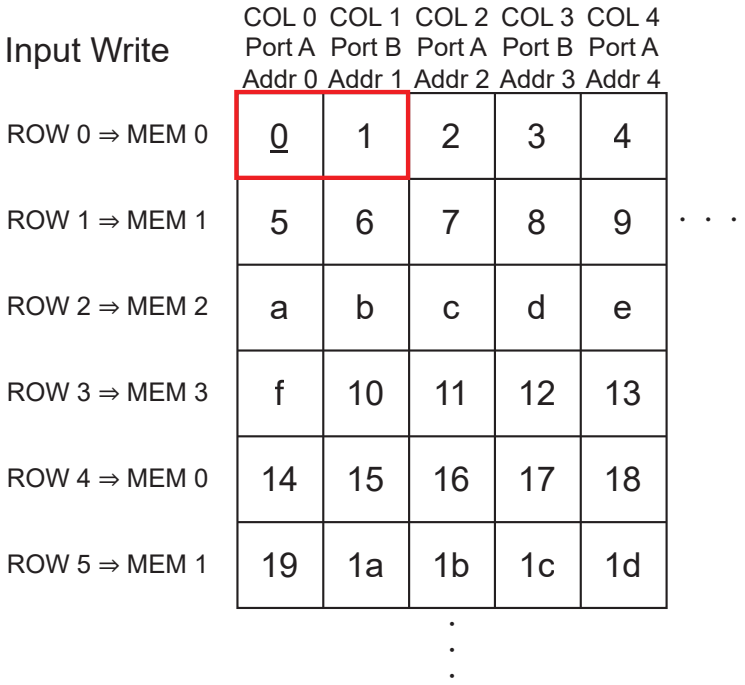


Figure 3.11: Writing data to the $K+1$ SWU memories, whereas data in the red frame are inputted and written at the same time.

Additionally, using $K+2$ row buffer memory modules instead of the minimum $K+1$ can avoid alternating runs between the previous layer module and the SWU output submodule. This occurs when the time taken to write a row of feature map data (at stride=2 for certain layers) is shorter than the sampling time for a row. Since only one row of data can be written in advance, when one row of data are written, the output of the previous module is blocked. After the line changed, two rows of new data are needed to start sampling, resulting in the alternating operation of the previous layer module and the SWU output submodule. By using $K+2$ modules, when the time to input a row is longer than the output time for a row, the entire block runtime is determined by the previous layer module, unaffected by the SWU output time. This will be explained in detail later when discussing the

running time.

When running MobileNetV2 with a configuration of $DC = 4$, each row buffer requires an on-chip memory configuration of 64-bit ports and a depth of 1,344, meaning at least three BRAMs are needed to provide sufficient capacity. However, in true dual-port mode, BRAMs have a maximum port width of 32 bits, so such a configuration cannot be achieved with only three BRAMs directly. For mid- to high-end devices where BRAMs are more plentiful, we extend the depth to 2,048, using four BRAMs or larger URAMs. While this approach may lead to resource waste, it avoids complex BRAM combinations that could increase routing complexity and reduce the maximum frequency.

On resource-constrained low-end devices, we use a custom structure to combine three BRAMs. These BRAMs operate at double-speed $clk2x$, doubling the effective port width to 64 bits, with the depth halved to 512. One solution is to use a cascaded configuration, but high-port-width BRAM cascading is only achievable on UltraScale and newer FPGAs. Direct cascading reduces the maximum frequency (tests on ZYNQ UltraScale+ speed grade 1 devices showed that cascading three BRAMs limits the max frequency at 593 MHz). To mitigate this, using multi-stage BRAM output registers in the cascade requires building with primitives, which increases output latency. A simpler and more efficient approach is to use the high bits of the address as the selection signal for BRAMs and connect multiple BRAMs via DEMUX and MUX.

SWU supports two operating modes: standard convolution mode and depthwise convolution mode, with some shared logic components across both modes:

3.6.1 Standard Convolution

In this mode, the SWU samples and outputs activations in the processing order of the PE Array described in previous sections, following the sequence: input channel (Syn-folded) $\rightarrow$ kernel row $\rightarrow$ kernel column $\rightarrow$ feature map row $\rightarrow$ feature map column. These data values are directly read from K memory modules.

When reading, the first port reads the data of a specific filter, and the second port reads the data of the next adjacent filter.

3.6.2 Depthwise Convolution

Unlike standard convolution, where there are separate input and output channel loops, depthwise convolution pairs input channels with output channels on a one-to-one basis, eliminating the need for two separate loops. Since the PE of the PEDW Array processes all pixels in a channel of the convolution kernel per clock cycle, there are no loops for the row and column of the kernel. Thus, the innermost loop becomes the channel loop.

The K memory modules in the SWU allow for parallel reads. If the necessary data are read directly from these modules, up to two partial convolution kernels can be output per clock cycle, with each kernel producing $1 \times K$ pixels for $DC \times 2$ channels. The PEDW Array requires a large amount of activation data per clock cycle to operate, specifically two full kernels, each containing $K \times K$ pixels of activations. To sample additional data, we added shift registers to the output ports of the memory modules.

When $K = 3$, each memory port requires at least two stages of shift registers, plus one additional stage to hold data when pipeline stalls prevent output. As shown in Figure. 3.12, each port, with 3 stages $\times$ 2 ports $\times$ K memory modules, can hold values for 6 columns and K rows. One output port accesses columns 0–2, and the other accesses columns 1–3. When output stalls, the data flow into the last reserve register, allowing one output port to access columns 2–4 and the other to access columns 3–5. If the channel loop is the innermost loop, an excessive number of shift register stages would be required—for instance, in MobileNetV2, the deepest channel count is 1,280, with 8-bit data, resulting in as many as $1280 \times 8 \times PP \times (2+1) \times K = 184320$ registers. This would be a significant challenge for device capacity and the fan-out of control signals needed for handling pipeline stalls.

Therefore, we adjusted the loop nesting, moving the channel loop outward. The innermost loop became the feature map row loop. We first process one complete row of a partial set of channels (a group of channels) and then slide across the row for all channels (channel group loop) before moving the sampling window to the next row (feature map column loop). The mode and computation order for the PEDW Array in SWU follow the sequence: feature map row $\rightarrow$ channel group $\rightarrow$ feature map column. The channel group size is tied to the parallelism of the PEDW Array; if the PEDW Array's parallelism reaches the maximum $DC \times 2$, only $(DC \times 2) \times 8 \times PP \times (2+1) \times K = 1152$ registers are needed, significantly reducing the number to a manageable level.

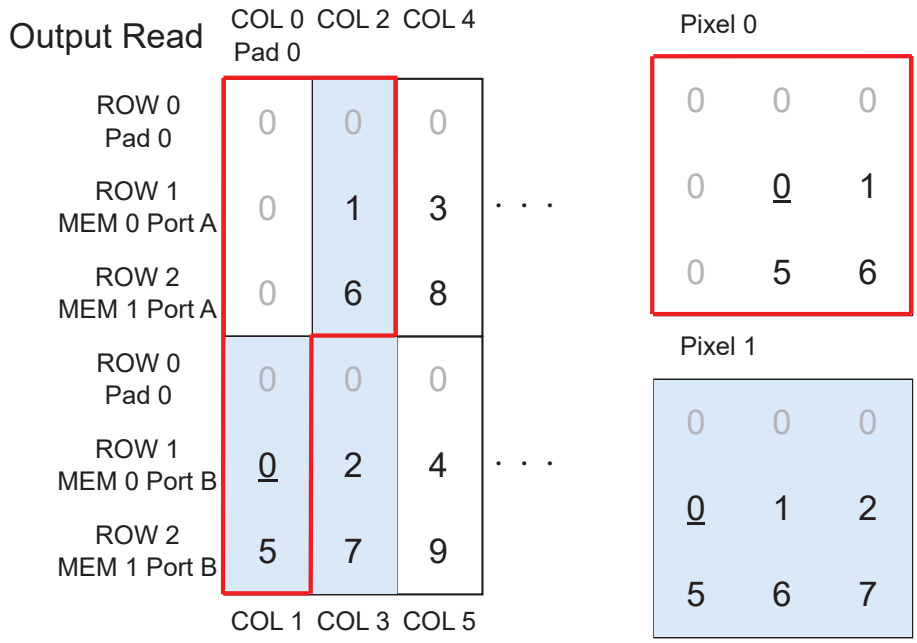


Figure 3.12: Reading data through shift registers, whereas data in red frame from columns 0-2 are for pixel 0 and data with light blue background from columns 1-3 are for pixel 1.

This approach requires a few extra clock cycles to load new row data into the shift register when the output row or channel group changes, potentially causing additional wait time when the feature map size is small. For stride=2, since the shift register always advances one slot per clock cycle, the data from output port 2 is invalid, reducing the effective pixel count processed in parallel in Formula 3.4 to 1.

Because this modified sampling and computation order differs, an additional channel-reordering module called DWC (Depth Wise convolution channel Converter) is added after the PEDW Array to restore the original channel-priority output order. When stride=2, half of the PEDW Array's output is invalid, and this module also removes invalid data. Its structure is similar to the SWU, requiring two row memory modules of the same size as those in the SWU—one for writing and one for reading. When the PEDW Array's parallelism is less than $DC \times 2$, the write port of the row buffer memory requires byte-enable support for bit-width conversion. The

SWU's dedicated output port for the PEDW Array is the only part of the computation module with a different width from other buses, with specific bit-width configurations shown in Figure. 3.13.

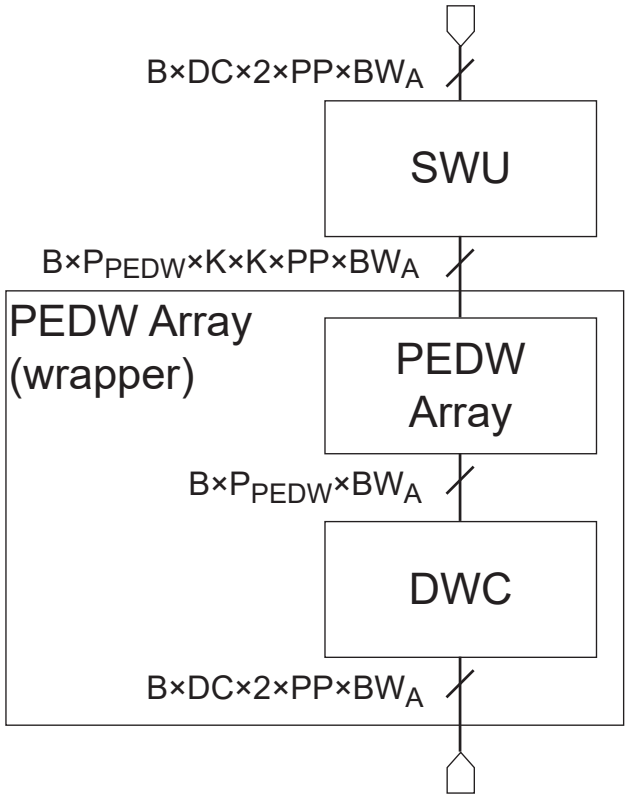


Figure 3.13: Bus bit-width changing in PEDW Array related path.

3.6.3 Running Time

The computation time for the SWU depends on various factors. In standard convolution mode, the SWU's data output rate matches the maximum input rate of the PE Array, so normally there is no bottleneck. However, if the stride equals 2, the number of memory modules is $K+1$, and the input time for one row is less than the output time for one row, a bottleneck may occur.

First, we need to define several variables. $CC_{SWU_{IN1}}$ represents the time required for the upstream module to output a row of data, and $CC_{SWU_{OUT1}}$ represents the time required for SWU to sample a row of data. In the case of stride is 2, when $CC_{SWU_{IN1}} > CC_{SWU_{OUT1}}$, the running time is equal to the

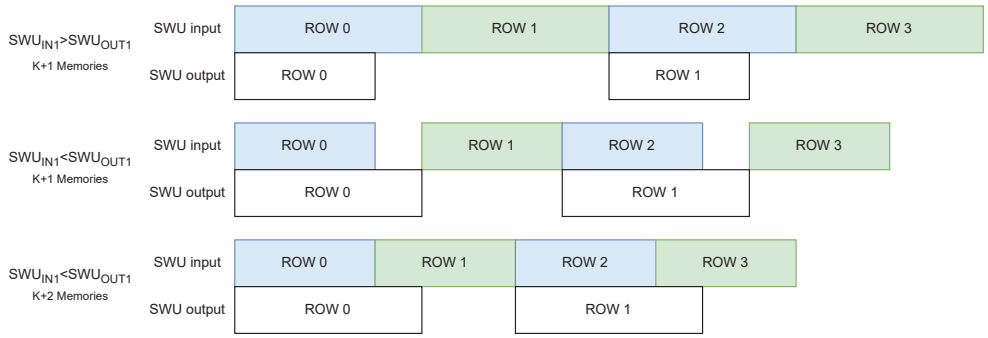


Figure 3.14: The running time of SWU under different conditions

time required by inputting data (the sum of the light blue area and green area in Figure. 3.14). On the other hand, when $CC_{SWU_{IN1}} < CC_{SWU_{OUT1}}$, the SWU's output time is the time taken by the SWU to output (the white area in Figure. 3.14) plus the time required for the previous layer to output half of all rows (the light green area in Figure. 3.14). This can be calculated with

$$CC_{SWU} = CC_{SWU_{OUT1}} \times H_{OFM} + CC_{SWU_{\alpha}}, \tag{3.5}$$

where

$$CC_{SWUOUT1} = \left\lceil \frac{W_{IFM}}{PP} \right\rceil \times \frac{C_{IFM}}{DC \times 2} \times K \times K; \tag{3.6}$$

$$CC_{SWU_{\alpha}} = CC_{SWU_{IN1}} \times H_{OFM}, \tag{3.7}$$

when $CC_{SWU_{IN1}} < CC_{SWU_{OUT1}}$ and stride $= 2$ and the number of memory modules is $K + 1$, otherwise $CC_{SWU_{\alpha}} = 0$.

In depthwise convolution mode, the calculation method differs, given by

$$CC_{SWUOUT1} = \left(\left\lceil \frac{W_{OFM}}{PP} \right\rceil + 2 \right) \times \frac{C_{OFM}}{P_{PEDW}}, \tag{3.8}$$

where $\frac{C_{OFM}}{P_{PEDW}}$ represents the number of times one row of the feature map is scanned, and $\left\lceil \frac{W_{OFM}}{PP} \right\rceil + 2$ is the time required to scan a row. The additional +2 accounts for the time needed to initialize the SRL when switching rows or channel groups. Both row and channel switches require SRL data initialization. The PP value is the actual number of parallel pixels, consistent with the PP in the PEDW Array's Formula 3.4. When stride $=$ 2, since the second port of the SWU output is invalid, PP is set to 1. In depthwise convolution mode, if conditions are met, $CC_{SWU_{\alpha}}$ is also added.

3.7 GAP

The GAP(Global Average Pooling) module performs global average pooling between the convolution and fully-connected layers in the model. The parallelism P_{GAP} can be adjusted from 1 to $DC \times 2$, but it must be a factor of $DC \times 2$. However, since GAP occupies minimal area and input-output bit-width conversion takes additional space, it is usually set to the maximum parallelism $DC \times 2$.

The GAP internally requires simple dual-port memory for the maximum channel count to temporarily store accumulated values for all channels. These memories use LUTRAM. The accumulation operation polls at the input end of these memories, with a maximum of P_{GAP} memories active at any time. After accumulating all values in a channel group, the results are multiplied by the reciprocal of the pixel count, rounded by adding 0.5 and getting the integer part. When GAP operates at maximum parallelism, it can input and output every clock cycle without causing a throughput bottleneck.

3.8 ADD

The ADD module combines data streams from the main and shortcut paths. Like the GAP module, the parallelism P_{ADD} can be adjusted from 1 to $2 \times DC$, though it is generally set to the maximum parallelism. Unlike other modules, the ADD module does not raise the ready signal early when valid is low to reduce pipeline gaps; it raises the ready signal only when the valid signals of both paths are high simultaneously.

After addition, the result is clamped to ensure no overflow before output. When ADD operates at maximum parallelism, it can input and output every clock cycle without causing a throughput bottleneck.

3.9 Weight

The Weight module manages weights and quantization parameters. The memory module part has been detailed in the PE Array section, so only the initialization part is covered here. The Weight module has a self-initialization circuit with an independent AXI port connected to the memory controller,

allowing it to independently load required weights and quantization parameters from the off-chip SDRAM.

Each memory module has its independent AXI master port, allowing it to prefetch the next layer's weights from off-chip memory into the unused on-chip memory space during execution.

The initialization process for standard convolution and depthwise convolution layers differs slightly: in the case of standard convolution, it transfers $DC \times DC$ weights (for DC PEs, i.e., one group) each time; in the case of depthwise convolution, it transfers the weights for one PE each time. When using a narrower AXI port or when one PE group (PE Array 0/1) or PE's (PEDW Array) weights exceed the AXI port's bit-width, weights are transferred across multiple clock cycles.

For quantization parameters, the maximum bit-width of 18 bits permitted by a single DSP is used, rounded up to 32 bits for transmission. The LSB half of the AXI bus transmits gamma, and the MSB half transmits beta. For each PE group in the PE Array 0/1, quantization parameters are transmitted over multiple clock cycles according to the actual AXI bus bit-width; for the PEDW Array, all PE quantization parameters are similarly transmitted over multiple cycles.

The time required to initialize the weights and quantization parameters for PE Arrays 0/1 is calculated as follows:

$$CC_{INIT} = (M \times Neu \times 2 \times Syn + N \times Neu) \times G, \tag{3.9}$$

where $M \times Neu \times 2 \times Syn$ is the time to initialize weights in one PE group and $N \times Neu$ is the time to initialize quantization parameters. M and N are folding factors applied when the bit-width of the data exceeds the AXI bus width. The folding factor M for weights is calculated by

$$M = \left\lceil \frac{BW_W \times DC \times DC}{BW_{AXI}} \right\rceil, \tag{3.10}$$

and N for quantization parameters by

$$N = \left\lceil \frac{BW_Q \times DC}{\frac{1}{2}BW_{AXI}} \right\rceil, \tag{3.11}$$

where BW_W is the bit-width of the weights, BW_Q is the bit-width of the quantization parameters and BW_{AXI} is the bit-width of the AXI bus.

On the other hand, the time needed to initialize weights and quantization parameters for the PEDW Array is given by

$$CC_{INITDW} = M \times C_{IFM} + N \times \frac{C_{IFM}}{P_{PEDW}}, \tag{3.12}$$

where M is the folding factor for weights

$$M = \left\lceil \frac{K \times K \times BW_W}{BW_{AXI}} \right\rceil, \tag{3.13}$$

and N for quantization parameters

$$N = \left\lceil \frac{BW_Q \times P_{PEDW}}{\frac{1}{2}BW_{AXI}} \right\rceil. \tag{3.14}$$

If the memory requirement of two adjacent blocks exceeds the design's memory capacity, initialization is divided into two parts: the first part is executed along with the PE Array/PEDW Array, while the second part initializes once the current layer releases weight memory space upon completion. At this point, the PE Array/PEDW Array will wait until this initialization completes before executing the next layer.

3.10 Controller

The controller module manages the START signals for all modules and provides the necessary operational parameters. It can be configured and monitored via the AXI-Lite bus.

When the next block needs to run, the controller identifies all related modules and marks them as pending. If a module is in the IDLE state, the START signal is activated, and the pending flag is cleared. For modules requiring weight initialization (such as the PE Array), both the PE Array and its initialization module must be IDLE before simultaneously triggering their START signals, preventing calculation errors caused by starting before weight initialization completes.

For initialization modules, the weight initialization and input initialization for the next image also have additional optional pending flags, which can be triggered at the start of initialization and cleared at the end for refined control. When all module pending states and initialization pending states are cleared, the block counter increments, and the next block is processed.

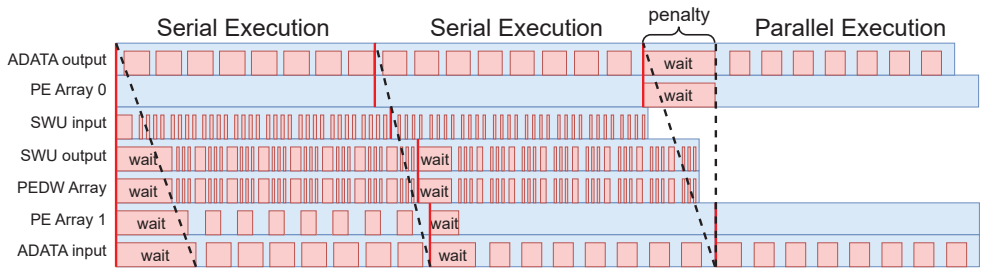


Figure 3.15: The optimal working state when executing multiple blocks, where the red area represents the stalling time.

The ideal operation state is shown in Figure. 3.15. (The red line marks task dispatch times, red area indicates waiting due to lack of input data or blocked by output port, and blue shows modules running normally. When module rates are inconsistent, modules with lower throughput may create bubbles.)

3.11 Improvement on Scalability

So far, the design has focused on edge computing devices like ZYNQ. When parallelism increases significantly, the SDRAM version without URAM may face severe performance degradation. To improve portability and scalability for achieving high performance with large parallelism, especially for high-capacity, high-performance data center accelerator cards, we have made several enhancements to the accelerator. In the previous design, to maintain address consistency between SDRAM and URAM versions, the address range in SDRAM version instructions was limited to match that of the URAM version. When targeting MobileNetV2, this capacity was set to 16 URAM units, or 512KiB, with each instruction accessing up to 512KiB $\times$ B.

To simplify data transfer, it is preferable for the tick index of an AXI burst transfer to correspond one-to-one with the sequence of parallel processed images, meaning the batch size should be padded to the nearest power of 2. This adds extra transfer time.

Table. 3.4 shows the impact of batch size B and PE parallelism P_{PE} on throughput when running MobileNetV2 with the SDRAM version at $clk2x =$ 600 MHz and $DC = 4$. Table. 3.4 shows FPS per DSP to indicate throughput per unit area. To visualize trends more clearly, results are presented in a

Table 3.4: FPS Matrix of SDRAM Version when Running MobileNetV2

B \ P_{PE}	8	16	24	32	40	48	56	64
1	197.9	375.7	468.6	633.6	597.8	666.2	611.3	631.6
2	395.1	729.5	887.2	1202.7	1103.8	1171.6	1048.3	1041.4
3	570.6	1030.7	1127.8	1439.5	1275.7	1182.7	1043.3	983.3
4	760.8	1374.3	1503.8	1919.3	1700.9	1577.0	1391.0	1311.0
5	884.0	1343.9	1228.1	1485.1	1258.5	1106.9	969.0	878.1
6	1060.8	1612.7	1473.7	1782.2	1510.2	1328.2	1162.8	1053.7
7	1237.7	1881.5	1719.4	2079.2	1761.9	1549.6	1356.6	1229.3
8	1414.5	2150.3	1965.0	2376.2	2013.6	1771.0	1550.3	1405.0
9	1236.7	1498.2	1228.8	1456.2	1203.9	1031.4	896.3	804.1
10	1374.2	1664.7	1365.4	1618.0	1337.6	1146.0	995.9	893.5
11	1511.6	1831.2	1501.9	1779.8	1471.4	1260.6	1095.5	982.8
12	1649.0	1997.6	1638.4	1941.6	1605.2	1375.2	1195.1	1072.1
13	1786.4	2164.1	1775.0	2103.4	1738.9	1489.8	1294.7	1161.5
14	1923.8	2330.6	1911.5	2265.2	1872.7	1604.4	1394.3	1250.8
15	2061.2	2497.1	2048.0	2427.0	2006.4	1718.9	1493.9	1340.2
16	2198.7	2663.5	2184.6	2588.8	2140.2	1833.5	1593.4	1429.5

Table 3.5: FPS per DSP Matrix of SDRAM Version when Running MobileNetV2

B \ P_{PE}	8	16	24	32	40	48	56	64
1	1.596	1.998	1.860	2.005	1.573	1.501	1.203	1.104
2	1.593	1.940	1.760	1.903	1.452	1.319	1.032	0.910
3	1.534	1.828	1.492	1.518	1.119	0.888	0.685	0.573
4	1.534	1.828	1.492	1.518	1.119	0.888	0.685	0.573
5	1.426	1.430	0.975	0.940	0.662	0.499	0.381	0.307
6	1.426	1.430	0.975	0.940	0.662	0.499	0.381	0.307
7	1.426	1.430	0.975	0.940	0.662	0.499	0.381	0.307
8	1.426	1.430	0.975	0.940	0.662	0.499	0.381	0.307
9	1.108	0.885	0.542	0.512	0.352	0.258	0.196	0.156
10	1.108	0.885	0.542	0.512	0.352	0.258	0.196	0.156
11	1.108	0.885	0.542	0.512	0.352	0.258	0.196	0.156
12	1.108	0.885	0.542	0.512	0.352	0.258	0.196	0.156
13	1.108	0.885	0.542	0.512	0.352	0.258	0.196	0.156
14	1.108	0.885	0.542	0.512	0.352	0.258	0.196	0.156
15	1.108	0.885	0.542	0.512	0.352	0.258	0.196	0.156
16	1.108	0.885	0.542	0.512	0.352	0.258	0.196	0.156

Table 3.6: FPS Matrix of URAM Version when Running MobileNetV2

B \ P_{PE}	8	16	24	32	40	48	56	64
1	197.9	375.7	468.6	634.3	598.6	667.5	612.6	633.3
2	395.7	751.4	937.3	1268.5	1197.1	1335.0	1225.1	1266.6
3	593.6	1127.0	1405.9	1902.7	1795.6	2002.4	1837.6	1899.8
4	791.5	1502.7	1874.6	2537.0	2394.2	2669.9	2450.1	2533.1
5	989.4	1878.4	2343.1	3171.0	2992.5	3336.9	3062.2	3165.9
6	1187.2	2254.0	2811.7	3805.2	3591.0	4004.3	3674.7	3799.1
7	1385.1	2629.7	3280.3	4439.4	4189.5	4671.7	4287.1	4432.2
8	1583.0	3005.4	3749.0	5073.6	4788.0	5339.1	4899.6	5065.4
9	1780.8	3380.9	4217.2	5707.0	5384.8	6002.7	5506.8	5692.9
10	1978.7	3756.6	4685.8	6341.1	5983.1	6669.7	6118.6	6325.5
11	2176.6	4132.2	5154.4	6975.2	6581.4	7336.7	6730.5	6958.0
12	2374.4	4507.9	5623.0	7609.3	7179.8	8003.6	7342.4	7590.5
13	2572.3	4883.5	6091.6	8243.4	7778.1	8670.6	7954.2	8223.1
14	2770.2	5259.2	6560.2	8877.5	8376.4	9337.6	8566.1	8855.6
15	2968.0	5634.8	7028.7	9511.6	8974.7	10004.5	9178.0	9488.2
16	3165.9	6010.5	7497.3	10145.7	9573.0	10671.5	9789.8	10120.7

heatmap, with better values in green and poorer values in red.

All other parameters are fixed at their maximum allowable values to avoid additional variables, so the throughput represents the theoretical maximum for the given parallelism. The results were obtained using behavioral simulation with Verilator [41]. In Tables. 3.4 and 3.5, the x-axis represents PE parallelism P_{PE}, and the y-axis represents batch size B. The DSP count per parallel image varies with P_{PE} values, specifically: 124, 188, 252, 316, 380, 444, 508, and 572.

The relevant content about the improvements discussed in this section and its subsections has been published in [42].

3.11.1 Increasing Batch Size Result in Negative Improvement

In Table. 3.5, we found a significant drop in FPS/DSP when batch sizes increase from 1 to 2, 2 to 3, 4 to 5, and 8 to 9. Additionally, total FPS tends to decrease as batch size increases, with this effect becoming more pronounced. This leads to an unexpected outcome: despite the increase in DSP usage and parallelism, the overall throughput actually declines.

Table 3.7: FPS per DSP Matrix of URAM Version when Running MobileNetV2

B \ P_{PE}	8	16	24	32	40	48	56	64
1	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
2	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
3	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
4	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
5	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
6	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
7	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
8	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
9	1.596	1.998	1.859	2.007	1.575	1.502	1.204	1.106
10	1.596	1.998	1.859	2.007	1.575	1.502	1.204	1.106
11	1.596	1.998	1.859	2.007	1.575	1.502	1.204	1.106
12	1.596	1.998	1.859	2.007	1.575	1.502	1.204	1.106
13	1.596	1.998	1.859	2.007	1.575	1.502	1.204	1.106
14	1.596	1.998	1.859	2.007	1.575	1.502	1.204	1.106
15	1.596	1.998	1.859	2.007	1.575	1.502	1.204	1.106
16	1.596	1.998	1.859	2.007	1.575	1.502	1.204	1.106

On the other hand, the simulation results of the URAM version are shown in Tables. 3.6 and 3.7. Compared with the simulation results of the previous SDRAM version, the throughput of the URAM version changes linearly when B changes.

We attempted to optimize the access pattern for off-chip SDRAM to reduce the impact of batch size variations on performance. First, we optimized and reused some instruction fields, expanding the maximum accessible off-chip memory per instruction from 512KiB $\times$ B to 256MiB $\times$ B.

Next, for running MobileNetV2, we allocated larger memory capacity, for example, 512KiB per image multiplied by the maximum batch size $B_{MAX} = 16$. The difference between 2MiB for $B = 4$ and 8MiB for $B = 16$ is negligible for high-capacity DDR4 SDRAM.

The base address of read/write instructions aligns with burst length $\times$ $B_{MAX} \times BW_{AXI}$. All values of B share identical instructions; when B is less than B_{MAX}, each read/write instruction accesses less space, skipping the subsequent space. The first data in the initial burst transfer of each instruction comes from the first parallel processed image, but for subsequent bursts, the first data may not come from the first image. The last data in the

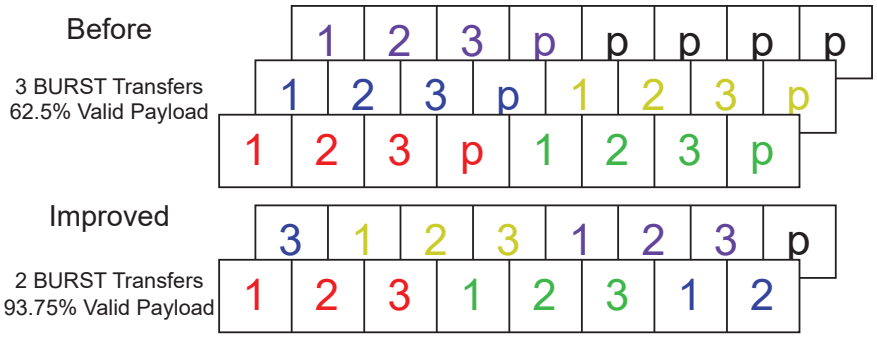


Figure 3.16: Burst transfer to off-chip SDRAM.

final burst may not be the last data from the last image, thus requiring some padding for write instructions and discarding padding for read instructions.

Figure. 3.16 compares the two methods for a burst length of 8, $B = 3$, and data transfer size of 5, where "p" denotes padding data. As shown, the new method requires less transfer time and is more efficient. Unlike the previous method where each burst had overhead, the new approach only incurs overhead on the last burst, which is negligible for large volumes of valid data.

After applying this improvement, the data transfer time for ADATA to off-chip memory changes, and the rounding-up parameter B' in Formula (3.1) becomes the original value B, with negligible overhead for the last burst.

3.11.2 Increasing PE Parallelism Result in Negative Improvement

From Table. 3.4, we observed that performance does not consistently increase with higher P_{PE}. Specifically, $P_{PE} = 16$ and $P_{PE} = 32$ show noticeably better performance compared to other values, as these values are factors of most layer channel counts in MobileNetV2. During actual operation, all layer channels must be padded to multiples of P_{PE}.

In Table. 3.6, which shows the throughput of the URAM version. A similar phenomenon is observed, though it is less pronounced than in the SDRAM version. This suggests that the URAM version is less affected. Notably, throughput slightly drops at $P_{PE} = 40$ and $P_{PE} = 56$ compared

Table 3.8: FPS Matrix of SDRAM Version when Running EfficientNetV1

B \ P_{PE}	8	16	24	32	40	48	56	64
1	159.4	306.9	398.6	503.5	549.3	588.4	572.4	575.0
2	318.4	599.1	760.7	965.9	1020.7	1049.6	990.7	963.1
3	463.1	855.4	988.5	1204.0	1200.3	1098.1	1005.6	943.7
4	617.5	1140.5	1318.0	1605.3	1600.5	1464.2	1340.8	1258.2
5	727.1	1158.2	1125.9	1335.1	1224.1	1071.7	962.4	868.4
6	872.5	1389.8	1351.1	1602.1	1468.9	1286.1	1154.9	1042.0
7	1017.9	1621.5	1576.2	1869.1	1713.7	1500.4	1347.3	1215.7
8	1163.4	1853.1	1801.4	2136.1	1958.6	1714.8	1539.8	1389.4
9	1058.8	1372.6	1184.9	1371.0	1204.6	1027.3	905.8	806.5
10	1176.5	1525.1	1316.5	1523.3	1338.5	1141.4	1006.4	896.1
11	1294.1	1677.6	1448.2	1675.6	1472.3	1255.6	1107.0	985.7
12	1411.8	1830.2	1579.9	1827.9	1606.2	1369.7	1207.7	1075.3
13	1529.4	1982.7	1711.5	1980.3	1740.0	1483.9	1308.3	1164.9
14	1647.1	2135.2	1843.2	2132.6	1873.8	1598.0	1409.0	1254.5
15	1764.7	2287.7	1974.8	2284.9	2007.7	1712.1	1509.6	1344.1
16	1882.4	2440.2	2106.5	2437.3	2141.5	1826.3	1610.2	1433.7

to adjacent values. However, in Figure. 3.7, it can be clearly observed that when $P_{PE} = 16$ and $P_{PE} = 32$, the results are significantly better than other values of P_{PE}.

We also evaluated the performance of EfficientNetV1, designed with Neural Architecture Search (NAS) technology [43] (removing SE blocks [44] that are unfavorable for hardware computation). NAS-designed networks often have more irregular shapes and non-uniform channel counts compared to manually designed networks, resulting in a lack of fixed factors. Simulation results with the SDRAM version are shown in Table. 3.8, where we observed that although throughput decreases with increasing P_{PE}, the trend is smoother than with MobileNetV2.

In fact, at $P_{PE} = 64$, half of the results' throughput exceeds that of MobileNetV2, even though the computational load of EfficientNetV1 is approximately 1.3 times that of MobileNetV2. This highlights a crucial issue: in early layers with fewer output channels, excess PEs produce invalid data, occupying quantization modules and data buses, which increases the computation load of the PEDW Array and degrades performance.

To address this issue, we added the effective PE count (in units of $DC \times 2$ PEs) for the last Neu fold to the instruction, and included logic in the

accelerator to skip quantization and output of invalid data. Now, the channel count only needs to be a multiple of $DC \times 2$ (or $DC \times 2 \times 2$ when merging two PE Arrays with Broadcast and Combine), significantly reducing overhead compared to the previous method, which required padding to P_{PE} multiples.

The clock cycle count required to execute one layer with the improved approach is calculated by

$$Neu' = \left\lfloor \frac{C_{OFM}}{P_{PE}} \right\rfloor \tag{3.15}$$

$$Rem = \left\lceil \frac{C_{OFM} \mod P_{PE}}{S_{PE}} \right\rceil \tag{3.16}$$

$$CC_{PE_{Rem=0}} = \max\left(Syn, \frac{P_{PE}}{S_{PE}}\right) \times Neu' \times \left\lceil \frac{W_{OFM}}{2} \right\rceil \times H_{OFM} \tag{3.17}$$

$$CC_{PE_{Rem\neq 0}} = CC_{PE_{Rem=0}} + \max(Syn, Rem) \times \left\lceil \frac{W_{OFM}}{2} \right\rceil \times H_{OFM}, \tag{3.18}$$

where Neu' is the value of Neu for the final fold when C_{OFM} is not a multiple of P_{PE}, and Rem is the clock cycle count occupied by the quantization module in the last Neu fold. When $Rem \neq 0$, some PE quantization outputs are skipped in the last fold, reducing the parameter in max(). When $Rem = 0$, the computation is consistent with the original method.

3.11.3 Implementation of `concat`

To enable our accelerator to handle more complex models where later layers require repeated reads of earlier layer outputs, a concatenation (`concat`) operation is essential. Thus, we implemented an almost overhead-free concat method.

Due to the limited capacity of on-chip memory, storing outputs from earlier layers on-chip is nearly impossible; therefore, we must store them in large off-chip memory. During writes, we allow a certain number of addresses

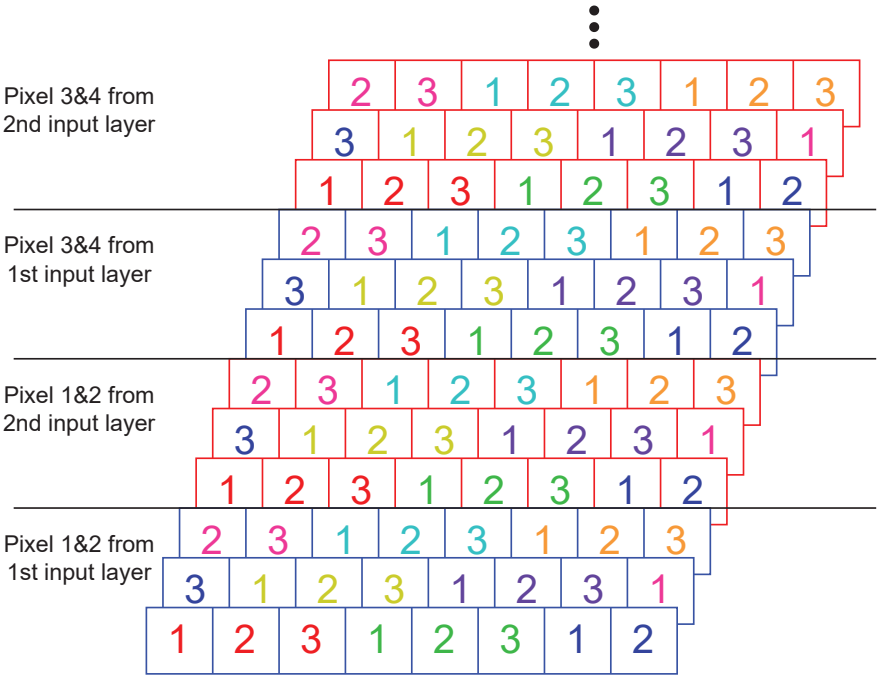


Figure 3.17: The implementation of `concat`.

to be skipped after each data segment. Similarly, the read process supports skipping. The alignment requirement of the base address for each burst was removed, meaning that the base address for the next access is only fully aligned after transferring $B \times$ burst length.

As shown in Figure. 3.17 (using the same example parameters as Figure. 3.16), $3 \times 8 = 24$ complete bursts must be completed to finish the transfer at the boundary between the red and blue sections. Then, the write pointer jumps to the start of the next blue section for the following write. When performing the `concat` operation, the red section is written after completing the first input layer, and subsequent layers write to the blue section, achieving a seamless `concat` operation during reads.

When DC is set to 4 and the burst length is 16, the minimum number of channels to skip or write to ensure address alignment is $4 \times 2 \times 16 = 128$. Since `concat` operations primarily occur in later layers/blocks, the impact of this higher minimum channel count is minimal. Additionally, the minimum

channel count can be reduced by sacrificing transfer efficiency (reducing burst length) in special cases.

This method also allows a layer to be divided into multiple segments for execution or a reverse residual block to be divided into two parts: the 1×1 convolution and 3×3 depth-wise convolution, along with the final 1×1 convolution. These parts can be further subdivided into smaller segments for independent execution. This flexibility enables the proposed accelerator to fit into smaller devices.

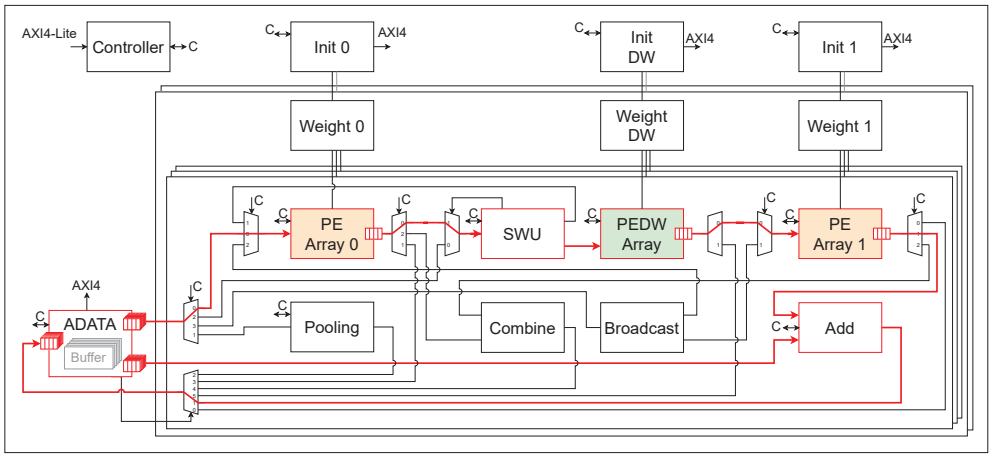
3.11.4 Utilize Wider AXI Port to Improve SDRAM Bandwidth Utilization

Our initial target device was the ZYNQ UltraScale+, whose PS-side AXI ports have a maximum bit width of 128 bits, matching the required bit width in the example configuration. When using the SDRAM version instructions, the initial layers fully utilize the ADATA MAXI0 port at 100% with a bit width of 128 bits. Since the ZYNQ UltraScale+ PS can only provide a maximum port width of 128 bits, increasing the width of the MAXI0 port does not improve performance due to the AXI slave port bottleneck on the PS side. In contrast, the URAM version requires very low bandwidth, imposing minimal demand on the port bandwidth.

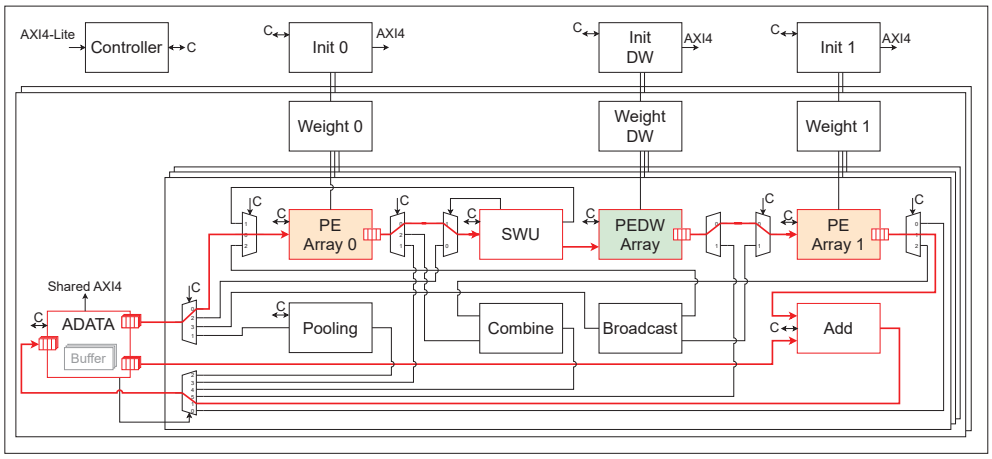
When we migrated our design to larger Virtex devices, the positions of the device's I/O pads determined the placement of the softcore memory controller MIG. This placement prevented the MIG from providing a large, continuous blank area for our accelerator, requiring the entire accelerator instance to be split into smaller, independent instances. In this split configuration, accelerator instances share the MIG controller, which not only increases the memory access volume related to weights but also introduces randomness in memory access, drastically reducing throughput.

After identifying this issue, we allowed multiple accelerator instances to share the top-level port, enabling larger accelerators to be split into smaller parts to address the problem. Additionally, this approach reduces the potential frequency drop risk by minimizing signal fan-out over large areas. The weight-related memory modules can be independently owned by each instance to handle the issue of non-continuous deployment space, or they can be shared proportionally to minimize BRAM usage.

Controller modules, memory initialization submodules, and other shared



(a) ADATA is not shared across instances



(b) ADATA is shared across instances

Figure 3.18: Multiple accelerator instances sharing the top-level port.

resources are synchronized across instances sharing the top-level port to ensure fully synchronized operation of computation modules in each instance. ADATA has two configurations: one where it is not shared across instances but includes multiple buffers and data paths to handle data from different instances (as shown in Figure. 3.18a). This configuration reduces the interconnection network between instances when deployed in non-continuous spaces but requires additional area. Another configuration involves ADATA being not shared among multiple instances, with only ADATA of instance 0 responsible for communication control with the AXI slave, while the remaining instances handle only the corresponding data transfer (as shown

in Figure. 3.18b). This configuration is used primarily in continuous spaces to achieve higher throughput.

Aside from these shared and partially shared modules, all other modules are completely independent across instances, maximizing the division into separate parts to minimize space requirements during routing.

Due to the limited range of AXI bus width values, the configurations achievable with this improvement are constrained. The number of images processed in parallel must be a multiple of powers of 2, such as 2, 4, 6, or 8. While padding with invalid data could allow odd-numbered values, this would contradict the original intent of the improvement.

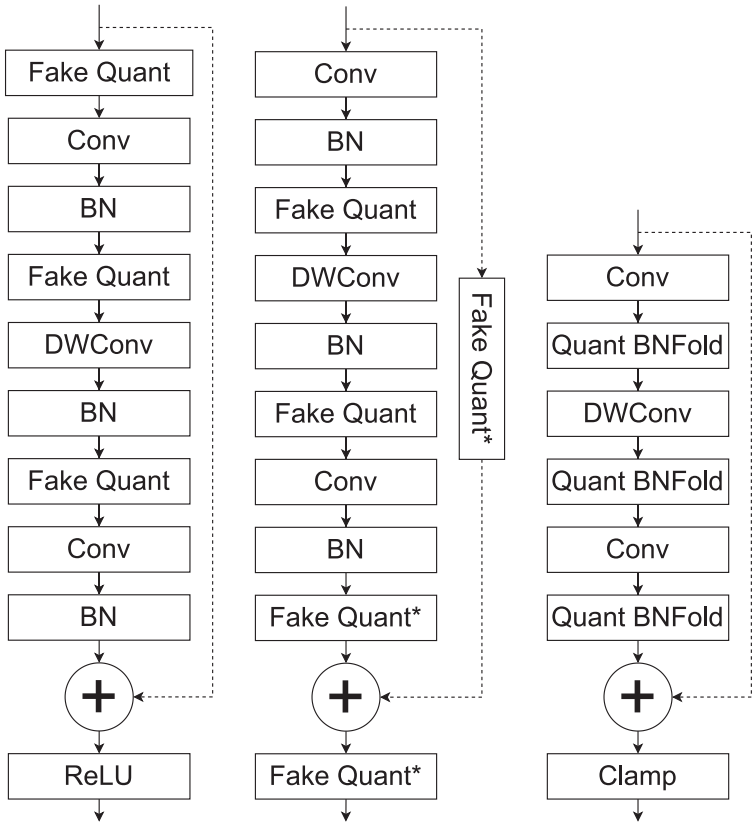
Additionally, it is important to note that the AXI4 protocol defines an upper limit for bus width, preventing ports wider than 1,024 bits from directly interfacing with existing systems. Consequently, in the example accelerator configuration, the port can be expanded up to 4 times its original width at most.

Furthermore, the MIG has a maximum port width of 512 bits. Since the off-chip interface of DDR memory operates in half-duplex mode and cannot read and write simultaneously, the effective bandwidth during simultaneous read and write operations is equivalent to a 256-bit port. Thus, when using DDR memory, expanding the port width to 512 bits may yield only limited performance gains.

Chapter 4

Experiments

4.1 Quantized Models



(a) Quantization for CPUs.

(b) Quantization for hardware during training.

(c) Quantization for hardware during inference.

Figure 4.1: Difference between quantization schemes.

Table 4.1: FPGA Devices

	LUT	LUTRAM	FF	BRAM	URAM	DSP
ZU7	230400	101760	460800	312	96	1728
ZU3	70560	28800	141120	216	0	360
VU13P	432000×4	197760×4	864000×4	672×4	320×4	3072×4
7Z007S	14400	6000	28800	50	0	66
7Z010	17600	6000	35200	60	0	80

First, we trained the 8-bit quantized MobilenetV2 for evaluation. As described before, compared with quantized models for CPU (Figure 4.1a), all the values should be quantized in the models for hardware. Figure 4.1b is the quantization method at the initial stage of training. The Fake Quant layers with * mark mean they share the quantization parameters.

The quantization scheme used is LSQ [45]. Signed values are quantized to $[-127, 127]$, and unsigned values are quantized to $[0, 255]$. After finishing the training with fake quantization modules, the fake quantization modules are replaced with real ones, and the BN layers are folded with the quantization parameters.

Due to the quantization parameters being quantized to 18 bits, the accuracy can drop. Fine tune on fully quantized modules is required to restore accuracy. The scales of the feature maps between blocks tend to become equal in a fine-trained model. So removing quantization on the shortcut path and replacing the quantization after the `add` with a `clamp` function does not affect the accuracy (Figure 4.1c). After training, MobileNetV2 got 71.546% Top-1 accuracy on ImageNet.

4.2 Performance Estimation

The first experiment focuses on edge computing with the ZYNQ UltraScale+ platform. Two FPGAs are selected for this stage: the mid-range device XCZU7EV-FFVC1156-2-E (referred to as ZU7) and the cost-optimized device XCZU3EG-SFVC784-1-I (referred to as ZU3), with their available resources listed in Table. 4.1.

We deploy the accelerator shown in Figure. 3.2, optimized for residual

blocks, on these two devices. The weight memory and other configurations are sufficient for running MobileNetV2. Based on the hardware resources of these FPGAs, particularly the number of DSPs and URAMs, we selected reasonable parallelism parameters. To simplify the design, most parameters of the accelerator are shared between the two devices.

The content of the subsections not related to improvements has been presented in [36], whereas the subsections addressing improvements have been published in [42].

4.2.1 Configurations

As explained in previous sections, DSP cascade height is closely tied to the computation density of a single PE, the maximum parallelism of various components, and the maximum bus bit width, so it is best set to the maximum value of 4. Given that the DSP count ratio between the two devices is 4.8 and that the ZU7 has a larger area, reducing the number of DSPs used can facilitate achieving the target frequency. To this end, we set the batch size $B = 4$ for the ZU7 and $B = 1$ for the ZU3. These values allow us to maintain DSP utilization at approximately 80%, minimizing congestion. The maximum supported configuration for both devices is $P_{PE} = 32$, while P_{PEDW} is set to its upper limit of 8, which both devices can handle.

With this configuration for running MobileNetV2, the weight memory required by the PE Array when using Broadcast and Combine modes consists of four 32-bit wide memories with a depth of 1,600 per group. Without Broadcast and Combine, the required depth doubles to 3,200. To demonstrate the minimal on-chip memory usage of our accelerator, we chose parameters with minimal capacity. However, it should be noted that using larger weight and quantization parameter memories can reduce extra wait time from segmented execution initialization, slightly improving throughput. With a word width of 32 bits, the BRAM depth is 1,024, so we rounded 1,600 up to 2,048, totaling two BRAMs in depth. As a result, $2 \times DC$ BRAMs per group, and group counts $G = \frac{P_{PE}}{DC} = 8$, the BRAMs used in total for the weights are $2 \times DC \times G = 64$ per PE Array.

The quantization parameter memory required for the PE Array is eight memories per group (four for gamma and four for beta), each with an 18-bit width and a depth of 30. As previously mentioned, this memory is suitable for LUTRAM. Each LUT has a capacity of 64 bits, configurable as 1×64 or

2×32. In our design, we consistently use the 1×64 configuration, so we round 30 up to 64.

The weight memory required by each PE in the PEDW Array has a width based on the actual weight width $K_{MAX} \times K_{MAX} \times BW_W$, with a depth of 120. The quantization parameter memory has a width of 18 and matches the weight memory's depth, rounded from 120 to 128.

4.2.2 Throughput Calculation

In accelerators interconnected with a Dataflow structure, all modules are pipelined, so each module relies on the previous module's output during operation. If one module takes longer than the others, it intermittently blocks the upstream module's output and cannot supply sufficient data to the downstream module, leading to extra wait time. Thus, the longest-running module determines the overall accelerator throughput.

On the other hand, in accelerators that use an Overlay processing structure, the time required to run a task equals the sum of all its subtask durations, determining the throughput. In CNN accelerators, each layer's overlay execution task constitutes a subtask, and the total time for all layers equals the time required to run the entire network.

In previous sections, we described the calculation of each module's runtime, allowing us to predict the total model execution time. As shown in Figure. 3.15, the time required to execute multiple layers within a block using the Dataflow structure is approximately equal to the time needed for the longest-running module in the block. This is represented as a parallelogram in Figure. 3.15 because later modules in the current block finish later, but earlier modules are already processing the next block, so the delay can be ignored. The total runtime is approximately the sum of the runtime for all blocks or layers, and the additional time CC_P(penalty) mentioned in Section. 3.4:

$$\left(\sum_i \begin{matrix}\max(CC_{AREAD_i}, CC_{AWRITE_i}, CC_{PE0_i}, CC_{PEDW_i}, CC_{PE1_i},\\ CC_{SWU_i}, CC_{INIT0_{i+1}}, CC_{INITDW_{i+1}}, CC_{INIT1_{i+1}})\end{matrix}\right) + CC_P. \tag{4.1}$$

We mainly deploy our accelerator on UltraScale+ series FPGAs. Due to the limitations of the maximum frequency (333 MHz) of the ZYNQ platform's PS AXI ports and various hard resources in this series, our accelerator achieves a maximum $clk2x$ frequency of 600 MHz on standard

Table 4.2: Estimated, Simulated, and Actual Performance

	Est. CC	Sim. CC	Act. CC	Act. FPS	Act. Delay
ZU7 URAM4	510032	507840	510589	2350	1.70ms
ZU7 SDRAM4	664656	671156	674097	1780	2.25ms
ZU3 SDRAM1	510032	508333	511976	586	1.71ms

0.85V UltraScale+ FPGA devices, corresponding to a $clk1x$ of 300 MHz. Achieving a $clk2x$ frequency above 600 MHz on UltraScale+ FPGAs is challenging. This limitation arises because the upstream ready signals of SWU and ADATA form critical paths. Due to the path-switching mechanism, handshake registers cannot be added to these paths, restricting the $clk1x$ paths to a maximum of 300 MHz. Therefore, we estimate throughput based on this frequency (using calculations and later simulations), and after synthesis and routing, the actual frequency will be used to measure actual throughput on the evaluation board.

The throughput is calculated as

$$FPS = \frac{f_{clk1x}}{CC} \times B, \tag{4.2}$$

where f_{clk1x} is the control/transfer frequency for $clk1x$, as the formula from previous sections is based on this frequency. For the URAM version, there is an initial time cost to preload images onto the on-chip inter-block feature map memory before processing the first batch. However, as images are processed continuously in batches, data for the next batch are already prefetched onto the chip before the first batch finishes, avoiding additional time. Thus, the initial loading time for the first batch is considered a bias, and its impact on the average runtime per batch decreases as more batches are processed. Therefore, we ignore this bias when discussing the runtime of the URAM version.

We present the calculation results in Table. 4.2 and the resolution is standard 224×224, where it is clear that, due to image polling on the AXI port for parallel image processing, the ZU7 SDRAM cycle count is significantly higher than the other two. The estimated time for ZU3 SDRAM1 and ZU7 URAM4 is consistent, as with a batch size of 1, there is no polling of the data across multiple images, preventing bus congestion; thus, the final term B' in Formula (3.1) is 1, leading to identical estimated values.

4.2.3 Simulation

We implemented the accelerator's various modules in RTL using Verilog HDL. We simulated our accelerator with Verilator [41], and the pre-synthesis behavioral simulation results are shown in Table. 4.2. Due to the platform's reliance on numerous proprietary bus interconnection IPs from AMD (Xilinx), open-source tools like Verilator are unable to simulate these private IPs. As a result, only behavioral-level simulations were conducted on the accelerator itself. The estimated throughput from our model closely matches the simulated throughput, confirming the accuracy of our throughput estimation model.

Through simulation, we observed that for MAC operations with the same number of DSPs, SDRAM1 under extremely high parallelism ($P_{PE} = 128$) achieves only 26.5% of the throughput of ZU7 SDRAM4 at $P_{PE} = 32$. The latency advantage is only 6.0%, due to the excessive I/O overhead incurred by high parallelism when processing layers with fewer output channels.

According to the simulation, using $K + 2$ memory in the SWU instead of $K + 1$ improved performance by approximately 3.7% for the ZU3 SDRAM1 and ZU7 URAM4 configurations. However, no significant change was observed for ZU7 SDRAM4. The improvement from adding more memory is minimal.

Compared to using parallel execution mode for the final layers, ZU7 URAM4 requires an average of 75,000 additional clock cycles, and ZU7 SDRAM4 requires an average of 66,000 additional cycles in designs without Broadcast or Combine, according to the simulation results. The BRAM capacity of ZU3 cannot support this configuration to independently run the head convolution layer. Switching from serial execution mode to parallel execution mode using Broadcast and Combine incurs a worst-case computational overhead of 7,200 clock cycles. However, simulation waveforms indicate an actual overhead of 5,241 clock cycles. This modest cost results in a significant improvement in throughput, making the trade-off highly favorable.

4.2.4 Simulation of the Design with Improvements

We simulated the proposed improvements using Verilator, with all parameters set to their maximum allowed values except for the parameters under test, to avoid interference. Starting with the improvement from Section 3.11.1, results for the SDRAM version are shown in Table. 4.3-4.6.

Table 4.3: FPS Matrix of SDRAM Version when Running MobileNetV2 with Improvement 3.11.1

B \ P_{PE}	8	16	24	32	40	48	56	64
1	197.9	375.7	468.6	633.6	597.8	666.2	611.3	631.6
2	395.1	729.5	887.2	1202.7	1103.8	1171.6	1048.3	1041.4
3	581.4	1062.0	1248.5	1641.5	1477.0	1434.1	1273.0	1232.7
4	760.8	1374.3	1503.8	1919.3	1700.9	1577.0	1391.0	1311.0
5	933.5	1625.6	1674.9	2121.7	1837.2	1664.3	1463.0	1352.1
6	1099.9	1838.9	1806.9	2236.7	1920.2	1710.1	1509.0	1376.8
7	1260.0	2004.8	1897.7	2317.0	1972.4	1744.3	1535.4	1393.8
8	1414.5	2150.3	1965.0	2376.2	2013.6	1771.0	1550.3	1405.0
9	1551.9	2278.9	2010.2	2424.2	2046.7	1786.8	1562.1	1412.2
10	1670.4	2365.9	2047.8	2464.3	2071.8	1799.6	1571.7	1416.8
11	1781.7	2437.4	2079.6	2498.0	2089.8	1810.0	1578.8	1420.5
12	1886.5	2500.3	2107.0	2524.4	2104.8	1818.2	1583.1	1423.4
13	1972.7	2548.4	2130.6	2544.6	2117.3	1823.3	1586.2	1425.6
14	2053.1	2591.1	2151.4	2561.7	2127.6	1827.3	1589.0	1427.6
15	2128.2	2629.2	2169.6	2576.4	2135.2	1830.6	1591.3	1428.7
16	2198.7	2663.5	2184.6	2588.8	2140.2	1833.5	1593.4	1429.5

Table 4.4: FPS per DSP Matrix of SDRAM Version when Running MobileNetV2 with Improvement 3.11.1

B \ P_{PE}	8	16	24	32	40	48	56	64
1	1.596	1.998	1.860	2.005	1.573	1.501	1.203	1.104
2	1.593	1.940	1.760	1.903	1.452	1.319	1.032	0.910
3	1.563	1.883	1.652	1.732	1.296	1.077	0.835	0.718
4	1.534	1.828	1.492	1.518	1.119	0.888	0.685	0.573
5	1.506	1.729	1.329	1.343	0.967	0.750	0.576	0.473
6	1.478	1.630	1.195	1.180	0.842	0.642	0.495	0.401
7	1.452	1.523	1.076	1.047	0.742	0.561	0.432	0.348
8	1.426	1.430	0.975	0.940	0.662	0.499	0.381	0.307
9	1.391	1.347	0.886	0.852	0.598	0.447	0.342	0.274
10	1.347	1.258	0.813	0.780	0.545	0.405	0.309	0.248
11	1.306	1.179	0.750	0.719	0.500	0.371	0.283	0.226
12	1.268	1.108	0.697	0.666	0.462	0.341	0.260	0.207
13	1.224	1.043	0.650	0.619	0.429	0.316	0.240	0.192
14	1.183	0.984	0.610	0.579	0.400	0.294	0.223	0.178
15	1.144	0.932	0.574	0.544	0.375	0.275	0.209	0.167
16	1.108	0.885	0.542	0.512	0.352	0.258	0.196	0.156

Table 4.5: FPS Matrix of URAM Version when Running MobileNetV2 with Improvement 3.11.1

B \ P_{PE}	8	16	24	32	40	48	56	64
1	197.9	375.7	468.6	634.3	598.6	667.5	612.6	633.3
2	395.7	751.4	937.3	1268.5	1197.1	1335.0	1225.1	1266.6
3	593.6	1127.0	1405.9	1902.8	1795.7	2002.4	1837.6	1899.9
4	791.5	1502.7	1874.6	2537.0	2394.2	2669.9	2450.1	2533.1
5	989.4	1878.4	2343.1	3171.2	2992.7	3337.2	3062.4	3166.2
6	1187.2	2254.0	2811.8	3805.3	3591.1	4004.5	3674.9	3799.3
7	1385.1	2629.7	3280.3	4439.5	4189.6	4671.8	4287.2	4432.4
8	1583.0	3005.4	3749.0	5073.6	4788.0	5339.1	4899.6	5065.4
9	1780.8	3381.0	4217.5	5707.7	5386.4	6006.2	5511.7	5698.4
10	1978.7	3756.7	4686.1	6341.7	5984.7	6673.5	6124.0	6330.9
11	2176.6	4132.3	5154.6	6975.8	6583.1	7340.5	6736.3	6963.0
12	2374.4	4508.0	5623.2	7609.8	7181.4	8007.7	7347.7	7594.9
13	2572.3	4883.6	6091.7	8243.8	7779.6	8674.6	7958.4	8226.6
14	2770.2	5259.2	6560.3	8877.8	8377.9	9340.7	8569.1	8858.2
15	2968.0	5634.9	7028.7	9511.8	8976.1	10006.0	9179.6	9489.5
16	3165.9	6010.5	7497.3	10145.7	9573.0	10671.5	9789.8	10120.7

Table 4.6: FPS per DSP Matrix of URAM Version when Running MobileNetV2 with Improvement 3.11.1

B \ P_{PE}	8	16	24	32	40	48	56	64
1	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
2	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
3	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
4	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
5	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
6	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
7	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
8	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
9	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
10	1.596	1.998	1.860	2.007	1.575	1.503	1.206	1.107
11	1.596	1.998	1.860	2.007	1.575	1.503	1.205	1.107
12	1.596	1.998	1.860	2.007	1.575	1.503	1.205	1.106
13	1.596	1.998	1.859	2.007	1.575	1.503	1.205	1.106
14	1.596	1.998	1.859	2.007	1.575	1.503	1.205	1.106
15	1.596	1.998	1.859	2.007	1.575	1.502	1.205	1.106
16	1.596	1.998	1.859	2.007	1.575	1.502	1.204	1.106

Clearly, as the batch size B increases, throughput also increases for each P_{PE} value. The trend in throughput change is shown in Figure. 4.2, where we observe that significant throughput increases only occur when P_{PE} is relatively small. For example, $P_{PE} = 8$ and $P_{PE} = 16$ are highlighted, showing that as P_{PE} increases, the throughput increment from increasing B diminishes. Since this improvement minimally affects the URAM version, it is not discussed separately.

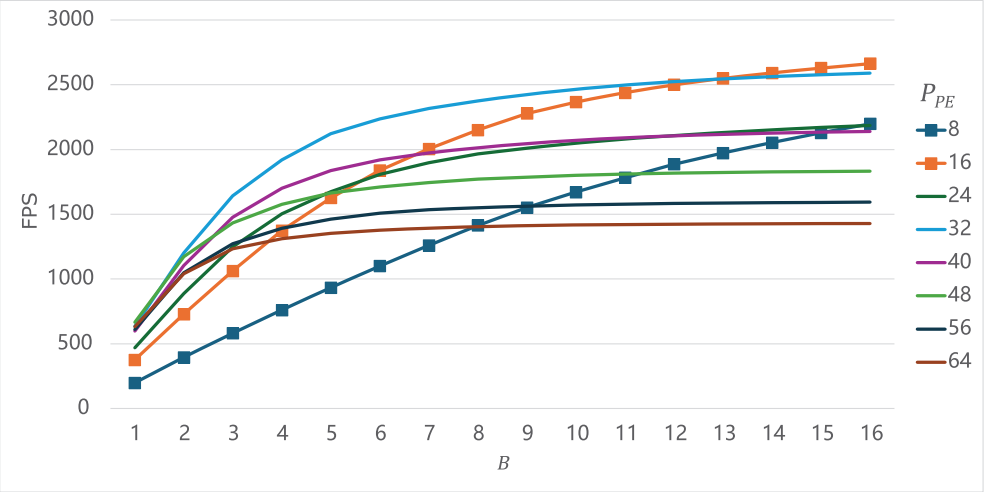


Figure 4.2: The trend in throughput variation as batch size increases.

Results of the improvement in Section 3.11.2 are listed in Table. 4.7-4.10, where increasing P_{PE} raises throughput for each B value. Figure. 4.3 illustrates the throughput trend, showing that results improve as the B value approaches the nearest power of 2. While the URAM version also benefits from this improvement, its results are similar to those achieved when both improvements are applied, as shown later.

Table 4.7: FPS Matrix of SDRAM Version when Running MobileNetV2 with Improvement 3.11.2

B \ P_{PE}	8	16	24	32	40	48	56	64
1	197.9	383.6	486.3	640.1	653.8	695.3	697.9	739.4
2	395.1	744.4	936.4	1216.7	1241.4	1315.9	1320.7	1395.0
3	570.6	1050.4	1302.0	1517.8	1543.4	1619.4	1624.3	1698.3
4	760.8	1400.6	1736.0	2023.7	2057.8	2159.2	2165.7	2264.4
5	884.0	1364.0	1568.7	1705.5	1738.0	1784.5	1804.8	1829.7
6	1060.8	1636.7	1882.5	2046.6	2085.6	2141.4	2165.8	2195.6
7	1237.7	1909.5	2196.2	2387.7	2433.2	2498.3	2526.8	2561.6
8	1414.5	2182.3	2510.0	2728.8	2780.8	2855.2	2887.8	2927.5
9	1236.7	1541.1	1641.8	1694.2	1708.5	1715.7	1723.1	1725.9
10	1374.2	1712.4	1824.2	1882.4	1898.3	1906.3	1914.6	1917.6
11	1511.6	1883.6	2006.7	2070.7	2088.2	2096.9	2106.0	2109.4
12	1649.0	2054.9	2189.1	2258.9	2278.0	2287.5	2297.5	2301.1
13	1786.4	2226.1	2371.5	2447.1	2467.8	2478.2	2488.9	2492.9
14	1923.8	2397.3	2553.9	2635.4	2657.7	2668.8	2680.4	2684.7
15	2061.2	2568.6	2736.4	2823.6	2847.5	2859.4	2871.9	2876.4
16	2198.7	2739.8	2918.8	3011.9	3037.3	3050.0	3063.3	3068.2

Table 4.8: FPS per DSP Matrix of SDRAM Version when Running MobileNetV2 with Improvement 3.11.2

B \ P_{PE}	8	16	24	32	40	48	56	64
1	1.596	2.040	1.930	2.026	1.721	1.566	1.374	1.293
2	1.593	1.980	1.858	1.925	1.633	1.482	1.300	1.219
3	1.534	1.862	1.722	1.601	1.354	1.216	1.066	0.990
4	1.534	1.862	1.722	1.601	1.354	1.216	1.066	0.990
5	1.426	1.451	1.245	1.079	0.915	0.804	0.711	0.640
6	1.426	1.451	1.245	1.079	0.915	0.804	0.711	0.640
7	1.426	1.451	1.245	1.079	0.915	0.804	0.711	0.640
8	1.426	1.451	1.245	1.079	0.915	0.804	0.711	0.640
9	1.108	0.911	0.724	0.596	0.500	0.429	0.377	0.335
10	1.108	0.911	0.724	0.596	0.500	0.429	0.377	0.335
11	1.108	0.911	0.724	0.596	0.500	0.429	0.377	0.335
12	1.108	0.911	0.724	0.596	0.500	0.429	0.377	0.335
13	1.108	0.911	0.724	0.596	0.500	0.429	0.377	0.335
14	1.108	0.911	0.724	0.596	0.500	0.429	0.377	0.335
15	1.108	0.911	0.724	0.596	0.500	0.429	0.377	0.335
16	1.108	0.911	0.724	0.596	0.500	0.429	0.377	0.335

Table 4.9: FPS Matrix of URAM Version when Running MobileNetV2 with Improvement 3.11.2

B \ P_{PE}	8	16	24	32	40	48	56	64
1	197.9	383.6	486.4	640.1	653.8	695.3	698.0	739.6
2	395.7	767.1	972.7	1280.2	1307.6	1390.5	1395.9	1479.1
3	593.6	1150.7	1459.0	1920.2	1961.2	2085.7	2093.7	2218.5
4	791.5	1534.2	1945.4	2560.3	2615.0	2780.9	2791.7	2958.0
5	989.4	1917.7	2431.6	3200.1	3268.4	3475.8	3489.1	3696.8
6	1187.2	2301.3	2917.9	3840.2	3922.1	4170.9	4186.9	4436.2
7	1385.1	2684.8	3404.3	4480.2	4575.8	4866.1	4884.7	5175.6
8	1583.0	3068.3	3890.6	5120.2	5229.5	5561.2	5582.6	5914.9
9	1780.8	3451.7	4376.5	5759.4	5881.2	6251.7	6272.9	6646.6
10	1978.7	3835.3	4862.8	6399.4	6534.7	6946.3	6969.9	7385.1
11	2176.6	4218.8	5349.1	7039.3	7188.1	7641.0	7666.9	8123.6
12	2374.4	4602.3	5835.4	7679.2	7841.6	8335.6	8363.9	8862.1
13	2572.3	4985.8	6321.7	8319.2	8495.0	9030.2	9060.9	9600.6
14	2770.2	5369.4	6808.0	8959.1	9148.5	9724.9	9757.9	10339.1
15	2968.0	5752.9	7294.2	9599.0	9802.0	10419.5	10454.9	11077.6
16	3165.9	6136.4	7780.5	10239.0	10455.4	11114.1	11151.9	11816.1

Table 4.10: FPS per DSP Matrix of URAM Version when Running MobileNetV2 with Improvement 3.11.2

B \ P_{PE}	8	16	24	32	40	48	56	64
1	1.596	2.040	1.930	2.026	1.721	1.566	1.374	1.293
2	1.596	2.040	1.930	2.026	1.720	1.566	1.374	1.293
3	1.596	2.040	1.930	2.026	1.720	1.566	1.374	1.293
4	1.596	2.040	1.930	2.026	1.720	1.566	1.374	1.293
5	1.596	2.040	1.930	2.025	1.720	1.566	1.374	1.293
6	1.596	2.040	1.930	2.025	1.720	1.566	1.374	1.293
7	1.596	2.040	1.930	2.025	1.720	1.566	1.374	1.293
8	1.596	2.040	1.930	2.025	1.720	1.566	1.374	1.293
9	1.596	2.040	1.930	2.025	1.720	1.564	1.372	1.291
10	1.596	2.040	1.930	2.025	1.720	1.564	1.372	1.291
11	1.596	2.040	1.930	2.025	1.720	1.564	1.372	1.291
12	1.596	2.040	1.930	2.025	1.720	1.564	1.372	1.291
13	1.596	2.040	1.930	2.025	1.720	1.564	1.372	1.291
14	1.596	2.040	1.930	2.025	1.720	1.564	1.372	1.291
15	1.596	2.040	1.930	2.025	1.720	1.564	1.372	1.291
16	1.596	2.040	1.930	2.025	1.720	1.564	1.372	1.291

Table 4.11: FPS Matrix of SDRAM Version when Running MobileNetV2 with Both Improvements

B \ P_{PE}	8	16	24	32	40	48	56	64
1	197.9	383.6	486.3	640.1	653.8	695.3	697.9	739.4
2	395.1	744.4	936.4	1216.7	1241.4	1315.9	1320.7	1395.0
3	581.4	1082.9	1351.8	1662.3	1693.0	1784.9	1790.9	1881.4
4	760.8	1400.6	1736.0	2023.7	2057.8	2159.2	2165.7	2264.4
5	933.5	1655.0	2033.1	2310.9	2363.4	2466.7	2476.7	2554.6
6	1099.9	1870.2	2267.3	2535.6	2603.5	2691.0	2729.5	2777.9
7	1260.0	2036.7	2404.3	2651.4	2707.6	2788.4	2825.0	2875.8
8	1414.5	2182.3	2510.0	2728.8	2780.8	2855.2	2887.8	2927.5
9	1551.9	2310.9	2596.8	2792.3	2840.6	2908.6	2935.0	2963.2
10	1670.4	2413.7	2669.5	2845.2	2890.3	2951.1	2974.1	2992.3
11	1781.7	2498.8	2732.0	2890.0	2932.0	2985.3	3006.7	3016.4
12	1886.5	2567.5	2780.5	2925.3	2962.9	3008.3	3027.4	3033.5
13	1972.7	2618.2	2821.6	2952.4	2985.4	3022.1	3038.2	3045.4
14	2053.1	2663.2	2857.9	2975.3	3005.1	3032.7	3047.8	3055.7
15	2128.2	2703.5	2890.0	2995.2	3022.3	3041.9	3056.0	3062.5
16	2198.7	2739.8	2918.8	3011.9	3037.3	3050.0	3063.3	3068.2

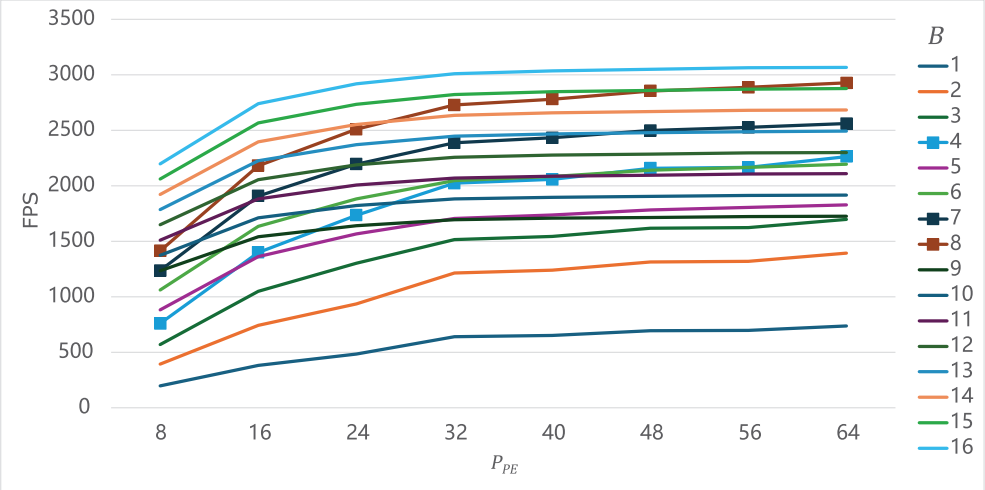


Figure 4.3: The trend in throughput variation as parallelism of PE increases.

The combined simulation results of the improvements from Section 3.11.1 and Section 3.11.2 are displayed in Tables. 4.11-4.14, showing that for both the SDRAM and URAM versions, throughput increases with rising B and

Table 4.12: FPS per DSP Matrix of SDRAM Version when Running MobileNetV2 with Both Improvements

B \ P_{PE}	8	16	24	32	40	48	56	64
1	1.596	2.040	1.930	2.026	1.721	1.566	1.374	1.293
2	1.593	1.980	1.858	1.925	1.633	1.482	1.300	1.219
3	1.563	1.920	1.788	1.753	1.485	1.340	1.175	1.096
4	1.534	1.862	1.722	1.601	1.354	1.216	1.066	0.990
5	1.506	1.761	1.614	1.463	1.244	1.111	0.975	0.893
6	1.478	1.658	1.500	1.337	1.142	1.010	0.895	0.809
7	1.452	1.548	1.363	1.199	1.018	0.897	0.794	0.718
8	1.426	1.451	1.245	1.079	0.915	0.804	0.711	0.640
9	1.391	1.366	1.145	0.982	0.831	0.728	0.642	0.576
10	1.347	1.284	1.059	0.900	0.761	0.665	0.585	0.523
11	1.306	1.208	0.986	0.831	0.701	0.611	0.538	0.479
12	1.268	1.138	0.919	0.771	0.650	0.565	0.497	0.442
13	1.224	1.071	0.861	0.719	0.604	0.524	0.460	0.410
14	1.183	1.012	0.810	0.673	0.565	0.488	0.429	0.382
15	1.144	0.959	0.765	0.632	0.530	0.457	0.401	0.357
16	1.108	0.911	0.724	0.596	0.500	0.429	0.377	0.335

Table 4.13: FPS Matrix of URAM Version when Running MobileNetV2 with Both Improvements

B \ P_{PE}	8	16	24	32	40	48	56	64
1	197.9	383.6	486.4	640.1	653.8	695.3	698.0	739.6
2	395.7	767.1	972.7	1280.2	1307.6	1390.5	1395.9	1479.1
3	593.6	1150.7	1459.0	1920.3	1961.3	2085.7	2093.8	2218.6
4	791.5	1534.2	1945.4	2560.3	2615.0	2780.9	2791.7	2958.0
5	989.4	1917.7	2431.7	3200.3	3268.7	3476.0	3489.3	3697.3
6	1187.2	2301.3	2918.0	3840.3	3922.3	4171.1	4187.1	4436.6
7	1385.1	2684.8	3404.2	4480.3	4575.9	4866.1	4884.8	5175.8
8	1583.0	3068.3	3890.6	5120.2	5229.5	5561.2	5582.6	5914.9
9	1780.8	3451.9	4376.8	5760.1	5883.0	6256.0	6280.0	6654.0
10	1978.7	3835.4	4863.1	6400.1	6536.5	6951.1	6977.6	7392.5
11	2176.6	4218.9	5349.3	7039.9	7190.0	7645.8	7674.8	8130.4
12	2374.4	4602.4	5835.6	7679.8	7843.5	8340.8	8370.9	8868.0
13	2572.3	4985.9	6321.8	8319.6	8496.9	9034.8	9066.3	9605.4
14	2770.2	5369.4	6808.1	8959.4	9150.3	9728.3	9761.8	10342.6
15	2968.0	5752.9	7294.2	9599.2	9803.7	10421.1	10457.0	11079.5
16	3165.9	6136.4	7780.5	10239.0	10455.4	11114.1	11151.9	11816.1

Table 4.14: FPS per DSP Matrix of URAM Version when Running MobileNetV2 with Both Improvements

B \ P_{PE}	8	16	24	32	40	48	56	64
1	1.596	2.040	1.930	2.026	1.721	1.566	1.374	1.293
2	1.596	2.040	1.930	2.026	1.720	1.566	1.374	1.293
3	1.596	2.040	1.930	2.026	1.720	1.566	1.374	1.293
4	1.596	2.040	1.930	2.026	1.720	1.566	1.374	1.293
5	1.596	2.040	1.930	2.026	1.720	1.566	1.374	1.293
6	1.596	2.040	1.930	2.025	1.720	1.566	1.374	1.293
7	1.596	2.040	1.930	2.025	1.720	1.566	1.374	1.293
8	1.596	2.040	1.930	2.025	1.720	1.566	1.374	1.293
9	1.596	2.040	1.930	2.025	1.720	1.566	1.374	1.293
10	1.596	2.040	1.930	2.025	1.720	1.566	1.374	1.292
11	1.596	2.040	1.930	2.025	1.720	1.565	1.373	1.292
12	1.596	2.040	1.930	2.025	1.720	1.565	1.373	1.292
13	1.596	2.040	1.930	2.025	1.720	1.565	1.373	1.292
14	1.596	2.040	1.930	2.025	1.720	1.565	1.373	1.292
15	1.596	2.040	1.930	2.025	1.720	1.565	1.372	1.291
16	1.596	2.040	1.930	2.025	1.720	1.564	1.372	1.291

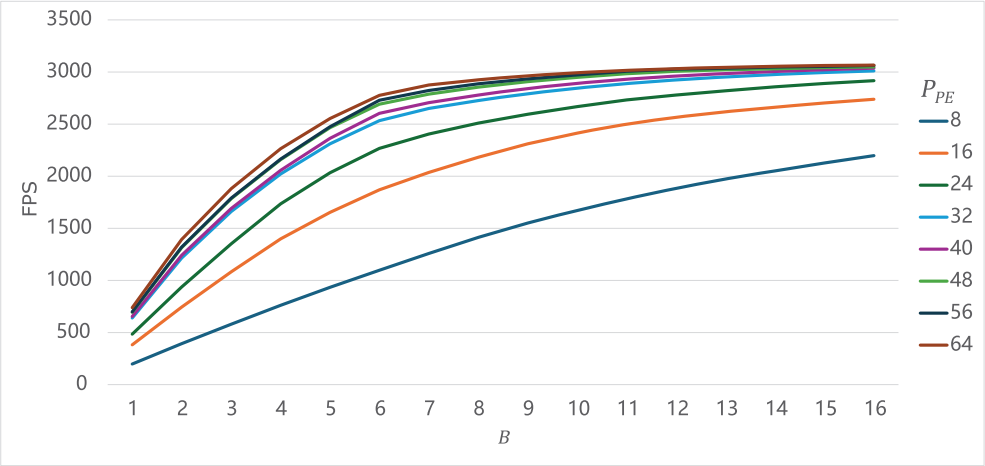
P_{PE}. For the SDRAM version, throughput growth slows as both values increase, while for the URAM version, it only slows as P_{PE} increases. Tables. 4.12 and 4.14 presents throughput per DSP, indicating area efficiency. For $P_{PE} = 16$ and $P_{PE} = 32$, maximum efficiency is observed when B is 4 or less. In contrast, $P_{PE} = 8$ is less efficient, as too many DSPs are dedicated to non-core operations such as quantization, reducing area efficiency.

Line graphs plotting FPS against parameters are shown in Figure. 4.4, similar to Figures. 4.2 and 4.3, shows no line crossover, proving that scalability has significantly improved after these improvements.

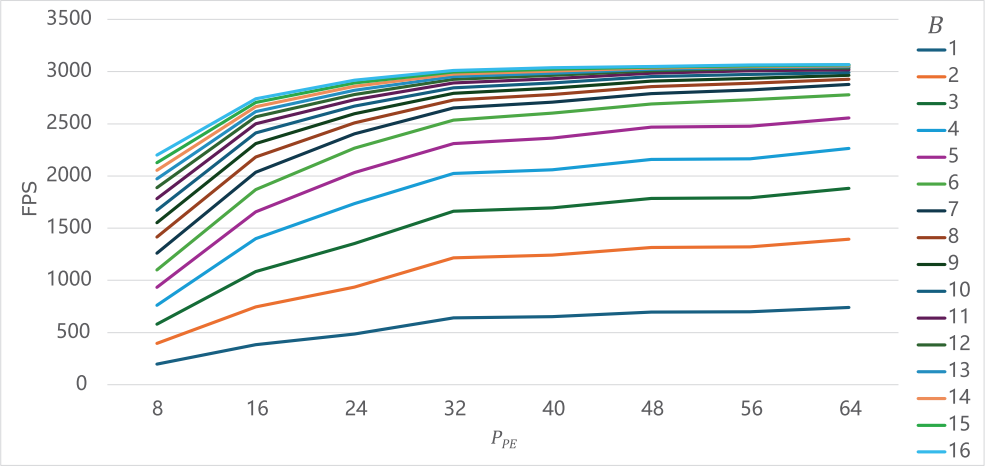
The results calculated with formulas from previous sections are pessimistic estimates. Compared to the simulated FPS matrix, the predicted average throughput is 98.1%, with a worst-case scenario of 105.1%, providing a fairly accurate estimate of theoretical performance.

4.3 Implementation

We implemented the accelerator using Vivado 2021.1 [46]. The implementation is divided into two parts: the first part involves deploying a small to



(a) The trend in throughput variation as batch size increases.



(b) The trend in throughput variation as parallelism of PE increases.

Figure 4.4: The trend of configuration with both improvement on batch size and PE parallelism.

svg

medium-scale accelerator suitable for edge computing on the ZYNQ UltraScale+ platform, without the later improvements discussed in Section 3.11. The second part verifies the effectiveness of the subsequent improvements, including implementing a large-scale accelerator suitable for data centers on the Virtex UltraScale+ platform and conducting extended experiments on certain improvements.

Similar to the previous section, subsections unrelated to improvements,

such as those detailing SDRAM1 on ZU3 and SDRAM4/URAM4 on ZU7, have been presented in [36]. On the other hand, subsections that highlight improvements, identified by “B” or “G” in their configuration names, have been published in [42]. The implementation described in [36] represents an earlier stage of development, where achieving the target frequency on the ZU7 platform was not yet stable. This dissertation adopts more optimized implementation strategies and refines additional critical paths, ensuring that the accelerator configuration achieves the target frequency consistently on the current device within a single die. As a result, the experimental results described in this disseration achieve slightly higher frequencies compared to those in [36] and [42].

4.3.1 Implementation on Edge Computing Device

The ZYNQ UltraScale+ FPGA has a hard memory controller located on the PS side, shared among all connected hard and soft cores for bandwidth. It also allows for deploying a soft memory controller in the PL side dedicated to user logic. The hard memory controller on the PS side and the external SDRAM it controls meet the needs of our accelerator without using additional logic resources, while also providing excellent random access performance. Therefore, we primarily utilize this hard controller.

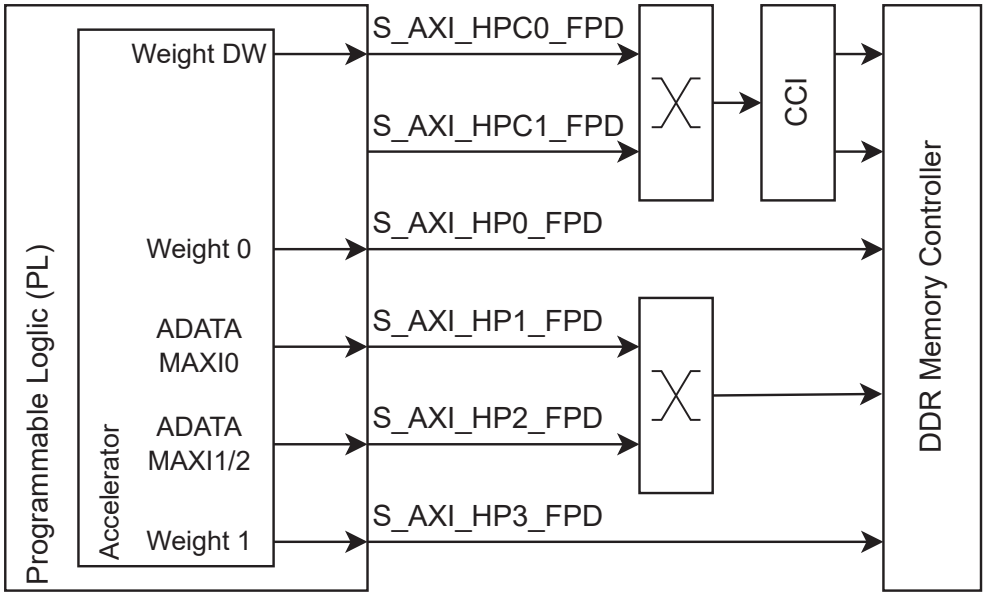


Figure 4.5: Port allocation on ZYNQ UltraScale+ platform.

The PS side of the ZYNQ UltraScale+ FPGA has multiple AXI slave ports, some of which share ports from the SDRAM controller. The sharing situation is shown in Figure. 4.5. To maximize performance, we avoid assigning two high-traffic master ports (such as the MAXI0 port for ADATA and the MAXI ports for Weight 0 and 1) to share an AXI slave port on the SDRAM controller. The MAXI ports of the Weight 0 and Weight 1 modules are connected to S_AXI_HP0_FPD and S_AXI_HP3_FPD, respectively, which do not share bandwidth with other PL ports. The MAXI port of the Weight DW module is connected to S_AXI_HPC0_FPD. ADATA's MAXI0 port is connected to S_AXI_HP1_FPD, while MAXI1 or MAXI2 connects to S_AXI_HP2_FPD. Although these two slave ports share bandwidth, the low traffic of MAXI1 and MAXI2 has little impact on overall throughput.

When our design is deployed on the edge computing platform, utilization may exceed the QoR (Quality of Results) recommendations. Therefore, we experimented with various strategy combinations to achieve a high clock frequency. Compared to previously published results, we have now identified the optimal strategy combination for our design, which enables more stable timing closure at $clk2x$ of 600 MHz with less effort. Thus, we updated our results to reflect this new strategy combination, while keeping the logic consistent.

The default synthesis strategy was used during logic synthesis. Based on our testing, the `Performance_ExplorerWithRemap` implementation strategy consistently yielded the best results. This is due to several factors: firstly, this strategy enables Post-Route Phys Opt Design, which allows additional optimization of critical paths after routing. Secondly, it activates `aggressive_remap` during the Opt Design phase before placement, compressing logic levels and shortening longer control logic paths, making it easier to meet timing requirements.

Our design also extensively uses hard blocks such as DSP, BRAM, and URAM, which have numerous input and output signals. Even with cascading, these hard blocks place significant pressure on routing resources, making routing congestion a critical issue. Proper placement has a substantial impact on the final results, and strategies with special placement methods occasionally yielded surprisingly good results.

Thus, base on the `Performance_ExplorerWithRemap` strategy, we employed additional placement directives. Two placement directives performed particularly well. The first, `EarlyBlockPlacement` (EBP), prioritizes the interconnections between hard blocks and places them accordingly, using these blocks as anchors around which general resources like LUTs and FFs are arranged. The DSP and BRAM used in the computation paths of PE Arrays 0/1 operate at higher frequencies, requiring stricter placement and routing distances. The lower-frequency LUT multipliers of the PEDW Array have more relaxed distance requirements, so prioritizing DSP and BRAM with higher frequencies often yields better placement and routing results. The second directive, `ExtraNetDelay_high` (ENDh), increases the estimated delay between long-distance components (a more pessimistic approach), encouraging closer placement to reduce critical path delays.

The implementation results are shown in Table. 4.15. The target frequency of $clk2x$ is 600 MHz, and all three configurations met the target frequency. Additionally, the operating frequency can be changed via the AXI slave port of the MMCM-related control module.

In the ZU3, there are relatively few DSPs occupying routing resources. Therefore, even with the majority of DSPs and BRAM utilized and with LUT usage reaching 55% (including 27% LUTRAM usage), it is still relatively easy for both directives to achieve the target frequency The results of the EBP directive are listed in the table. The ZU7SDRAM4 and ZU7URAM4 configurations use a very similar amount of logic and arithmetic resources

Table 4.15: Implementation Result on ZYNQ UltraScale+ Platform

Device		ZU7	ZU7	ZU3
Configuration		URAM4	SDRAM4	SDRAM1
Directive		ENDh	ENDh	both, EBP
Overall	LUT	122781	122306	38524
	LUTRAM	13181	13774	7898
	FF	195067	196880	61748
	BRAM	248	248	158
	URAM	64	0	-
	DSP	1264	1264	316
	Power(W)	20.729	17.466	7.083
Accelerator	LUT	120299	119759	36985
	LUTRAM	13043	13636	7760
	FF	190910	192759	57303
	BRAM	248	248	158
	URAM	64	0	-
	DSP	1264	1264	316
	Power(W)	17.01	13.773	3.887
$clk2x$(MHz)		600	600	600
Sim. CC	URAM	507840	-	-
	SDRAM	-	671156	508333
Act. CC	URAM	510589	-	-
	SDRAM	-	674097	511976
Act. FPS	URAM	2350.227	-	-
	SDRAM	-	1780.159	585.965
Act. Band (GiB/s)	URAM	2.915858	-	-
	SDRAM	-	7.128928	3.867033

but show noticeable differences in memory resources. In both configurations, BRAM is used to store weights, filters' data, and small capacity row data of SWU, resulting in identical BRAM usage. However, ZU7URAM4 utilizes an additional 64 URAMs as inter-block buffers, whereas ZU7SDRAM4 does not use any URAM. Both configurations achieve the target frequency, although they require more effort compared to ZU3SDRAM1. Despite using additional URAM in ZU7URAM4, these URAMs are cascaded in groups of eight, which maximizes the use of dedicated cascade routing resources, allowing the target frequency to be achieved.

We validated our accelerator on the ZYNQ UltraScale+ MPSoC ZCU104 evaluation board (ZU7) and the ALINX Baidu EdgeBoard FZ3B (ZU3).

When running our accelerator on the ZCU104 at the maximum $clk2x$ frequency of 600 MHz, the power management chip (PMIC) triggers over-current protection (the current reported by Vivado is far below the maximum sustained current value for a 10-year lifespan on page 389 of UG583 v1.28), cutting off the power supply. As the ZCU104 is an evaluation kit, it has a lower current limit than the silicon's maximum allowable current to minimize the risk of damage due to improperly designed logic.

Therefore, for evaluations on the ZCU104, we first reduced the $clk2x$ frequency to 500 MHz and the clk1x frequency to 250 MHz to verify that our accelerator operates correctly on the actual FPGA development board. We then restored the frequencies to the maximum level and set all weights and input images to zero. This approach halts the operation of DSPs and LUT-based multipliers, reducing switching power, while keeping the timing and quantity of data transfers unchanged. This allows testing at the highest frequency, where the accelerator demands more bandwidth and runs more clock cycles. The actual runtime on the FPGA closely matched the simulated runtime.

The actual runtime is shown in Tables. 4.2 and 4.15. Due to the limited weight memory capacity, the accelerator frequently pauses to wait for initialization to complete. Accessing the actual off-chip SDRAM introduces additional latency, and the need for self-refresh further extends initialization time, resulting in slightly lower performance than the simulated results. Runtime analysis of the AXI bus waveform reveals that during periods of high bus activity, the valid signal on the R channel of the AXI slave port of the Zynq PS periodically de-asserts for brief intervals, despite numerous pending read tasks. This behavior suggests delays in providing read data, likely

caused by factors such as SDRAM row switching or self-refresh operations. Nevertheless, thanks to the efficient design of the hard memory controller in the PS, the discrepancy between simulation and real-world results remains minimal.

4.3.2 Implementation on High-end Device

High-end FPGAs like Virtex have very large hardware resource capacities. To control costs and improve yield, these high-end FPGAs are typically made from multiple dies connected using Stacked Silicon Interconnect (SSI) technology, with each die referred to as a Super Logic Region (SLR). These SLRs are placed on a large silicon interposer manufactured with a more mature process that offers higher yield. The SLRs are interconnected through this silicon interposer, achieving high bandwidth and low-latency inter-die connections. However, compared to intra-die connections, the inter-die connections still have limited routing resources and higher latency. Dataflow architecture accelerators can map modules from different layers to multiple SLRs, with limited interconnect width between modules. The limited physical connections between SLRs will not significantly impact performance.

However, our accelerator resembles an Overlay architecture, where modules are tightly coupled with denser interconnects. Placing an instance of the accelerator across multiple SLRs may significantly reduce the frequency. Therefore, it is preferable to place an instance of the accelerator within a single SLR. Currently, the highest DSP count on a single die in AMD FPGAs is 4,272 in the ZYNQ UltraScale+ RFSoC series [47] and 3,984 in the Versal Prime/Premium series [48], which makes it impossible to implement the configuration shown in the lower right corner of Table. 4.12 (highlighted in orange to red). Thus, these results cannot be fully verified on an actual FPGA.

We used the XCVU13P-FHGB2104-2 (VU13P) accelerator card to verify these configurations, with its hardware resources shown in Table. 4.1. This accelerator card consists of four interconnected dies with identical resource capacities using SSI technology, giving it four SLRs. Additionally, it is equipped with four channels of 72-bit 2,400 MT/s DDR4 ECC SDRAM, each connected to the IO pad of a separate SLR. Each SLR on the VU13P has up to 3,072 DSPs, more than most devices, with hardware resources

ratio (including BRAM, URAM, and LUTs) closely matching the needs of our accelerator. Since it's not feasible to maximize all configurations like in the simulated evaluations in Section 4.2.4, the actual throughput of the implemented accelerator is slightly lower.

Based on previous simulation results, we determined the most suitable configuration: $P_{PE} = 32$, $B = 8$, with a depth of 2,048 for Weight 0/1. This configuration is similar to the previous ZYNQ platform implementation, except for batch size. We marked configurations that used batch size improvements from Section 3.11.1 with a mark "B" and those using P_{PE} improvements from Section 3.11.2 with a mark "G" (Group). Simulations showed that, with improvements from Section 3.11.1 and Section 3.11.2, SDRAM8BG requires 944,941 clock cycles on $clk1x$, while URAM8BG only needs 502,972 cycles. Without any improvements, SDRAM8 requires 1,076,982 cycles and URAM8 requires 507,841 cycles. The small difference is because the batch size improvement in Section 3.11.1 does not apply when B is a power of 2.

Unlike the previous accelerator configurations for edge computing platforms that required separate implementations for URAM and SDRAM versions, the configuration on this accelerator card is much larger, and there are many platform components, making it challenging to estimate the time required for separate synthesis, placement, and routing. It was shown that the two versions mainly differ in URAM usage, so we integrated both versions such that the URAM version's physical implementation is logically compatible with the SDRAM version's instructions. Thus, only the URAM8BG physical configuration is implemented, and with corresponding instructions, both URAM8BG and SDRAM8BG can be evaluated.

As shown in the floorplan in Figure. 4.6, platform components share SLR1, while the accelerator and its connected AXI interconnect module fully utilize SLR2. In this layout, the purple area represents the accelerator, yellow represents the AXI bus interconnect, green represents the DDR4 SDRAM controller MIG, and red represents XDMA. The results show that our accelerator fully utilized the hardware resources of SLR2. The resource usage statistics for the accelerator alone are: 55% LUT, 11% LUTRAM, 43% FF, 55% BRAM, 40% URAM, and 82% DSP. Detailed results are shown in Table. 4.16.

After placement and routing, $clk2x$ achieved the target frequency of 600 MHz. Even though our accelerator and platform interconnect components

Table 4.16: Implement Result of URAM8BG on VU13P

Device		VU13P
Configuration		URAM8BG
Directive		EBP
Overall	LUT	345200
	LUTRAM	38450
	FF	521943
	BRAM	531.5
	URAM	128
	DSP	2531
	Power(W)	53.262
Accelerator	LUT	235625
	LUTRAM	21359
	FF	374870
	BRAM	368
	URAM	128
	DSP	2528
	Power(W)	36.876
$clk2x$(MHz)		600
Sim. CC	URAM	502972
	SDRAM	944942
Act. CC	URAM	533069
	SDRAM	1071667
Act. FPS	URAM	4502.231
	SDRAM	2239.502
Act. Band (GiB/s)	URAM	3.638745
	SDRAM	6.901092

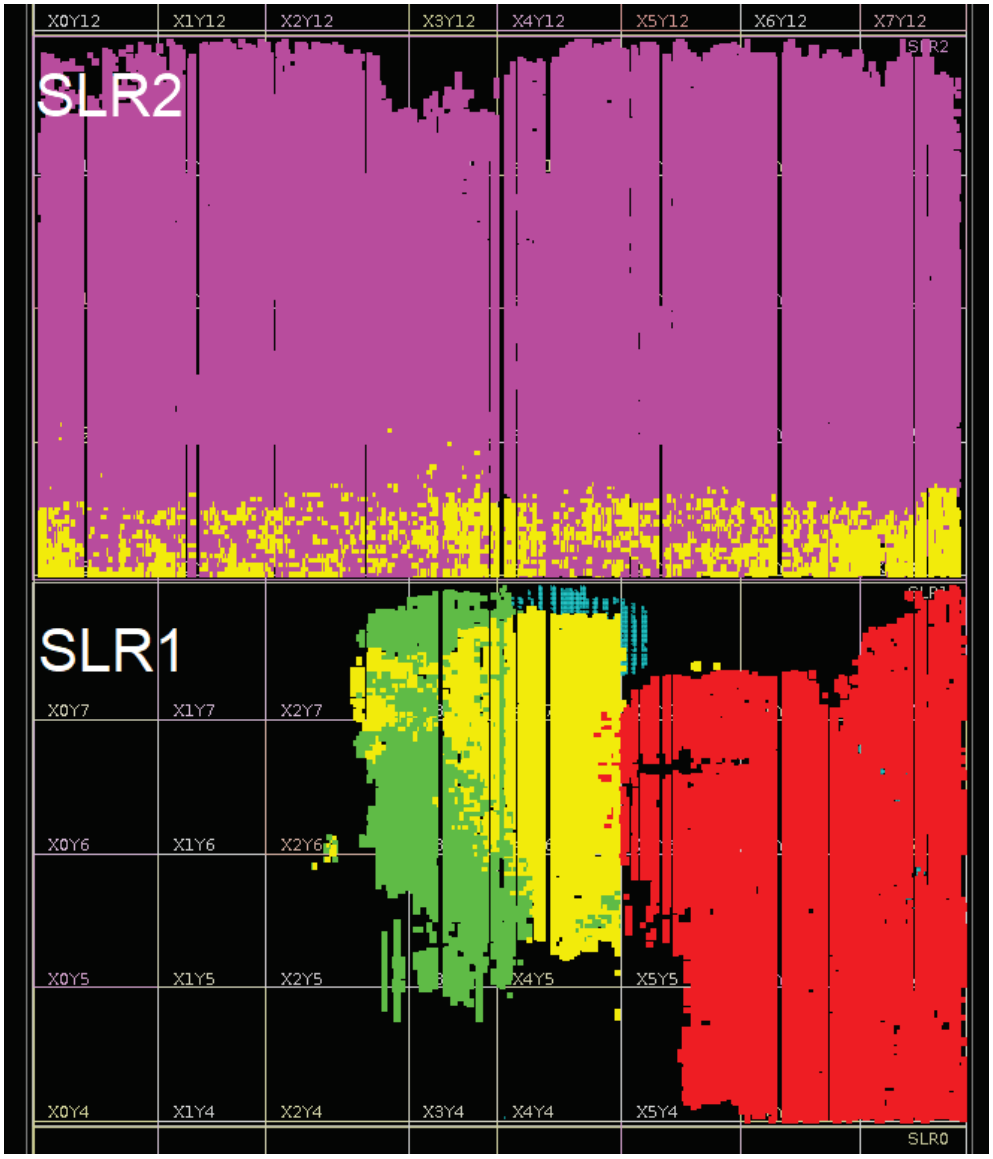


Figure 4.6: The floorplan for single instance occupying an SLR.

heavily occupied SLR2's hardware resources, the target frequency was still met, demonstrating that our accelerator can maintain high operating frequencies even when using a large area with high parallelism.

Actual testing showed that SDRAM8BG required 1,071,667 clock cycles, whereas the simulation required only 944,942 cycles, achieving only 88.2% of the simulated performance, with a throughput of 2,239.5 FPS, approximately

82.1% of the theoretical maximum. The required off-chip memory bandwidth was 6.90 GiB/s. For URAM8BG, the actual clock cycles were 533,069, achieving 94.4% of the simulated performance, with a throughput of 4,502.2 FPS, approximately 88.0% of the theoretical maximum, with an off-chip memory bandwidth of 3.64 GiB/s.

The difference between simulation and actual results is mainly due to limitations in the efficiency of the off-chip memory controller MIG, and the VU13P board's DDR4 SDRAM x16 chips (native x16, not twin-x8 which means packaged x8 chips) have only 8 banks, half the number of x8 chips, resulting in lower random access performance. [49] Additionally, due to the limited capacity of the weight memory in the tests, compared to the maximum throughput achieved with maximum capacity weight memory, there were additional pauses, reducing throughput slightly. However, a higher ratio was still achieved, though increasing the weight memory capacity could lead to a large increase in area with limited potential for performance improvement.

With $B = 8$, resource usage is very high, making it challenging to verify each improvement individually. Furthermore, the improvements in Section 3.11.1 mainly target cases where B is not a power of 2, so $B = 8$ does not demonstrate their effects. Therefore, we focused on the case of $B = 6$. The floorplan used is the same as for $B = 8$. Regardless of whether the improvements were applied, the resource utilization results were similar, with the accelerator occupying approximately 41% of SLR2's hardware resources for LUT, 9% for LUTRAM, 33% for FF, 46% for BRAM, 30% for URAM, and 62% for DSP.

Table. 4.17 compares simulation results with actual results, demonstrating that the throughput relative to the theoretical maximum remains relatively consistent in both simulated and actual measurements, confirming the effectiveness of these improvements on real hardware. Compared to URAM6, URAM6B had less impact on throughput but significantly reduced bandwidth usage, bringing the usage rate for the same throughput down to 88.4% of the original. URAM6G slightly improved throughput. For both improvements enabled URAM6BG, the simulation result reached 93.2% of the theoretical maximum, while the actual result was 87.9%. On the other hand, for SDRAM8BG, the simulation result reached 92.9% of the theoretical maximum, while the actual result was 83.5%. The off-chip memory bandwidth usage of SDRAM6BG reached 35.7%.

Table 4.17: Implement Result of URAM8 under Different Conditions on VU13P

Device		VU13P			
Configuration Directive		URAM6 both, EBP	URAM6B both, EBP	URAM6G both, EBP	URAM6BG both, EBP
Overall	LUT	288495	288755	289024	289437
	LUTRAM	34599	34600	34600	34596
	FF	431593	431530	431721	431572
	BRAM	471.5	471.5	471.5	471.5
	URAM	96	96	96	96
	DSP	1899	1899	1899	1899
	Power(W)	44.697	42.77	45.023	44.605
Accelerator	LUT	177936	178253	178113	178555
	LUTRAM	17509	17509	17509	17507
	FF	284470	284326	284583	284461
	BRAM	308	308	308	308
	URAM	96	96	96	96
	DSP	1896	1896	1896	1896
	Power(W)	28.683	26.917	29.061	28.689
$clk2x$(MHz)		600	600	600	600
Sim. CC	URAM	507841	507824	502971	502954
	SDRAM	1076982	860536	944942	764169
Act. CC	URAM	538299	537412	532965	532987
	SDRAM	1215508	957016	1071649	849932
Act. FPS	URAM	3343.867	3349.386	3377.332	3377.193
	SDRAM	1480.862	1880.846	1679.654	2117.816
Act. Band (GiB/s)	URAM	3.603391	3.189831	3.639455	3.216314
	SDRAM	7.05324	6.989882	6.901208	6.831407

When the depth for the Weight 0/1 was increased to the device's maximum allowable value of 4,096, the actual measurement for SDRAM6BG reached 84.2% of the theoretical maximum, and URAM6BG reached 95.2%. To further enhance the MIG's efficiency, the burst transfer length was doubled to 32, which increased SDRAM6 throughput to 90.2%. Reducing the DDR4 memory width led to a decrease in throughput but significantly increased bandwidth usage. When the width was reduced to 8 bits, the off-chip memory bandwidth usage reached 83.8%. This behavior resembles the improvement achieved by increasing burst transfer length to improve MIG efficiency, as it increases the burst transfers for a single memory access command, reduces row switching, and lowers randomness. Additionally, we experimented with modifying the priority arbitration strategy for AXI SmartConnect and AXI Crossbar, but no significant improvement was observed.

The board-level evaluation effectively assessed the actual performance of the accelerator with improvements, even when substantially increasing area usage, showing its capability to maintain a high frequency despite severe congestion. Individual assessments in various scenarios confirmed the effectiveness of these improvements on the actual evaluation board.

4.3.3 Fully Utilizing VU13P

We attempted to achieve maximum throughput on the VU13P device. As previously mentioned, placing components across different SLRs can significantly reduce the maximum frequency. Additionally, as the user I/O for each SLR is centrally located, MIG and AXI interconnect components must be centrally placed. The tight interconnections between two PE Arrays in our accelerator would cause severe congestion if routed through areas occupied by MIG and other modules.

With this in mind, we split the $B = 6$ configuration into two parts, each with $B = 3$, placing them on opposite sides of an SLR, as shown in Fig. 4.7. The VU13P has four SLRs, giving eight slots. However, XDMA occupies a large area, making it difficult to place a $B = 3$ configuration accelerator in that slot, so we abandoned the XDMA slot, placing one instance in each of the remaining seven slots. MIG in SLR3, SLR2, and SLR0 is shared by two instances, while MIG in SLR1 is dedicated to instance 7.

After synthesis and placement and routing, our accelerator utilized 37% of the device's LUT, 11% of LUTRAM, 28% of FF, 57% of BRAM, 26% of

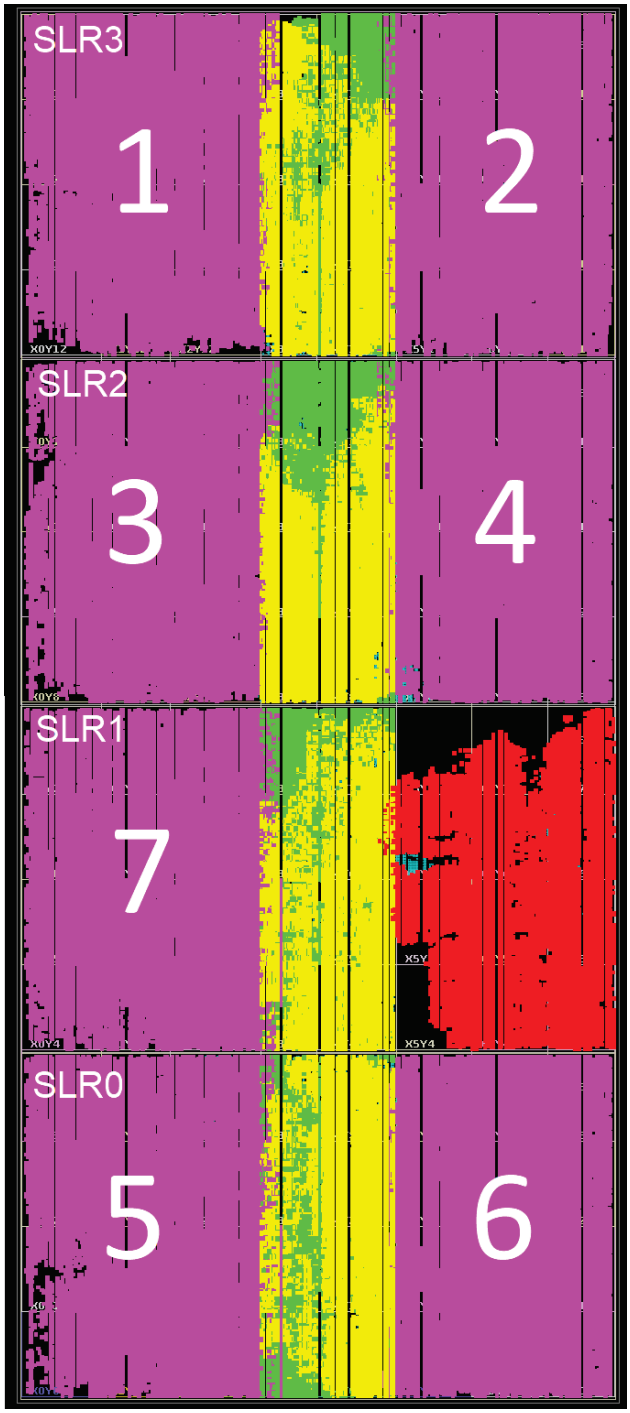


Figure 4.7: The floorplan for multiple instances.

URAM, and 55% of DSP. Including platform-related components, the total usage was 55% of LUT, 20% of LUTRAM, 41% of FF, and 68% of BRAM, with URAM not used by the platform, and minimal DSP usage, keeping the percentage unchanged. Specific information is shown in Table. 4.18.

Due to the smaller size of each instance, $clk2x$ easily achieved the target frequency of 600 MHz. When running these instances individually, the throughput per instance reached 1,424.3 FPS for SDRAM3BG and 1,687.8 FPS for URAM3BG, reaching 91.6% and 94.3% of the simulation results, respectively. Compared to the theoretical maximum, this achieved 85.7% and 87.9%.

However, when all instances started simultaneously to simulate actual load, resource contention emerged between instances sharing the MIG, leading to a significant throughput reduction compared to individual execution. We optimized the timing of the user channel writes to the bar in XDMA to be as close as possible. Since PCIe drivers and WinAPI require reinitializing the write pointer after each address write, this takes longer. Writing the start command to all seven instances took a total of 0.023 ms, averaging about 1,000 clock cycles at $clk1x$ per instance, which is negligible compared to the runtime of at least 500,000 clock cycles, approximating simultaneous execution.

For SDRAM3BG, when instances 1-6 started simultaneously, their throughput dropped to 785.5 FPS, about half of the original value. This highlights that when eight ports sequentially access MIG, the sequential access appears highly random from the MIG's perspective, significantly reducing efficiency. The throughput of instance 7 remained unchanged, with a total throughput of 6,137.5 FPS for all seven instances, only 50.5% of the simulation result. For URAM3BG, when instances 1-6 started simultaneously, their throughput only slightly dropped to 1,503.9 FPS, showing minimal change. This demonstrates the critical importance of using URAM as an inter-block buffer under complex conditions. The total throughput of the seven instances reached 10,710.9 FPS, with a smaller drop from individual execution, achieving 84.0% of the simulation result.

In this section, we explored the computational potential of VU13P, successfully achieving a throughput of over 10,000 FPS for running MobileNetV2 on a single chip. To our best knowledge, the only current FPGA accelerator reaching this level is AMD's Alveo V70, running DPU C20B14CU1 [50] and achieving 11,544.90 FPS. However, it primarily relies on the AI Engine (2-gen

Table 4.18: Implementation Result of 7 URAM3BG Instances on VU13P

Device		VU13P Individual	VU13P Simultaneous Inst. 7	Inst. 1-6	Sum
Configuration		URAM3BGx7	URAM3BG	URAM3BG	URAM3BGx7
Directive		EBP			
Overall	LUT	950671			
	LUTRAM	154892			
	FF	1412933			
	BRAM	1833.5			
	URAM	336			
	DSP	6816			
	Power(W)	124.081			
Accelerator	LUT	646334			
	LUTRAM	86394			
	FF	974720			
	BRAM	1526			
	URAM	336			
	DSP	6804			
	Power(W)	85.065			
$clk2x$(MHz)		600			
Sim. CC	URAM	502929			
	SDRAM	578910			
Act. CC	URAM	533240	533096	598497	
	SDRAM	631880	631886	1145725	
Act. FPS	URAM	11814.56	1688.25	1503.766	10710.85
	SDRAM	9970.242	1424.307	785.5286	6137.479
Act. Band (GiB/s)	URAM	12.22496	2.581298	2.299226	16.37666
	SDRAM	32.98214	5.415642	2.986815	23.33653

AIE-ML tiles) unique to the Versal series, which has a significant advantage over our DSP-based accelerator.

4.3.4 Implementation for Improved Performance

In Section 3.11.4, we introduced an improvement aimed at addressing the severe throughput drop observed when two accelerator instances sharing the same MIG run SDRAM3BG simultaneously, as noted in the previous section. The new physical implementation configuration is named URAM3x2BG, where "3" represents the batch size, and "x2" indicates that two sub-instances with a batch size of 3 share the ports to reduce memory access randomness and volume. Since this configuration aims to minimize the routing impact caused by the MIG and AXI interconnect modules placed centrally in the SLR, we need to deploy independent weight memories on both sides for these two sub-instances. Thus, the hardware resource usage in this configuration is close to the total usage of two URAM3BGs. The operational instructions for URAM3x2BG are identical to those of configurations without instance port sharing, and its input-output data format is fully compatible with URAM6BG.

Even with minimized cross-region connections, leaving only essential control signals and weight initialization buses, achieving the target frequency of 600 MHz for $clk2x$ remains challenging. To meet this frequency goal, we imposed several layout constraints to ensure that the placement aligns as closely as possible with our design. First, we ensured that shared modules between the two instances are centrally located within the SLR, so the signal fanout to corresponding modules in both instances has equal-length paths. With the increased number of modules in the central region, and additional signal fanouts to both sides, congestion in this area became even more pronounced.

Originally, we made no modifications to the XDMA's AXI bus, maintaining a maximum width of 512 bits to fan out across the four MIGs in the SLRs. However, XDMA's bandwidth could not fully utilize a four-lane 512-bit AXI bus (up to a single 512-bit lane or four 128-bit lanes). Due to the increased congestion in the central region, we adjusted the bus width to a more suitable value. According to our simulation, the maximum achievable throughput in this configuration aligns with twice the values for URAM3BG and SDRAM3BG, reaching 3,579.0 FPS. Writing image data to SDRAM

managed by the MIG requires a bandwidth of 1.34 GiB/s. Given a 250 MHz frequency for the SLR interconnect network, a 64-bit AXI interconnect network between SLRs meets the bandwidth demand. We developed custom modules to ensure the AXI interconnect correctly handles the bit-width conversion.

Additionally, we constrained the placement of PE Arrays 0 and 1 within each instance. Through repeated attempts, the automatic placer failed to achieve optimal solutions. The most common outcome was arranging PE Arrays 0 and 1 horizontally in the left space for one instance, while placing them vertically on the right for the other instance, resulting in excessive routing distances for signals from the central shared modules to the leftmost PE Array. Another suboptimal outcome was placing both left and right arrays vertically but inconsistently, with PE Array 0 positioned on the upper left and on the lower right, leading to cross-routing and longer routing distances. Thus, we opted for manual layout. In the floorplan shown in Figure. 4.8, the central horizontal region of the SLR has a clear gap, and each SLR is split into upper and lower sections. The upper section houses PE Array 0, and the lower section PE Array 1, with the initialization modules centrally located, ensuring equal-length fanout in the horizontal direction. For the SLR occupied by XDMA, we used only half of it to house the physical configuration for the URAM3x1BG accelerator.

Ultimately, we achieved the $clk2x$ target frequency of 600 MHz. The hardware resources used by the accelerator account for 37% of LUTs, 10% of LUTRAM, 30% of FFs, 57% of BRAM, 26% of URAM, and 54% of DSPs. Including platform components, resource utilization for the entire device reached 51% of LUTs, 16% of LUTRAM, 39% of FFs, and 69% of BRAM, with URAM and DSP percentages unchanged. Detailed information is provided in Table. 4.19. Compared to URAM6BG implemented in Section 4.3.2, URAM3x2BG's resource usage is similar, except for slightly higher BRAM and LUTRAM usage due to the addition of identical weight memory on both sides of the SLR.

In terms of throughput, URAM3x2BG achieves a significant improvement, reaching 3,377.5 FPS, while SDRAM3x2BG achieves 2,738.1 FPS. This brings the total throughput to 11,821.5 FPS for the URAM configuration and 9,642.0 FPS for the SDRAM configuration, representing notable gains compared to previous results, especially for the SDRAM version, which achieves approximately 1.6 times the prior performance.

Table 4.19: Implementation Result of 3 URAM3x2BG Instances and 1 URAM3x1BG Instance on VU13P

Device		VU13P		
Configuration		URAM3x2BGx3+ URAM3x1BGx1	URAM3x2BG	URAM3x1BG
Directive		EBP		
Overall	LUT	885870		
	LUTRAM	129119		
	FF	1360008		
	BRAM	1847		
	URAM	336		
	DSP	6648		
	Power(W)	144.55		
Accelerator	LUT	647642	184786	93285
	LUTRAM	79741	22693	11662
	FF	1028063	293295	148177
	BRAM	1526	436	218
	URAM	336	96	48
	DSP	6636	1896	948
	Power(W)	104.581	29.782	15.234
$clk2x$(MHz)		600		
Sim. CC	URAM		502929	502929
	SDRAM		578910	578910
Act. CC	URAM		532940	532840
	SDRAM		657390	630384
Act. FPS	URAM	11821.54	3377.491	1689.062
	SDRAM	9642.003	2738.101	1427.701
Act. Band (GiB/s)	URAM	12.23233	3.216597	2.58254
	SDRAM	31.92529	8.832248	5.428548

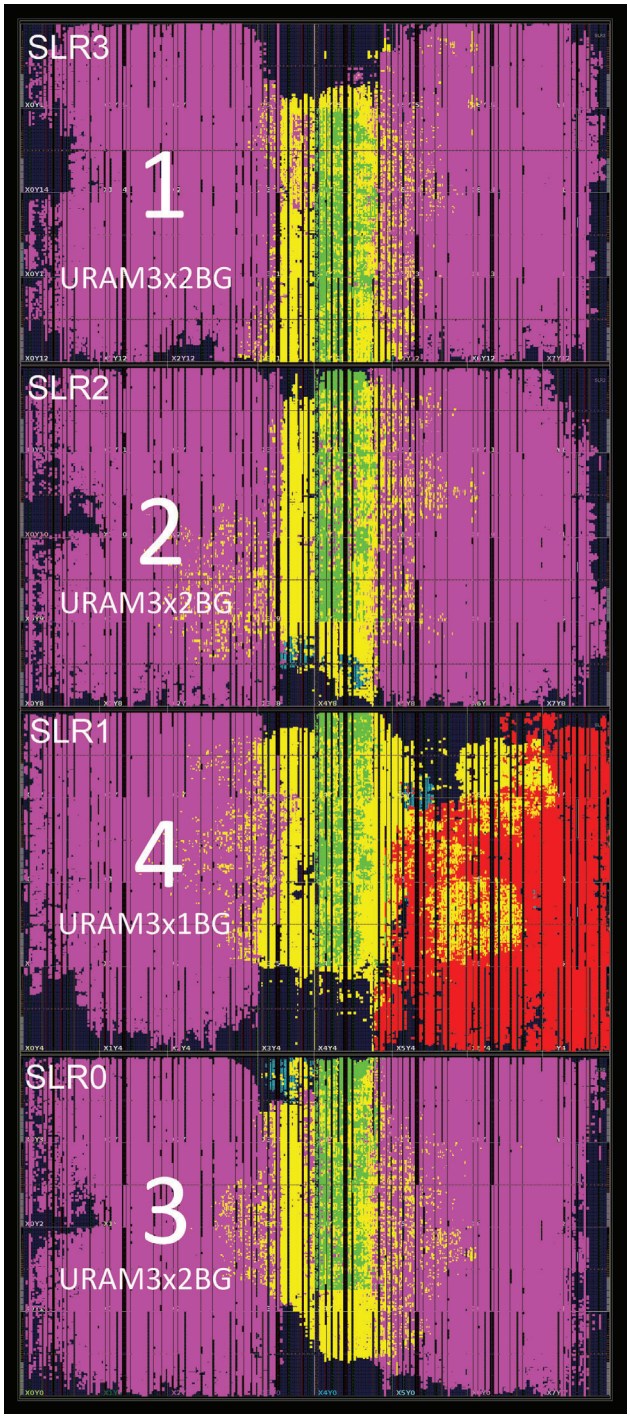


Figure 4.8: The floorplan for multiple instances.

4.3.5 Utilizing Improvements for Previous Implementations

4.3.5.1 URAM4BG, SDRAM1BG and URAM2x2BG on ZYNQ

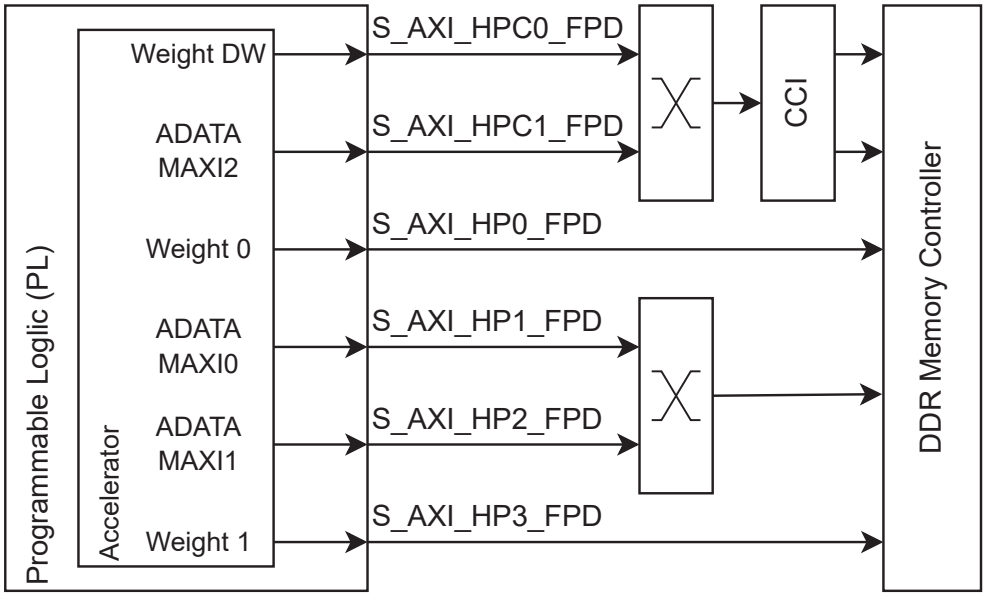


Figure 4.9: Port allocation on ZYNQ UltraScale+ platform for improved design.

We applied these subsequent improvements to the ZU3, ZU7. For the ZU3, we used the SDRAM1BG configuration; for the ZU7, we implemented URAM4BG. The new AXI port allocation is shown in Figure. 4.9. The results are listed in Table. 4.20, showing a slight performance increase in the new implementations.

As previously mentioned, the maximum AXI port width on the PS side of the ZYNQ UltraScale+ platform is 128 bits, and increasing the port width beyond this limit may not yield additional benefits. We are also exploring ways to surpass this limit. A reasonable solution is to use two ports for ping-pong operations, alternately transferring higher-width data bursts across the two ports. These data are first processed through a FIFO to ensure that the high-width side can transmit or receive data at full speed before converting to a lower width that connects to the PS side's AXI port.

Table 4.20: Implementation Result with Improvements on ZYNQ UltraSCALE+ Platform

Device		ZU7	ZU3
Configuration		URAM4BG	SDRAM1BG
Directive		EBP	both, EBP
Overall	LUT	124708	38500
	LUTRAM	13746	7901
	FF	198410	62452
	BRAM	248	158
	URAM	64	-
	DSP	1264	316
	Power(W)	21.627	7.418
Accelerator	LUT	122085	35945
	LUTRAM	13608	7763
	FF	193665	57945
	BRAM	248	158
	URAM	64	-
	DSP	1264	316
	Power(W)	18.294	4.683
$clk2x$(MHz)		600	600
Sim. CC	URAM	502937	-
	SDRAM	637501	502888
Act. CC	URAM	504938	-
	SDRAM	640293	506430
Act. FPS	URAM	2376.529	-
	SDRAM	1874.142	592.382
Act. Band (GiB/s)	URAM	2.94849	-
	SDRAM	6.58572	3.61872

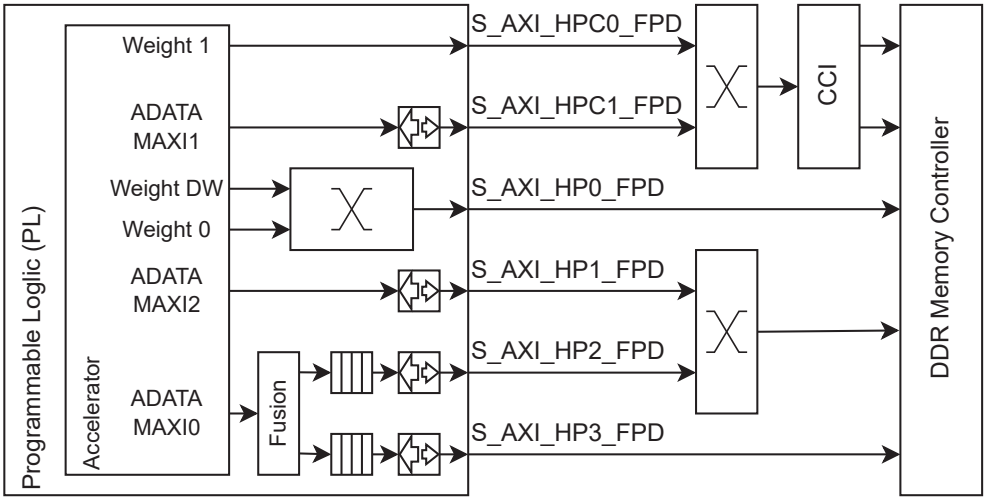


Figure 4.10: The Fusion of two AXI slave ports on ZYNQ UltraScale+ platform.

The specific connections, as shown in Figure. 4.10, are optimized per port load to ensure that high-bandwidth-demanding ports receive sufficient bandwidth. This approach allows us to implement the URAM2x2BG configuration on the ZU7. Since the aim is to increase port width for higher throughput, both instances continue sharing the weight memory despite being split. The implementation results are shown in Table. 4.21, and compared with Tables. 4.15 and 4.20, hardware resource usage between URAM4, URAM4BG, and URAM2x2BG remains similar. However, the URAM2x2BG configuration significantly enhances SDRAM mode throughput, further closing the gap with URAM mode.

4.3.5.2 URAM2x4BG and URAM4x2BG on VU13P

Additionally, we split the URAM8BG configuration on the VU13P into URAM2x4BG and URAM4x2BG with all other conditions unchanged. Since the allocated layout area is a single SLR, there are no routing constraints between instances, so all four instances share the weight memory. Since URAM8BG already achieved the target frequency, its split versions also easily reached this goal. The results, shown in Table. 4.22, reveal that while the split implementations' area remains nearly identical to URAM8BG, throughput saw a substantial boost, especially with SDRAM-based instructions.

Table 4.21: Implementation Result of a URAM2x2BG Instances on ZU7

Device		ZU7
Configuration		URAM2x2BG
Directive		EBP
Overall	LUT	130220
	LUTRAM	14975
	FF	206765
	BRAM	265
	URAM	64
	DSP	1264
	Power(W)	19.338
Accelerator	LUT	121478
	LUTRAM	13566
	FF	192918
	BRAM	248
	URAM	64
	DSP	1264
	Power(W)	15.728
$clk2x$(MHz)		600
Sim. CC	URAM	502920
	SDRAM	527504
Act. CC	URAM	505190
	SDRAM	561004
Act. FPS	URAM	2375.344
	SDRAM	2139.022
Act. Band (GiB/s)	URAM	2.94702
	SDRAM	7.516506

Table 4.22: Implement Result of URAM2x4BG on VU13P

Device		VU13P	
Configuration		URAM2x4BG	URAM4x2BG
Directive		ENDh	EBP
Overall	LUT	348869	348302
	LUTRAM	39950	38941
	FF	526686	523293
	BRAM	531.5	531.5
	URAM	128	128
	DSP	2531	2531
	Power(W)	47.841	50.866
Accelerator	LUT	234920	235798
	LUTRAM	21275	21319
	FF	373193	373956
	BRAM	368	368
	URAM	128	128
	DSP	2528	2528
	Power(W)	35.18	34.369
$clk2x$(MHz)		600	600
Sim. CC	URAM	502920	502937
	SDRAM	527520	637501
Act. CC	URAM	532829	532828
	SDRAM	696823	751356
Act. FPS	URAM	4504.259	4504.268
	SDRAM	3444.203	3194.225
Act. Band (GiB/s)	URAM	3.640384	3.640391
	SDRAM	10.61342	9.843101

However, compared to SDRAM2x4's simulation result, the achieved throughput is significantly lower, reaching only 75.7%. This is because, despite MIG providing a 512-bit, 300 MHz AXI slave port for 64-bit, 2,400 MT/s DDR4 SDRAM, the SDRAM interface is half-duplex; read and write operations must alternate rather than occur simultaneously. Consequently, the maximum bandwidth during full-speed read and write equates to only half the effective width of the AXI bus. For the initial layers, the 512-bit AXI bus's read and write channels are fully occupied, causing delays as MIG cannot handle such a large data volume without waiting.

In further simulation, we limited the AXI port's throughput to one transfer every two clock cycles, resulting in 631,701 clock cycles, which is close to the actual FPGA-measured outcome. Note that restricting transfer rates in the simulation might yield worse results than actual performance, as we did not implement outstanding transactions in the simulation. This limitation means that if there are bursts of high data transfer, the simulation cannot balance the load by adding a FIFO.

On the other hand, the measured performance of the SDRAM4x2 configuration reached 84.8% of the simulation result, falling within the reasonable margin of performance loss attributed to the MIG controller. However, despite its higher efficiency, the achieved throughput is still lower than that of the SDRAM2x4BG configuration. This indicates that while using wider ports may result in efficiency loss, it positively impacts throughput, making it a reasonable choice.

4.3.5.3 URAM4x4BG on VU13P

Splitting a single accelerator instance into multiple instances with minimized inter-instance routing not only enhances throughput and simplifies meeting target frequency requirements but also enables implementing a highly parallel accelerator across multiple SLRs. We attempted to use two SLRs on the VU13P for a URAM4x4BG configuration, with each SLR containing two instances, and each pair having independent copies of weight memory. Thus, cross-SLR connections consist only of control signals and memory initialization buses. These signals have input/output registers at the relevant module ports, mostly operating at $clk1x$, with a few multi-cycle paths working at $clk2x$, meeting timing requirements consistent with $clk1x$. Consequently, even without using Laguna registers for inter-SLR

communication and instead connecting directly, the timing requirements can be met fairly easily.

The floorplan is shown in Figure. 4.11. Compared to the previous setup using only one SLR, using two SLRs requires more considerations. First, we placed the modules shared among the four instances at the top of SLR2 to minimize the distance to SLR3 at the top. These modules mainly communicate with the MIG, and to avoid occupying space designated for the accelerator, we continue using the SDRAM located on SLR1, as in the previous floorplan. However, since the connection from the center of SLR1 to the top of SLR2 is quite long, pipeline registers are required.

We used AXI register slice IP in timing-driven automatic multi-SLR crossing mode for the connections. In this mode, the IP automatically generates multi-stage pipeline registers based on the transmission distance and creates a FIFO at the destination end to avoid using control signals for long-distance pipeline registers. For the ADATA-related ports, half of the endpoints are at the top of SLR2 and the other half at the bottom of SLR3. When transferring data from the accelerator to the MIG (AR, AW, and W channels), the AXI register slice consolidates the src end signals from different SLRs near the destination MIG slave port using multiple chained registers. However, when the MIG responds to the accelerator's read/write commands (R and B channels), due to the requirement that the FIFO on the dest end of the AXI register slice must be placed together, signals can only be transmitted to the top of SLR2. Further transmission to SLR3 skips registers, and the ADATA input ready signal does not pass through registers, causing the maximum frequency to drop sharply to below 400 MHz.

To address this, we added handshake registers between modules on the ADTA AXI ports for each instance by channel. After adding these, the accelerator's maximum frequency increased to 519.5MHz. The failure to achieve the target frequency of 600 MHz is not due to cross-SLR paths but rather the relatively uniform distribution of inter-die connection points created using silicon via technology. Multiple AXI Slice Register IPs are scattered across SLR2 (the sparse yellow dots in SLR2 of Figure 4.11), consuming a large number of vertical routing resources. This prevents the computation and weight modules located in SLR2 from being compactly placed, ultimately lowering the frequency. Although relocating the AXI Interconnect module to the top of SLR2 could reduce the number of paths crossing SLR2, it would excessively encroach on the accelerator's available

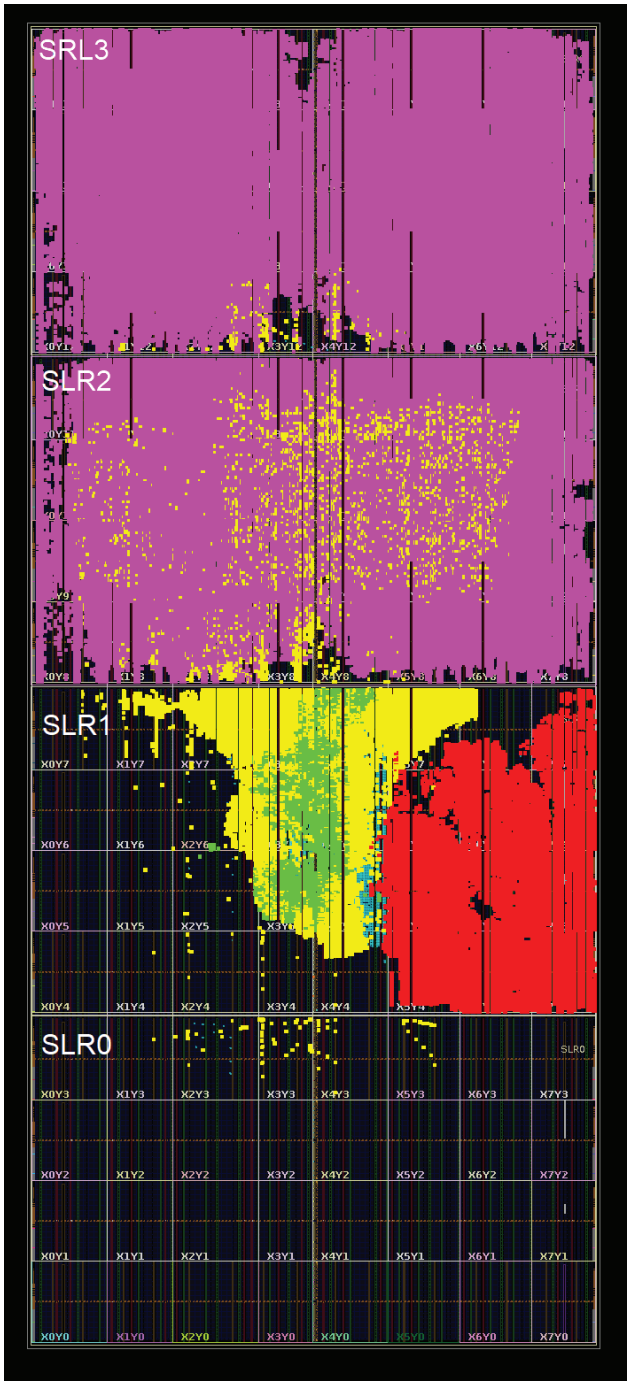


Figure 4.11: The floorplan for single instances crossing two SLRs.

area. Thus, we had to accept the current result. The implementation details are shown in Table. 4.23, and the actual operating frequency obtained by adjusting the MMCM is in the bracket. Actual verification on the VU13P accelerator showed that URAM-based instructions ran for 523,093 clock cycles, achieving a throughput of 7,933.6 FPS, reaching 96.1% of the simulation results; SDRAM-based instructions ran for 923,216 clock cycles, achieving a throughput of 4,495.2 FPS, only 69.1% of the simulation. This discrepancy stems from the same reasons as for URAM2x4BG running SDRAM-based instructions: we limited port transfer speed during simulation, which then took 930,062 clock cycles, longer than actual runtime.

4.3.6 Implementation on Ultra-Small Devices

Section 3.11.3 introduces a method to implement `concat` and split the execution of inverted residual blocks without any overhead, enabling our accelerator to fit ultra-small devices. Spartan and Artix devices have very limited hardware resources, and MIG and AXI interconnect components can take up more than half of available resources. Therefore, we used the smallest devices in the ZYNQ 7000 series, specifically the XC7Z010CLG400-1 (7Z010) and XC7Z007SCLG400-1 (7Z007S), as listed in Table. 4.1. The $DC = 4$ configuration would require at least 124 DSPs ($P_{PE} = DC \times 2$), which is too large to fit. We reduced it to $DC = 2$, lowering DSP requirements to 42. Due to the limited LUT resources, adapting our accelerator became challenging, requiring us to move some accumulation operations to the DSPs using ALU24 mode, increasing DSP utilization to 50%.

Compared to the DSP48E2 [30] in the UltraScale series (with a multiplier width of 27 $\times$ 18 bits), the DSP48E1 [29] in the 7 series has a multiplier width of only 25 $\times$ 18 bits. Thus, when using DSP packing, the cascade lacks sufficient guard bits, requiring additional LUTs for addition in the 7 series, reducing area efficiency. For the 7Z007S, this is still insufficient, so we allowed the remaining DSPs to inefficiently perform 8 $\times$ 8 multiplications in the PEDW Array.

In the previous configuration, the accelerator's instruction memory used 631 LUTs, which is excessive for small devices. We moved this memory to off-chip memory and accessed it via an additional AXI port when needed. Due to the limited BRAM in the 7Z007S, we switched the BRAMs used for the SWU and DWC to a lower-frequency but more resource-efficient configuration. The

Table 4.23: Implement Result of URAM4x4BG on VU13P

Device		VU13P
Configuration		URAM4x4BG
Directive		ENDh
Overall	LUT	595180
	LUTRAM	65533
	FF	924871
	BRAM	899.5
	URAM	256
	DSP	5059
	Power(W)	88.264
Accelerator	LUT	471987
	LUTRAM	42007
	FF	751793
	BRAM	736
	URAM	256
	DSP	5056
	Power(W)	67.542
$clk2x$(MHz)		519.5(518.75)
Sim. CC	URAM	502939
	SDRAM	637523
Act. CC	URAM	523093
	SDRAM	923216
Act. FPS	URAM	7933.58
	SDRAM	4495.156
Act. Band (GiB/s)	URAM	4.696508
	SDRAM	12.87997

remaining BRAMs are used for weight memory, with approximately 32 to 36 available. For each PE Array, we used 16 BRAMs, as there are two groups in total, with 8 BRAMs per group. The height of cascaded DSPs is two, so each position has four available DSPs. At this point, the port width is 16, and the total depth of 4 BRAMs is 8,192.

Finally, we successfully adapted the accelerator to these ultra-small devices. The resource utilization is shown in Table. 4.24, with a maximum operating frequency of 330.8 MHz for $clk2x$ on the 7Z010 and 230 MHz on the 7Z007S. When deployed on 7-series FPGA devices, the maximum frequency varies by architecture. For the Artix fabric used in the 7Z010 and 7Z007S, the maximum frequency is about 330 MHz, while for the Kintex/Virtex fabric, it reaches 430 MHz, far below the 600 MHz of the UltraScale+ series.

When running MobileNetV2 instructions, starting from block 6, additional independent weight initialization time is required, which in the previous configuration was only needed after block 11. Block 14 was split into two parts. The second 1×1 convolution layer in block 16 and the head convolution layer were each divided into four parts, and the final fully connected layer was split into 16 parts.

Simulation results showed that 5,631,128 clock cycles were required, resulting in a throughput of 28.4 FPS on the 7Z010 and 20.4 FPS on the 7Z007S. Actual measurements on the 7Z010 development board recorded 5,640,858 clock cycles, equivalent to 28.4 FPS.

In this section, we successfully deployed our accelerator on an ultra-small FPGA, using the zero-overhead `concat` method. This further demonstrates that the improved accelerator requires significantly fewer resources than other methods, highlighting its flexibility and scalability.

Table 4.24: Implement Result of SDRAM1BG on 7Z010 and 7Z007S

Device		7Z010	7Z007S
Configuration		SDRAM1BG	
Directive		-	-
Overall	LUT	14187	14122
	LUTRAM	4733	5369
	FF	15345	16316
	BRAM	58	50
	URAM	-	-
	DSP	50	66
	Power(W)	2.496	2.029
Accelerator	LUT	13051	13005
	LUTRAM	4627	5263
	FF	13662	14643
	BRAM	58	50
	URAM	-	-
	DSP	50	66
	Power(W)	0.896	0.501
$clk2x$(MHz)		330.8(320)	230
Sim. CC	URAM	-	-
	SDRAM	5631128	5631128
Act. CC	URAM	-	
	SDRAM	5640858	
Act. FPS	URAM	-	
	SDRAM	28.36448	
Act. Band (GiB/s)	URAM	-	
	SDRAM	0.187189	

Chapter 5

Analysis

In the experimental chapter, we analyzed many metrics. In this chapter, we will provide additional analysis on aspects that were not previously covered. The relevant content of Sections 5.1 and 5.2 has been presented in [36].

5.1 Bandwidth Savings

Since our goal is to increase throughput by reducing external memory accesses, in this section we discuss how much memory access our accelerator saves compared to a typical Overlay architecture accelerator.

First, we look at the unmodified design implemented on the ZYNQ platform prior to the improvements from Section. 3.11. We calculated the memory bandwidth required by the accelerator, as shown in Table. 5.1. For the URAM version, only weights, input images, and output results need to be transferred during runtime. In contrast, the SDRAM version requires additional transfers of feature maps between inverted residual blocks, so its bandwidth demand is higher than the URAM version. Both versions avoid transferring feature maps within the inverted residual blocks. The number of channels within an inverted residual block in MobileNetV2 is six times that between blocks. Therefore, our design in SDRAM mode theoretically saves up to 88.9% of external memory access when transferring feature maps. When running three-layer blocks, the average savings is 87.66%, slightly lower than the theoretical value of 88.9% due to varying input and output sizes in certain blocks. The overall model savings is 77.46%.

The multi-instance computation modules that share weights and process multiple images in parallel saved a significant amount of bandwidth for weight transfers. ZU7 SDRAM4 requires 2.4 times the data transfer amount of ZU3 SDRAM1 but processes four times the number of images. If multiple images could not be processed in parallel with shared weights, maintaining

Table 5.1: Off-chip SDRAM Access

	Single Run	Bandwidth	Feature Map/Frame
ZU7 URAM4	5203.75KiB	2.92GiB/s	394KiB
ZU7 SDRAM4	16796.75KiB	7.13GiB/s	3292.25KiB
ZU3 SDRAM1	6920KiB	3.87GiB/s	3292.25KiB

the same throughput on ZU7 SDRAM4 would require an additional 4.62 GiB/s bandwidth to transfer weights needed for the extra three images per batch.

For example, with ZU7 SDRAM4, if all data were transferred from/to external memory and FPS remained unchanged, the read/write data for (b) would require an additional 11.48 GiB/s bandwidth. Similarly, (c) would need an additional 7.74 GiB/s, plus the aforementioned 4.62 GiB/s, totaling 30.97 GiB/s. According to our measurements, when ZYNQ UltraScale+ is equipped with 64-bit 2,400MT/s DDR4 SDRAM, the maximum bandwidth for the PL side accessing PS-side SDRAM is about 13 GiB/s, which cannot meet these demands. Thanks to our bandwidth-saving design, even the highest-bandwidth-demanding design doesn't fully utilize the platform's maximum bandwidth.

Next, we provide a detailed analysis of how the proposed method impacts bandwidth savings. Figure. 5.1 shows the total data transfer needed to infer a single image at each location when running MobileNetV2 for all three-layer blocks, and (a) and (d) include inputs and outputs for two-layer blocks as well as other standalone layers.

The bandwidth values in the results tables of the previous sections were calculated based on the values in Figure. 5.1 and subsequent Figure. 5.2. For (a) to (f), the calculation is based on the actual transmitted feature map as $H_{FM} \times \lceil \frac{W_{FM}}{2} \rceil \times 2 \times C'_{FM} \times BW_A$, where BW_A is the quantization bit width, and C_{FM} is padded according to the rules described in previous sections, to the nearest multiple of 8, P_{PE}, or $2 \times P_{PE}$ is explicitly padded to the nearest multiple of 2 in the formula. During the final fully connected layer of the model, since the feature map size is 1×1, there are no two pixels, leading to a small amount of waste.

For (g), the transmission amount is calculated based on the padded channel count. For conventional convolutions, the formula is $K \times K \times C'_{IFM} \times C'_{OFM} \times BW_W + 2 \times C_{OFM} \times BW'_Q$, where BW_W is the quantization bit

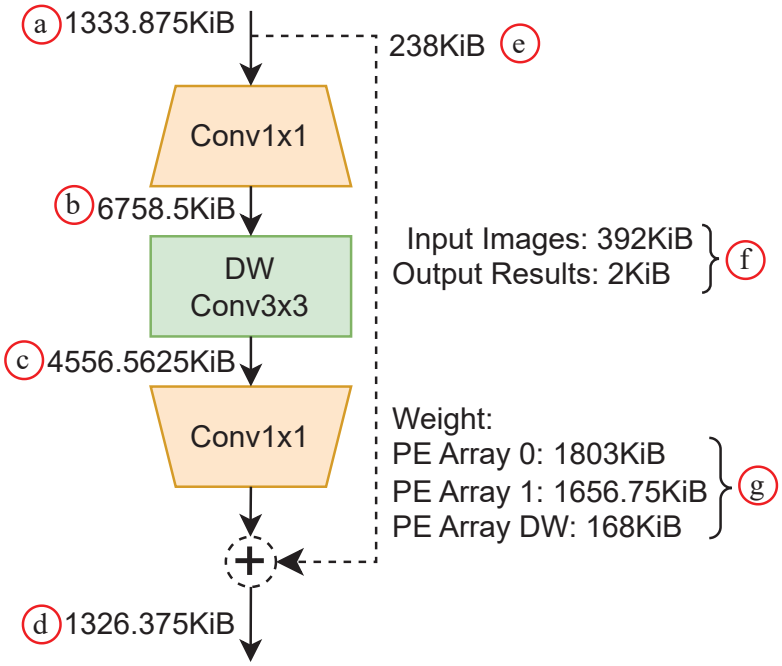


Figure 5.1: The amount of data transfer required for inferring a single image.

width, and BW'_Q is the bit width of the quantization parameter, which is 18, rounded up to the nearest power of 2. The factor of 2 accounts for the two quantization parameters, γ and β. The fully connected layer calculation is similar. Depthwise convolutions do not require multiplication by C_{IFM} .

The calculations from ⓐ to ⓖ do not account for additional waste caused by AXI burst transfers. Simulations can include this waste, and the ratio of bandwidth calculated from the table to the simulated results varies by scenario. When the burst length is 16:

- For ZU3 SDRAM1, the ratio is 99.55%.
- For ZU7 URAM4, the ratio is 99.42%.
- For ZU7 SDRAM4, the ratio is 99.82%.

When our accelerator infers multiple images simultaneously, ⓐ to ⓕ increase based on the number of images, while ⓖ remains unchanged. Considering a real-world scenario: when the batch size is 4 and throughput is approximately between the ZU7 SDURAM4 and ZU7 URAM4 results, around 2,000 FPS, it would save bandwidth equivalent to $ⓖ \times 2000 \times \frac{3}{4} =$

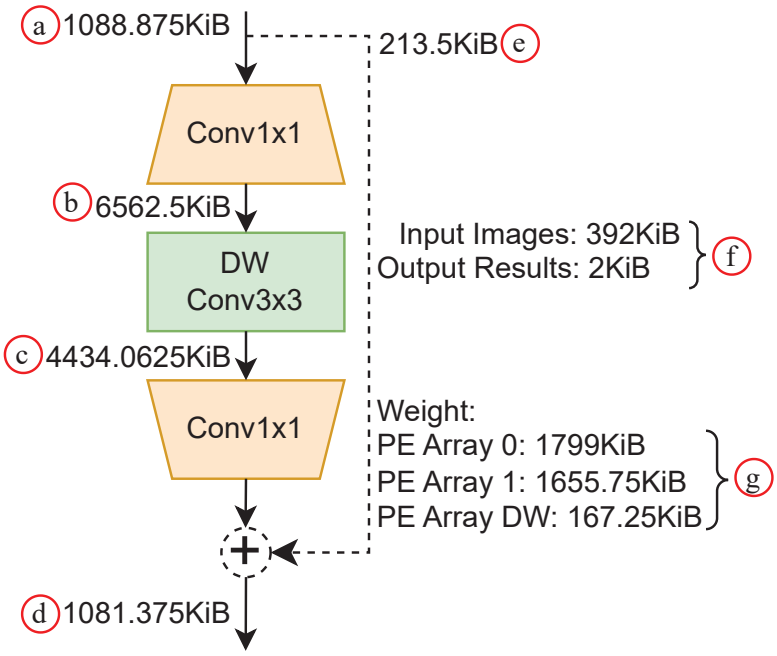


Figure 5.2: The amount of data transfer required for inferring a single image with imporved design.

5.19 GiB/s. Our block-level execution method, which only stores inter-block data in external SDRAM, saves $(ⓑ + ⓒ) \times 2000 = 21.58$ GiB/s of bandwidth. Temporarily storing this data on-chip using URAM saves $(ⓐ + ⓓ + ⓔ) \times 2000 = 5.53$ GiB/s of bandwidth, and the final required bandwidth is $ⓕ \times 2000 + ⓖ \times 2000 \times \frac{1}{4} = 2.48$ GiB/s. This shows that our block-level execution method contributes the most savings. When using a typical Overlay architecture accelerator, the total bandwidth requirement is the sum of the previous results: 34.78 GiB/s. For a Dataflow architecture accelerator, the required bandwidth is $ⓕ \times 2000 = 0.75$ GiB/s.

These calculations show that our method saves approximately 76.97% (SDRAM version) to 92.87% (URAM version) of bandwidth compared to Overlay architecture accelerators. Compared to Dataflow architecture accelerators, we have only minor weight transfer overhead, giving our architecture better scalability and flexibility.

5.2 Achieving the Throughput Limit

For the case with the highest bandwidth in Table. 5.1, ZU7 SDRAM4, we analyzed the bandwidth usage during the execution of each block/layer. Due to the complexity of the actual runtime, the exact timing for each module and memory sampling module is staggered, and their execution durations do not fully align. This makes it difficult to determine when one module in a block completes its current task and moves on to the next block while other modules are still finishing up their tasks. As a result, these overlapping times cannot be easily attributed to a single block. Given the accuracy of our throughput estimation model, we analyzed the execution time based on this model. Figure. 5.3 shows the ratio of the average off-chip memory bandwidth to the execution time of each current block, where the green area includes feature map transfers and the blue area represents weight transfers.

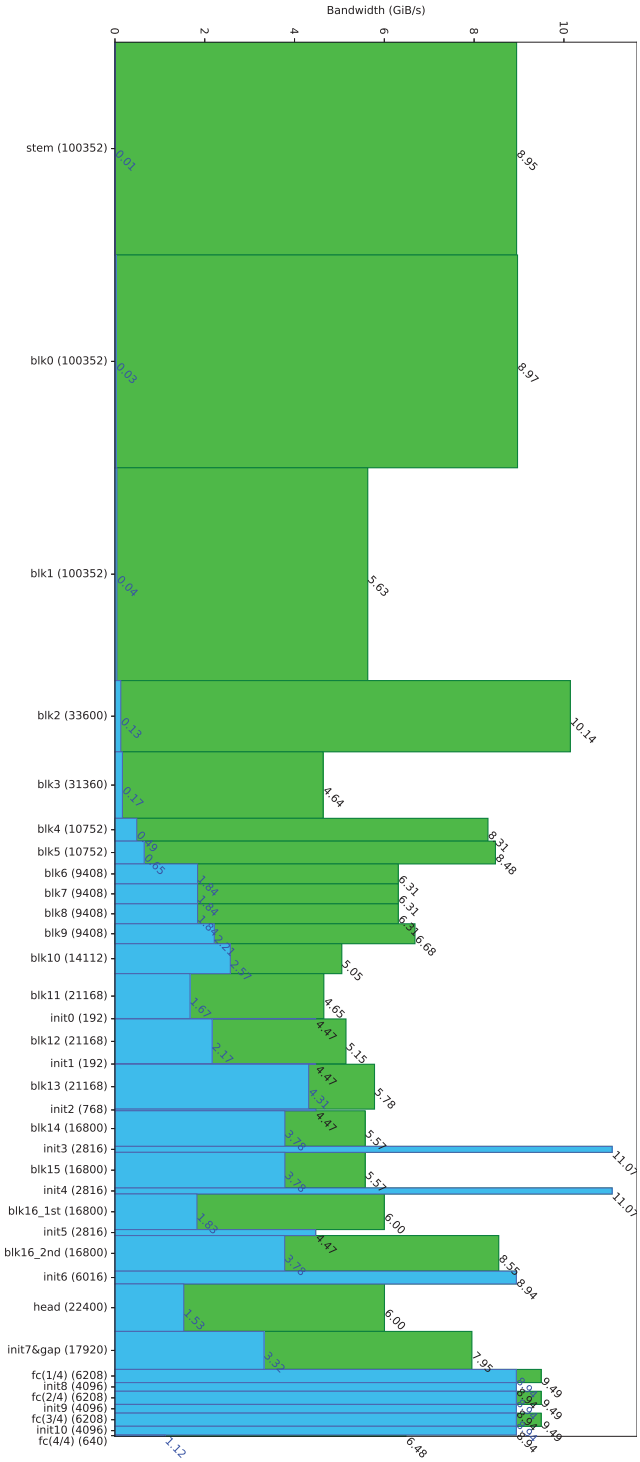


Figure 5.3: Average off-chip memory bandwidth requirements in GiB/s for each block/layer when running MobileNetV2 with ZU7 SDRAM4.

The decimal labels on top of each bar show the average off-chip memory bandwidth requirements in GiB/s for feature map and weight transfers, with blue labels indicating only weight transfers. The integer numbers in brackets after labels represent the number of $clk1x$ clock cycles required to execute each block/layer. For clarity, the bar width also represents execution time.

The chart shows that AXI port memory access is relatively high when executing the initial and final blocks/layers, but lower for the middle blocks. Early on, feature map transfers dominate, while weight transfers dominate later on. When executing the "stem" convolution layer and "blk0", we can see that the memory access frequency, excluding the blue weight transfers, reaches around 9 GiB/s on average. These layers lack shortcut paths and only use ADATA's MAXI0 port, meaning the read and write channels for that port are fully utilized. Additionally, "blk2" has an average memory access bandwidth of 10.14GiB/s due to a shortcut path that divides the load between the read and write channels of ADATA's MAXI0 port and the read channel of MAXI1, resulting in an average of 3.38GiB/s per channel, without reaching full capacity. Although these blocks/layers saturate more than one AXI port's bandwidth, the sequential read/write operations are easily managed by the hard memory controller on the PS side of the ZYNQ UltraScale+.

In the fully connected layers at the end, which involve a large number of parameters, the weight initialization module consumes a very high bandwidth. The blue portion of these bars, representing weight transfer, shows an average of around 9GiB/s (with an additional "init6" for weight initialization in the head convolution layer). This indicates 100% utilization of the read channels of the AXI master ports of the weight initialization modules in PE Arrays 0 and 1. However, the hard memory controller on the PS side does not become a bottleneck for these fully loaded sequential read ports.

Among all tasks, "init3" and "init4" show the highest transfer rates. However, the narrow width of these bars indicates that these initializations take very little time relative to the other instructions. Even if a higher bandwidth requirement slightly extends the initialization time, the overall performance impact is minimal, as confirmed by our measurements.

The weight transfers in the blue area are also necessary for the URAM version. Additionally, in the URAM version, image prefetching can begin as early as during the execution of "blk2" and should be completed before image processing ends. This prefetching can utilize idle bandwidth flexibly,

ensuring more stable throughput for URAM.

We tested performance degradation when the CPU consumes its maximum SDRAM bandwidth to assess the worst-case impact on our accelerator. The CPU's peak memory bandwidth is 14.06 GiB/s, which drops to 10.44 GiB/s when the accelerator is running. As SDRAM bandwidth usage approaches its limit, our ZU7 SDRAM4 performance decreases to 1,558 FPS.

Next, we aimed to achieve maximum throughput under bandwidth limitations. Since the URAM version's upper limit is too high for current FPGA devices, we could only test the SDRAM version with a lower limit. The ZU7's hardware resources are insufficient to achieve throughput at the PS-side DDR4 SDRAM bandwidth saturation limit. Utilizing the CPU to use part of the bandwidth results in dynamic bandwidth allocation by the memory controller, which cannot establish a "hard" limit.

After careful consideration, we decided to limit the maximum bandwidth via the AXI interface. By connecting all AXI master ports of our accelerator through a shared AXI SmartConnect to the PS, we limited our accelerator's memory bandwidth to 4.47 GiB/s $\times$ 2 (full duplex). In contrast to Figure. 3.1, where half-duplex SDRAM is used, we needed to consider reads and writes separately here. In this setup, the throughput limit was 1,632 FPS. We measured an actual throughput of 1,531 FPS on the evaluation board, reaching about 94% of the upper limit.

Table. 5.2 provides the maximum bandwidth required per layer/block, with separate statistics for read and write bandwidths, along with the ratio of required to available bandwidth. When the ratio is below 1, bandwidth is not a bottleneck; when it exceeds 1, bandwidth becomes a bottleneck. In this case, multiplying the ratio by the originally estimated runtime yields the estimated runtime under bandwidth constraints. From this table, we see that write operations never become a bottleneck, while read operations are a bottleneck in the initial and final layers, reducing throughput. Layers or blocks that do not become bottlenecks do not fully utilize the available bandwidth, hence achieving only 94% of the limit. This indicates that our accelerator can effectively utilize available bandwidth in most layers.

The data transfer requirements throughout the model when running one image with the improvements related to P_{PE} enabled design are shown in Figure.5.2. In Figure. 5.2, compared to Figure. 5.1, for ⓐ to ⓕ, only the channel counts need to be padded to the nearest multiple of 8 or 2×8. For ⓖ, the number of invalid channels is reduced, and the padded

Table 5.2: Ratio of required bandwidth and available bandwidth.

	Bandwidth(GiB/s)		Ratio	
	read	write	read	write
stem	4.476	4.470	1.001	1.000
blk0	4.497	4.470	1.006	1.000
blk1	4.514	1.118	1.010	0.250
blk2	6.806	3.338	1.522	0.747
blk3	3.743	0.894	0.837	0.200
blk4	5.701	2.608	1.275	0.583
blk5	5.867	2.608	1.313	0.583
blk6	4.820	1.490	1.078	0.333
blk7	4.820	1.490	1.078	0.333
blk8	4.820	1.490	1.078	0.333
blk9	5.193	1.490	1.162	0.333
blk10	3.563	1.490	0.797	0.333
blk11	3.659	0.993	0.819	0.222
init0	4.470	0.000	1.000	0.000
blk12	4.153	0.993	0.929	0.222
init1	4.470	0.000	1.000	0.000
blk13	5.308	0.473	1.187	0.106
init2	4.470	0.000	1.000	0.000
blk14	4.977	0.596	1.113	0.133
init3	11.074	0.000	2.477	0.000
blk15	4.977	0.596	1.113	0.133
init4	11.074	0.000	2.477	0.000
blk16_1st	2.423	3.576	0.542	0.800
init5	4.470	0.000	1.000	0.000
blk16_2nd	7.357	1.192	1.646	0.267
init6	8.941	0.000	2.000	0.000
head	2.427	3.576	0.543	0.800
init7&gap	7.791	0.160	1.743	0.036
fc(1/4)	9.402	0.092	2.103	0.021
init8	8.941	0.000	2.000	0.000
fc(2/4)	9.402	0.092	2.103	0.021
init9	8.941	0.000	2.000	0.000
fc(3/4)	9.402	0.092	2.103	0.021
init10	8.941	0.000	2.000	0.000
fc(4/4)	5.588	0.894	1.250	0.200

C_{IFM} for conventional convolutions and C_{OFM} for depthwise convolutions are both decreased. However, to avoid increasing the complexity of the weight initialization module, C_{OFM} is still padded to the nearest multiple of P_{PE} or $2 \times P_{PE}$ for weight initialization, even though the padded weights may not always participate in computations. Similarly, simulations were used to assess the waste caused by burst transfers. The ratio of bandwidth calculated from the table to simulation results also varies by scenario. When the burst length is 16, across all scenarios (including results calculated without the improvements in Figure. 5.1), the best case is 99.95% (VU13P SDRAM4x4BG). The worst case is 99.42% (ZU7 URAM4).

Our implementation on the high-end FPGA device VU13P achieves ultra-high parallelism, but without a multi-instance shared-port design, we cannot fully utilize the bandwidth provided by the 64-bit 2,400 MT/s DDR4 SDRAM. The design with two instances sharing ports barely utilizes the bandwidth fully when executing the "stem" convolution layer as shown in Figure. 5.4 but does not exceed it, while the design with four sub-instances sharing ports can exceed it.

Figure. 5.5 shows the average bandwidth utilization when running each block or layer in the SDRAM2x4BG scenario. The figure shows that only the "stem" convolution layer, "blk0", and "init7 & gap" fully utilize the bandwidth.

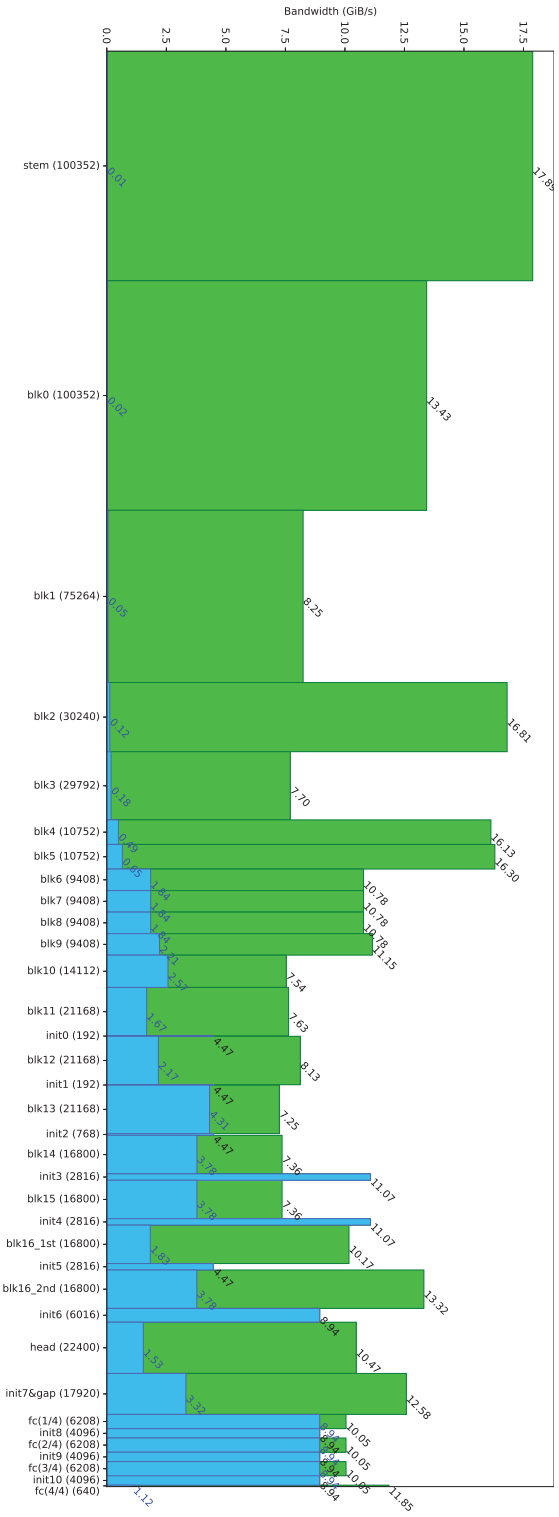


Figure 5.4: Average off-chip memory bandwidth requirements in GiB/s for each block/layer when running MobileNetV2 with SDRAM4x2BG on VU13P.

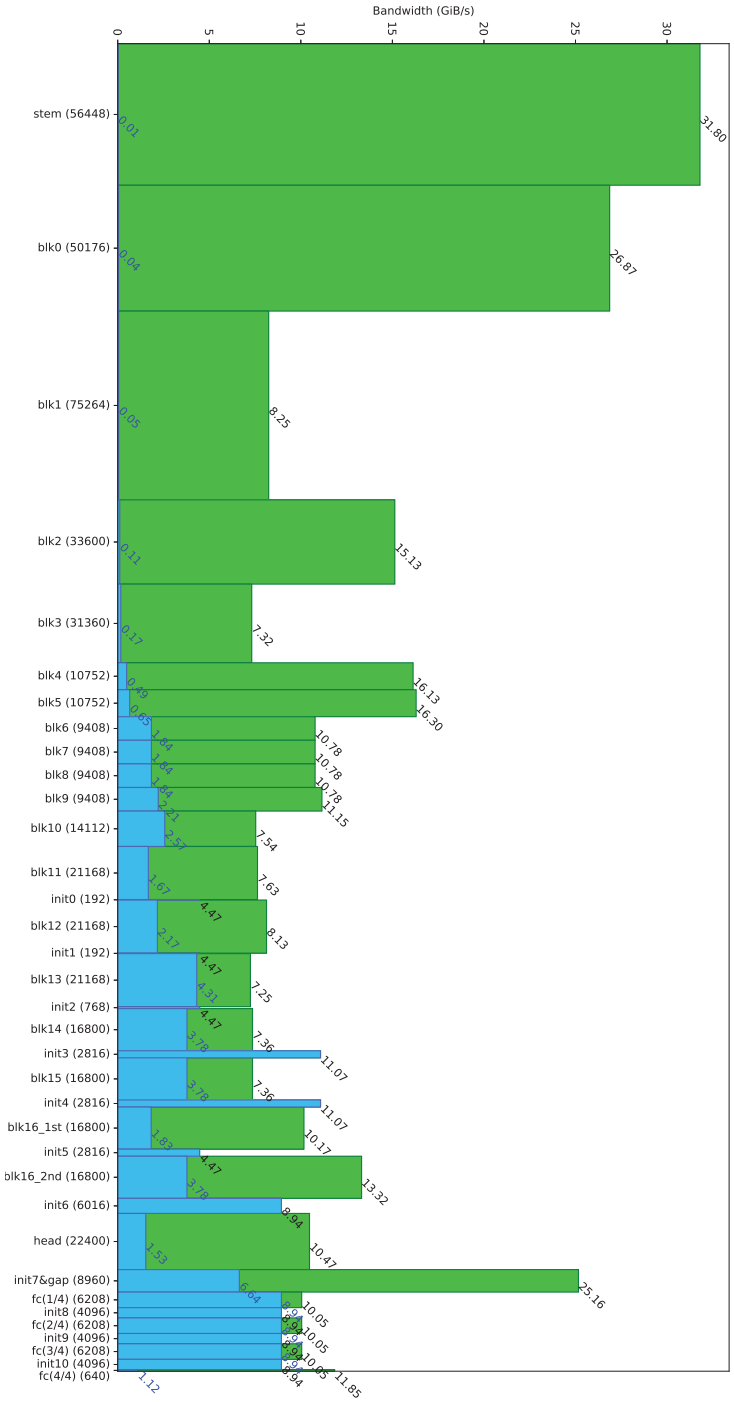


Figure 5.5: Average off-chip memory bandwidth requirements in GiB/s for each block/layer when running MobileNetV2 with SDRAM2x4BG on VU13P.

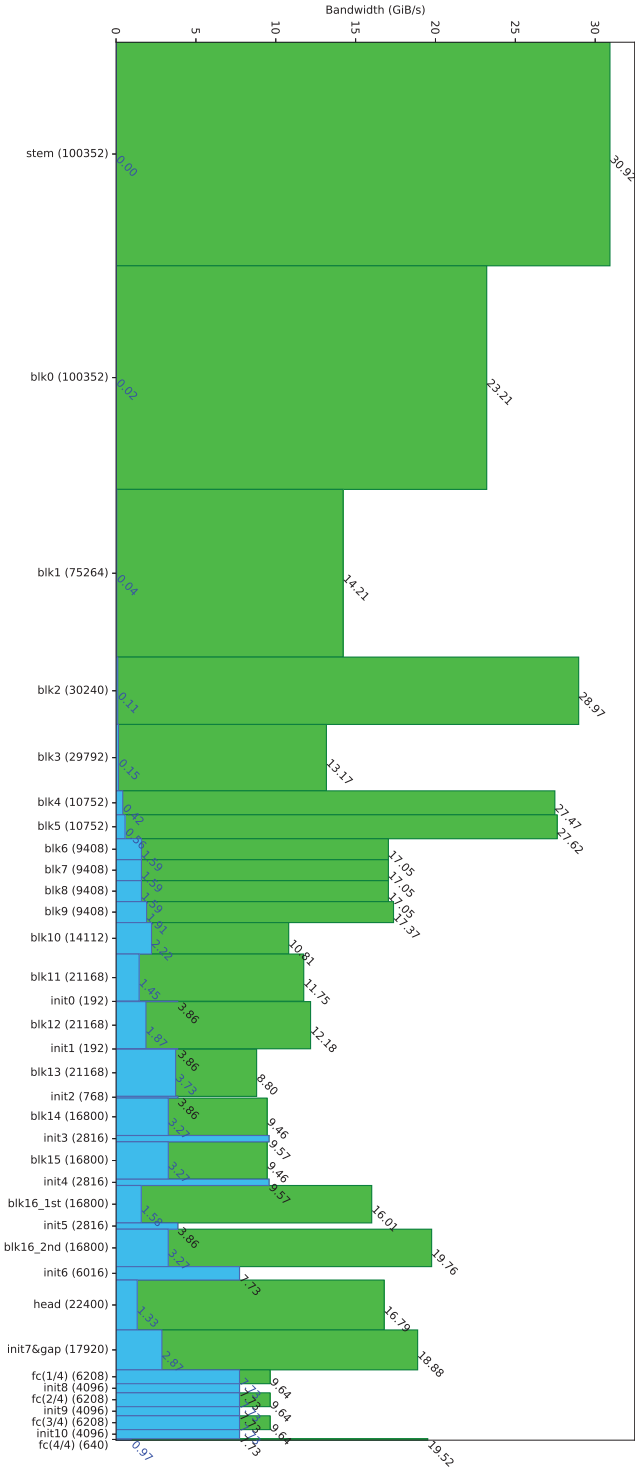


Figure 5.6: Average off-chip memory bandwidth requirements in GiB/s for each block/layer when running MobileNetV2 with SDRAM4x4BG on VU13P.

For an ultra-high parallelism URAM4x4BG accelerator built using two SLRs on the VU13P running SDRAM-version instructions, the average data transfer per clock cycle per layer or block is shown in Figure. 5.6. The figure shows that in most layers, all available DDR4 SDRAM bandwidth is fully utilized. For SDRAM4x4BG using 64-bit 2,400 MT/s DDR4 SDRAM, the theoretical throughput upper limit is 6,241.418 FPS (when MIG efficiency is 100%, but unfortunately, under absolutely ideal conditions, the efficiency can only reach 91%.), while our accelerator achieved an actual throughput of 4,495 FPS, reaching 72%. Since the latter layers or blocks did not fully utilize bandwidth, further throughput improvement would require increased parallelism, particularly in the number of PEs per image (P_{PE}). However, the device capacity of VU13P no longer allows us to further increase parallelism. Therefore, even with a large accelerator using resources from two SLRs, full utilization of the off-chip memory bandwidth is unachievable, highlighting the low memory bandwidth required by our accelerator.

These experiments show that our accelerator's performance remains relatively stable even under limited bandwidth conditions. However, achieving full bandwidth utilization to reach the throughput limit is challenging with current devices, further demonstrating the low off-chip memory bandwidth requirement of our accelerator.

5.3 Effective DSP Utilization Ratio

Our accelerator primarily relies on DSPs for computation, making the effective DSP utilization ratio a key efficiency metric. Performance counters were integrated into the simulation to monitor this indicator. As shown in Table. 5.3, the results for ZU3 SDRAM1 and ZU7 URAM4 are almost identical, while ZU7 SDRAM4 shows a noticeable drop in performance. Furthermore, the DSP utilization ratio for PE Array 0 is much higher than for PE Array 1. This is because the early layers require a longer execution time but have fewer output channels, making parallel computation with dual PE Arrays less effective. As a result, PE Array 1 remains idle for extended periods. Additionally, as the feature map size decreases, PE Array 1's workload significantly declines. When running the entire model, the poorer performance is primarily due to the fully connected layer, which has a low computation load but large weight size, leading to extended time for initialization.

Table 5.3: Effective Working Time Ratio of DSP

	The Entire Model		Tree-layers Block Only	
	PE Array 0	PE Array 1	PE Array 0	PE Array 1
ZU7 URAM4	82.08%	54.63%	95.06%	65.59%
ZU7SDRAM4	62.18%	41.39%	87.86%	61.20%
ZU3 SDRAM1	82.00%	54.58%	95.06%	65.59%

For this metric, our accelerator demonstrates a high DSP utilization ratio when processing the middle layers with a high computational load, indicating efficient use of hardware arithmetic resources. Table. 5.4 also provides the DSP utilization ratios in each PE Array per layer, calculated from the time the first valid data is received until layer execution is complete. Since the ZU3 SDRAM1 and ZU7 URAM4 results are nearly identical, only one is displayed.

Table 5.4: Effective Working Time Ratio of DSP for Each Block/Layer

	ZU7 URAM4		ZU7 SDRAM4	
	PE Array 0	PE Array 1	PE Array 0	PE Array 1
stem	100.00%		53.20%	
blk0		96.58%		23.72%
blk1	100.00%	25.36%	74.49%	19.41%
blk2	95.38%	93.45%	95.38%	93.45%
blk3	99.43%	26.19%	99.43%	26.19%
blk4	92.05%	87.92%	92.05%	87.92%
blk5	87.71%	87.96%	87.71%	87.96%
blk6	97.72%	53.66%	97.72%	53.66%
blk7	100.00%	100.00%	100.00%	100.00%
blk8	100.00%	100.00%	100.00%	100.00%
blk9	100.00%	100.00%	100.00%	100.00%
blk10	87.34%	100.00%	87.34%	100.00%
blk11	98.72%	100.00%	98.72%	100.00%
blk12	100.00%	100.00%	99.84%	100.00%
blk13	100.00%	51.53%	99.84%	51.53%
blk14	100.00%	100.00%	99.66%	100.00%
blk15	100.00%	100.00%	99.66%	100.00%
blk16_1st	100.00%		99.66%	
blk16_2nd*	100.00%	85.64%	82.43%	72.42%
head*	100.00%	100.00%	99.99%	100.00%
fc(1/4)*	99.85%	100.00%	58.21%	58.25%
fc(2/4)*	99.85%	100.00%	74.77%	74.89%
fc(3/4)*	99.85%	100.00%	74.77%	74.89%
fc(4/4)*	99.85%	100.00%	74.77%	74.89%

* Running at the parallel execution mode.

Chapter 6

Comparison

We compared our accelerator (the versions without further improvements, so there are no "B" or "G" marks in the configuration name) with other works. Table. 6.1 shows the implementation comparison on mid-range devices, and Table. 6.2 shows the comparison on cost-optimized devices. The content related to comparisons has been presented in [36].

In the mid-range device comparison, our ZU7 URAM4 used the fewest LUTs but outperformed other works in terms of FPS, FPS/DSP, and FPS/1kLUT, even when many of these works used the larger ZU9. Our ZU7 SDRAM4 achieved slightly lower FPS than others but maintained comparable throughput with significantly lower resource utilization.

In the cost-optimized device comparison, since few works implement the full MobileNetV2, we included works that use custom low-bit quantized networks. Our ZU3 SDRAM1 outperformed all other works, even surpassing most mid-range device implementations.

Regarding on-chip memory usage, our ZU7 URAM4 utilized a large amount of on-chip memory, whereas ZU7 SDRAM4 used less on-chip memory compared to most other works. ZU3 SDRAM1 had comparable on-chip memory capacity with other works.

In Tables. 6.1 and 6.2, [1] is a fast Overlay architecture accelerator on all types of devices, designed with dedicated modules for depthwise convolution layers. It establishes a data path from expanded convolution layers to depthwise convolution layers, allowing these two layers to execute continuously. On cost-optimized devices, its area efficiency (FPS per DSP and FPS per 1kLUT) is comparable to our design. However, on mid-range devices, its area efficiency drops significantly, falling far below our design. Its performance on mid-range devices cannot improve further because, according to our calculations, the average bandwidth is about 8.9 GiB/s, which should reach the peak limit. On the other hand, on cost-optimized devices, it

Table 6.1: Comparison on Mid-range Devices

	[1]	[51]	[7]	[52]	[8]	[14]*	[53]*	ours	
	2019	2020	2021	2021	2022	2023	2020	URAM4	SDRAM4
Network	MobileNetV2						MobileNetV1	MobileNetV2	
FPGA	ZU9	7K325t	7V690t	ZU9	ZU9	ZU9	Alveo U250	ZU7	ZU7
FPS	810	326	302	382	538	1910	1800	2350	1780
Top-1(%)	68.1	-	70.8	72.0	65.67	72.98	70.4	71.55	71.55
Prec.	*w*8*a*8	*w*8*a*8	*w*8*a*8	*w*8*a*8	*w*8/*w*4*a*5	*w*8*a*8	*w*4*a*4	*w*8*a*8	*w*8*a*8
DSP	2070	704	2160	576	2092	1283	109	1264	1264
BRAM	771	192.5	941.5	-	440.5	691	885	248	248
URAM	0	0	0	0	0	0	22	64	0
LUT	162k	174k	308k	125k	180k	170k	474k	123k	122k
FF	301k	241k	433k	143k	-	154k	463k	195k	197k
FPS/DSP	0.39	0.46	0.15	0.66	0.26	1.49	16.51	1.86	1.41
FPS/1kLUT	5.00	1.87	0.98	3.06	2.99	11.20	3.80	19.10	14.50

* Dataflow architecture accelerator.

Table 6.2: Comparison on Cost-optimized Devices

	[1] 2019	[8] 2022	[11]* 2018	[54] 2019	[55] 2021	[13]* 2021	ours SDRAM1
Network	MobileNetV2		Custom				MobileNetV2
FPGA	ZU2	7Z020	ZU3	ZU3	ZU3	ZU3	ZU3
FPS	205	132	183	66	48	77	586
Top-1(%)	68.1	65.57	50.3	68.3	71.8	70.06	71.55
Prec.	$w8a8$	$w8/w4a5$	$w1a2$	$w4a4$	$w1a1.4$	$w$mixed$a8$	$w8a8$
DSP	212	208	-	360	224	360	316
BRAM	145	123	216	159	201	215.5	158
LUT	31k	41k	36k	52k	51k	61k	39k
FF	47k	-	-	42k	-	55k	62k
FPS/DSP	0.97	0.73	-	0.18	0.21	0.21	1.85
FPS/1kLUT	6.61	3.20	5.08	1.27	0.94	1.26	15.03

* Dataflow architecture accelerator.

does not reach the bandwidth limit and is thus less affected by bandwidth constraints.

Their performance is lower than ours because they used a smaller ZU2 device as a cost-optimized option. Regarding FPS per DSP, our design's lead on cost-optimized devices is less than on mid-range devices.

[53] and [11] are Dataflow architecture accelerators built by FINN. As previously mentioned, running large models on them requires a large amount of on-chip memory. Even with the Alveo U250 (whose LUTs are 8.5× and DSPs are 7.1× that of the ZU7), its performance is comparable to ours. Due to the lack of DSP use for 4-bit multiplication, [53]'s FPS per DSP is extremely high. In [11], due to memory capacity limitations, custom networks were adopted. This work used low-bit binary neural networks (BNNs) with fewer parameters, so its accuracy is not comparable. Except for on cost-optimized devices, almost all works trade off between performance and accuracy. Our design's FPS per DSP and FPS per 1kLUT are much higher than other works, making our design highly area-efficient.

The Dataflow architecture accelerator introduced in [14] is targeted at MobileNetV2. Compared to our ZU7 URAM4, it achieves slightly lower throughput on a larger device, yet has significantly higher LUT utilization. As a Dataflow architecture accelerator, it is difficult to port to smaller devices, reducing its flexibility and scalability. It uses a modified version of MobileNetV2, resulting in higher accuracy than the original MobileNetV2.

Additionally, [14] proposed a hybrid approach of using both on-chip and off-chip memory for storing weights. According to the information provided in the paper, the final fully connected layer weights are stored off-chip. If all 3.53M parameters of MobileNetV2 were stored on-chip, at least 766 BRAMs with the capacity of 36Kbit would be required. However, by moving the final fully connected layer's weights off-chip, the requirement is reduced to just 488 BRAMs. This suggests that approximately 200 BRAMs are used for FIFOs, row buffers, and filter buffers.

In contrast, our design uses only 128 BRAMs for weights and 30 BRAMs per image processed in parallel for row buffers and filter buffers. Additionally, in the URAM version, 16 URAMs per image are allocated for inter-block buffering. As a result, the required on-chip weight memory capacity in our reference design is reduced to 16.7% of a conventional Dataflow architecture accelerator and 26.2% of [14].

If we use the minimum required memory depth of 1,600 instead of

the rounded-up value of 2,048—meaning all blocks are executed without splitting—then the BRAM usage for weights reduces from 128 to 96 (plus a small amount of LUTRAM). In this case, the required on-chip weight memory drops to 12.5% of a conventional Dataflow architecture accelerator and 19.7% of [14].

If split-execution is applied, as shown in Table 4.24 in the previous chapter, the BRAM usage for weights further decreases to 32. This value is already lower than the on-chip memory requirements of most Overlay architecture accelerators.

In Table. 5.1, we provide additional information about the amount of data transfer required to process a single frame by transferring feature map data to off-chip SDRAM. As [52] reported, their result was 3,392 KiB, slightly higher than our SDRAM version but significantly higher than our URAM version.

6.1 Running Other Networks

Our accelerator can be adapted to different network models by modifying the modules and interconnect structures shown in Figure. 3.2. Therefore, we also compare the performance of our accelerator architecture when running different networks. It also includes computationally intensive network models, for which the FPGA-related academic community mainly focuses on pruning and using lower bit quantization. Only commercial accelerator IP products support computing conditions consistent with ours, and there are also few accelerator works targeting these remaining atypical network models, so no comparison with other works is performed. The content related of this section has been presented in [36].

The example configuration in Figure. 3.2 enables our accelerator to run the Minimalistic version of MobileNetV3 [56] without any modifications. The Minimalistic version is a streamlined network more suited to specialized accelerators. Compared to the standard MobileNetV3, it replaces 5×5 convolution kernels with more area-efficient 3×3 kernels and removes the Squeeze-and-Excitation (SE) block [44]. The SE block globally averages the feature map of the depthwise convolution output, then applies two fully connected layers (or 1×1 convolution layers) to "squeeze" and "expand" back to the original channel count to compute scaling weights for each channel. This scaling weight is then multiplied by the depthwise convolution

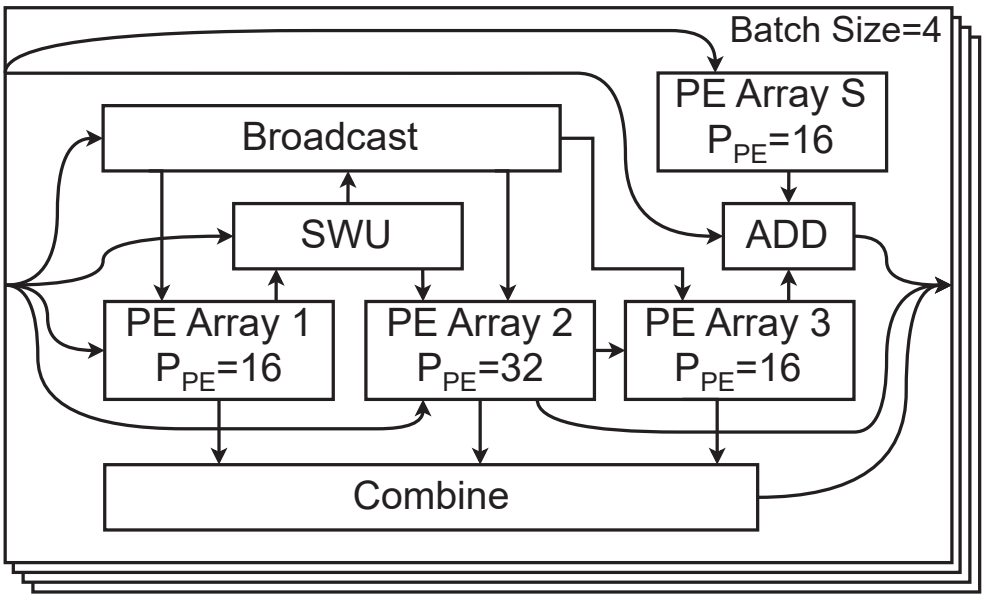


Figure 6.1: The configuration for ResNet-50.

result, channel by channel. Global pooling within the block requires that the depthwise convolution in the preceding module be fully completed to produce an output. After two fully connected layers, data from the previous layer must also be read, which reduces efficiency on both dedicated hardware and on Mobile CPUs (the original target of MobileNet). Thus, the SE block is removed in the Minimalistic version of MobileNetV3.

Running EfficientNet b0 requires replacing the SWU with a K=5 version. Additionally, some of the simplifications applied to the Minimalistic version are also necessary, such as removing the SE block. Due to the increase in the maximum convolution kernel size, the number of compute units required per PE in the PEDW Array increases to $\frac{25}{9}$ of the original amount. We halved P_{PEDW} to maintain reasonable LUT utilization.

EfficientNetV2 requires adding a path from PE Array 0 to PE Array 1. As its inter-block feature maps are too large to fit into the URAM of ZU7, there is no URAM version.

Running ResNet-50 requires modifying its structure to match that in Figure. 6.1, removing the PEDW Array and adding two additional PE Arrays. Along with the two original PE Arrays, PE Arrays 1 to 3 execute ①, ②, and ③ in Figure. 2.2b, while PE Array S executes ④. Given the large

Table 6.3: Evaluation of Running Other Networks

Network		GMACs	Params	DSPs	FPS	Bandwidth	GMAC/s
MobileNetV2	URAM4	0.3	3.53M	1264	2437	3.02GiB/s	731
	SDRAM4			1264	1849	7.20GiB/s	555
MobileNetV3 Small Minimalistic	URAM4	0.065	2.09M	1264	6236	5.53GiB/s	405
	SDRAM4			1264	3538	7.77GiB/s	230
MobileNetV3 Large Minimalistic	URAM4	0.209	4.00M	1264	2634	3.56GiB/s	551
	SDRAM4			1264	1869	7.79GiB/s	390
EfficientNet b0	URAM4	0.39	4.83M	1232	1766	2.75GiB/s	689
	SDRAM4			1232	1490	6.49GiB/s	581
EfficientNetV2-S(P)	SDRAM4	8.37	17.78M	1264	99	4.06GiB/s	826
EfficientNetV2-S(S)	SDRAM4	8.37	17.78M	1264	83	2.07GiB/s	699
ResNet50	SDRAM4	4.09	24.40M	1576	237	5.80GiB/s	968

weight size for each layer, URAM is also used to store weights. In the fourth stage, the weight size for even a single block exceeds URAM capacity, so a parallel execution strategy is used to process all layers in this stage.

In our evaluation, we considered the range of resource usage within the ZU7 and aimed to maximize weight memory capacity. The estimated results are shown in Table. 6.3. GMAC and Params represent the effective values provided by the original papers, open-source code repositories, or framework model libraries, excluding any extra overhead caused by rounding the channel count. GMAC/s (billion multiply-accumulate operations per second) is calculated based on these effective values, while bandwidth is based on the actual amount of data transferred. This approach aims to provide a conservative estimate.

Compared to the estimates in Table. 4.2, sufficient memory capacity resulted in a 3.6% (ZU7 URAM4) and 2.4% (ZU7 SDRAM4) throughput increase when running MobileNetV2.

For EfficientNetV2, there are two versions: P and S. In the new Fused-MBConv case, P denotes parallel execution using two PE Arrays, while S denotes serial execution. Due to the significant difference in computational

load between the two layers in Fused-MBConv, the S version has slightly lower performance, although it can save approximately half the memory bandwidth. Notably, in P mode, transitioning from serial execution mode to parallel execution mode occurs twice, incurring two penalty periods, CC_{P1} and CC_{P2}, compared to only one CC_P in S mode. Despite the additional switching overhead accounting for less than 0.1% of the total runtime, it results in a throughput improvement exceeding 19%. Thus, the benefits gained from the transition significantly outweigh the incurred overhead, making it a highly advantageous trade-off.

When running the classic ResNet-50, our accelerator does not significantly outperform other existing works in terms of performance per unit area. The reason is that computation resources are already approaching the device's limits. While our accelerator does not achieve significant throughput gains on computation-intensive networks, it requires less off-chip SDRAM bandwidth. Running ResNet-50 at a throughput of 237 FPS, our accelerator needs 5.80 GiB/s of SDRAM bandwidth. In comparison, the DPU (DPUCZDX8G B4096) achieves a throughput of 194 FPS for an 8-bit quantized ResNet-50 on the ZU9, but requires three parallel instances, with each instance averaging 3.06 GiB/s of bandwidth.

In computation-intensive models, GMAC/s is very high, approaching theoretical limits. On the other hand, when running lightweight models, the low computational load fails to fully saturate the PEs, causing a decrease in GMAC/s. For memory bandwidth, all scenarios remain within reasonable ranges.

We ran EfficientNet b0 and the Minimalistic version of MobileNetV3 on the evaluation board, with actual performance matching the estimated results at an average rate of 96.4%. The largest error occurred with MobileNetV3 Small Minimalistic, where the SDRAM version achieved only 89.1% of the estimated throughput. Simulation results were close to the estimated throughput, but in actual board-level execution, shorter computation time led to significant SDRAM latency impacts. The remaining results were close to the estimates.

6.2 Object Detection

In addition to splitting and executing inverted residual blocks, the method introduced in Section 3.11.3 also supports the `concat` operation, enabling

Table 6.4: FPS Matrix of URAM Version when Running YOLOv3

B \ $P_{PE_{3\times3}}$	8	16	24	32	40	48	56	64	72	80	88	96
1	1.2	2.5	3.5	4.8	5.5	6.4	7.1	8.9	9.0	9.4	9.9	9.9
2	2.5	4.9	7.0	9.5	11.0	12.8	14.3	17.8	18.0	18.8	19.7	19.8
3	3.7	7.4	10.5	14.3	16.5	19.2	21.4	26.7	27.0	28.2	29.6	29.7
4	5.0	9.8	14.0	19.1	22.0	25.6	28.5	35.6	36.0	37.7	39.5	39.6
5	6.2	12.3	17.5	23.8	27.6	32.0	35.6	44.5	45.1	47.1	49.3	49.5
6	7.4	14.7	21.0	28.6	33.1	38.4	42.8	53.4	54.1	56.5	59.2	59.4
7	8.7	17.2	24.5	33.4	38.6	44.9	49.9	62.3	63.1	65.9	69.0	69.3
8	9.9	19.6	28.0	38.2	44.1	51.3	57.0	71.2	72.1	75.3	78.9	79.2
9	11.2	22.1	31.5	42.9	49.6	57.7	64.2	80.1	81.1	84.7	88.8	89.1
10	12.4	24.5	35.0	47.7	55.1	64.1	71.3	89.0	90.1	94.2	98.6	99.0
11	13.7	27.0	38.5	52.5	60.6	70.5	78.4	97.9	99.1	103.6	108.5	108.9
12	14.9	29.4	42.0	57.2	66.1	76.9	85.5	106.8	108.1	113.0	118.4	118.8
13	16.1	31.9	45.5	62.0	71.6	83.3	92.7	115.7	117.1	122.4	128.2	128.7
14	17.4	34.3	49.0	66.8	77.1	89.7	99.8	124.6	126.2	131.8	138.1	138.6
15	18.6	36.8	52.5	71.5	82.7	96.1	106.9	133.5	135.2	141.2	148.0	148.5
16	19.9	39.2	56.0	76.3	88.2	102.5	114.1	142.4	144.2	150.6	157.8	158.4

more complex models to be run. To evaluate this, we used popular object detection networks as examples. The target networks are the commonly used SSD networks [57] (SSDMobileNetV2 [58], SSDliteMobileNetV2 [59]) and YOLOv3 [60], where the input image size is 300×300 for the two SSD networks and 416×416 for YOLOv3. Estimated throughput for each network is listed in Tables 6.4-6.9. The content of this section has been published in [42].

The YOLOv3 accelerator includes two PE Arrays with different parallelism for executing 3×3 and 1×1 convolutions in Darknet [61]. The 1×1 convolution PE Array has a much lower computational load than the 3×3 convolution PE Array, so in the evaluation, the parallelism of the 1×1 convolution array, $P_{PE_{1\times1}}$, is fixed at 8. For various values of $P_{PE_{3\times3}}$, the number of DSPs needed to process each image in parallel is 100, 128, 160, 192, 224, 256, 288, 320, 352, 384, 416, and 448. Therefore, the maximum throughput for ZU7, constrained by DSPs, is approximately 44.5 FPS, for a single SLR in the VU13P, around 71.2 FPS, and for the entire VU13P device, around $26.7\times7=186.9$ FPS.

By calculating FPS per DSP based on DSP count, we found that when $P_{PE_{3\times3}} = 64$, the throughput per unit area was highest, approaching a 9:1

Table 6.5: FPS Matrix of SDRAM Version when Running YOLOv3

B \ $P_{PE_{3\times3}}$	8	16	24	32	40	48	56	64	72	80	88	96
1	1.2	2.5	3.5	4.8	5.5	6.4	7.1	8.9	9.0	9.4	9.9	9.9
2	2.5	4.9	7.0	9.5	11.0	12.8	14.3	17.8	18.0	18.8	19.7	19.8
3	3.7	7.4	10.5	14.3	16.5	19.2	21.4	26.5	26.8	28.0	29.3	29.4
4	5.0	9.8	14.0	19.1	22.0	25.6	28.5	34.9	35.4	36.9	38.5	38.7
5	6.2	12.3	17.5	23.8	27.5	31.9	35.5	42.9	43.4	45.3	47.2	47.4
6	7.4	14.7	21.0	28.4	32.7	38.0	42.1	50.3	50.9	53.1	55.3	55.5
7	8.7	17.2	24.5	32.9	37.9	43.9	48.6	57.4	58.1	60.5	62.9	63.1
8	9.9	19.6	27.8	37.4	43.0	49.5	54.7	64.2	65.0	67.6	70.0	70.2
9	11.2	22.1	31.2	41.8	48.0	55.0	60.7	70.7	71.5	74.3	76.8	77.0
10	12.4	24.4	34.4	45.6	52.3	59.7	65.7	75.8	76.6	79.5	82.0	82.3
11	13.6	26.7	37.5	49.3	56.4	64.2	70.5	80.5	81.3	84.3	86.9	87.1
12	14.8	28.9	40.6	52.9	60.3	68.5	75.0	84.9	85.8	88.8	91.4	91.7
13	16.0	31.1	43.5	56.4	64.1	72.6	79.2	89.0	89.9	93.0	95.6	95.9
14	17.3	33.4	46.4	59.7	67.7	76.4	83.1	92.9	93.8	96.8	99.5	99.7
15	18.5	35.5	49.1	62.9	71.3	79.8	86.6	96.5	97.4	100.5	103.0	103.3
16	19.7	37.7	51.7	66.1	74.6	83.0	89.9	99.9	100.8	103.9	106.3	106.6

Table 6.6: FPS Matrix of URAM Version when Running SSD-MobileNetV2

B \ P_{PE}	8	16	24	32	40	48	56	64
1	68.2	132.6	174.0	234.6	250.6	271.3	274.6	296.6
2	136.4	265.3	348.0	469.2	501.3	542.5	549.2	593.1
3	204.6	397.9	521.9	703.8	751.9	813.8	823.9	889.7
4	272.8	530.6	695.9	938.4	1002.5	1085.1	1098.5	1186.3
5	341.1	663.2	869.9	1172.9	1253.1	1356.4	1373.1	1482.8
6	409.3	795.9	1043.9	1407.5	1503.8	1627.6	1647.7	1779.4
7	477.5	928.5	1217.8	1642.1	1754.4	1898.9	1922.4	2076.0
8	545.7	1061.2	1391.8	1876.7	2005.0	2170.2	2197.0	2372.5
9	613.9	1193.8	1565.8	2111.3	2255.6	2441.4	2471.6	2669.1
10	682.1	1326.4	1739.8	2345.9	2506.3	2712.7	2746.2	2965.7
11	750.3	1459.1	1913.8	2580.5	2756.9	2984.0	3020.8	3262.2
12	818.5	1591.7	2087.7	2815.1	3007.5	3255.2	3295.5	3558.8
13	886.7	1724.4	2261.7	3049.6	3258.2	3526.5	3570.1	3855.4
14	955.0	1857.0	2435.7	3284.2	3508.8	3797.8	3844.7	4151.9
15	1023.2	1989.7	2609.7	3518.8	3759.4	4069.1	4119.3	4448.5
16	1091.4	2122.3	2783.6	3753.4	4010.0	4340.3	4394.0	4745.1

Table 6.7: FPS Matrix of SDRAM Version when Running SSD-MobileNetV2

B \ P_{PE}	8	16	24	32	40	48	56	64
1	68.2	132.6	174.0	234.6	250.6	271.3	274.6	296.6
2	136.4	260.2	340.5	442.9	474.3	516.2	526.2	568.8
3	202.6	378.4	488.2	612.3	648.9	697.0	709.1	759.3
4	266.5	487.8	623.3	745.2	785.7	835.9	846.5	899.5
5	328.2	585.0	734.2	856.8	895.1	947.0	957.8	1011.3
6	387.2	669.0	824.4	947.7	986.7	1038.7	1040.9	1089.5
7	443.9	735.9	888.8	1009.2	1045.9	1083.6	1090.6	1130.5
8	498.8	795.7	940.9	1053.4	1081.1	1116.1	1126.5	1157.7
9	551.7	849.2	984.7	1090.1	1110.0	1138.2	1147.8	1175.6
10	597.6	895.3	1022.9	1115.5	1134.3	1156.1	1165.0	1184.8
11	641.2	934.7	1056.3	1136.6	1154.3	1171.2	1176.1	1192.5
12	682.6	970.3	1085.3	1152.2	1169.1	1184.1	1185.6	1198.9
13	717.4	998.0	1104.8	1163.4	1179.3	1188.9	1190.2	1202.6
14	750.1	1023.1	1122.1	1173.1	1188.2	1193.0	1194.2	1205.7
15	781.0	1045.7	1137.5	1181.7	1193.5	1196.5	1197.7	1208.4
16	810.1	1066.3	1151.3	1189.2	1198.1	1199.6	1200.7	1210.2

Table 6.8: FPS Matrix of URAM Version when Running SSDlite-MobileNetV2

B \ P_{PE}	8	16	24	32	40	48	56	64
1	72.4	140.8	186.2	248.6	268.8	291.7	297.9	311.7
2	144.9	281.6	372.5	497.2	537.6	583.4	595.8	623.3
3	217.3	422.4	558.7	745.8	806.4	875.0	893.6	935.0
4	289.8	563.2	744.9	994.4	1075.2	1166.7	1191.5	1246.6
5	362.2	704.0	931.1	1243.0	1344.0	1458.4	1489.4	1558.3
6	434.6	844.8	1117.4	1491.7	1612.8	1750.1	1787.3	1869.9
7	507.1	985.6	1303.6	1740.3	1881.5	2041.7	2085.2	2181.6
8	579.5	1126.4	1489.8	1988.9	2150.3	2333.4	2383.1	2493.2
9	652.0	1267.2	1676.1	2237.5	2419.1	2625.1	2680.9	2804.9
10	724.4	1408.0	1862.3	2486.1	2687.9	2916.8	2978.8	3116.5
11	796.9	1548.8	2048.5	2734.7	2956.7	3208.4	3276.7	3428.2
12	869.3	1689.6	2234.8	2983.3	3225.5	3500.1	3574.6	3739.8
13	941.7	1830.4	2421.0	3231.9	3494.3	3791.8	3872.5	4051.5
14	1014.2	1971.2	2607.2	3480.5	3763.1	4083.5	4170.3	4363.1
15	1086.6	2112.0	2793.4	3729.1	4031.9	4375.2	4468.2	4674.8
16	1159.1	2252.8	2979.7	3977.8	4300.7	4666.8	4766.1	4986.4

Table 6.9: FPS Matrix of SDRAM Version when Running SSDlite-MobileNetV2

B \ P_{PE}	8	16	24	32	40	48	56	64
1	72.4	140.8	186.2	248.6	268.8	291.7	297.9	311.7
2	144.9	275.9	363.7	470.2	509.9	556.5	573.1	600.4
3	215.0	400.5	519.9	646.6	693.0	745.7	765.4	789.9
4	282.1	515.3	661.6	782.8	833.4	882.1	899.9	924.5
5	347.1	616.3	776.2	891.4	938.3	987.4	1005.3	1029.7
6	409.1	702.8	867.9	977.3	1024.2	1072.4	1080.7	1104.0
7	468.6	770.6	928.0	1032.3	1076.0	1108.8	1121.6	1142.9
8	526.0	830.6	975.0	1070.7	1104.2	1134.3	1150.1	1168.1
9	581.2	881.9	1013.8	1102.2	1127.0	1153.8	1168.3	1184.3
10	628.5	925.1	1047.2	1122.8	1145.9	1169.8	1183.3	1192.0
11	673.4	961.3	1076.2	1140.2	1161.9	1183.2	1192.3	1198.4
12	715.9	993.5	1101.2	1155.0	1175.6	1194.6	1199.9	1203.7
13	751.1	1017.8	1116.9	1165.4	1184.8	1198.1	1202.9	1206.4
14	784.0	1039.6	1130.7	1174.4	1192.8	1201.0	1205.5	1208.7
15	815.0	1059.3	1142.9	1182.4	1197.3	1203.5	1207.7	1210.6
16	844.2	1077.1	1153.7	1189.4	1201.1	1205.6	1209.6	1211.7

computational ratio between the two PE Arrays in the most regular layers. However, in many irregular layers, the 1×1 PE Array remains idle when running a single layer, with the actual optimal parallelism ratio at 8:1, and 9:1 being the next best.

The results for SSDMobileNetV2 and SSDliteMobileNetV2 show similar trends to the MobileNetV2 results used for classification, as the main computational load is still in the backbone network. The throughput for each device is slightly less than half of the classification results.

Chapter 7

Optimization

7.1 Utilizing URAM to Store Weights

The ratio between BRAM and URAM varies for different devices. Some devices have limited BRAM but a large amount of URAM. Therefore, we consider allowing the use of URAM as the weight memory for the PE Array. Compared to BRAM, URAM has a much larger capacity, equal to the sum of eight BRAMs' capacities, but it has lower flexibility and does not support dual-clock operations. Therefore, additional modifications are required to ensure consistent functionality. Additionally, the port width of URAM cannot be modified. For weights, the maximum sum of all DSP weights at one cascade position in a group is only $4 \times 8 = 32$, which cannot fully utilize the capacity of URAM. Therefore, we allow binding multiple groups of PEs together to share one URAM.

7.2 Optimization Problem Description

Porting our accelerator to other devices involves considerable work and requires many parameters to be considered. The proportional relationship between parallelism parameters significantly affects the final performance. For instance, increasing the size of the PE Array does not always result in a proportional increase in throughput, which may lead to suboptimal performance despite using more resources. Based on the details of the accelerator in the previous chapters, we summarize the key parameters in Table. 7.1.

Below are explanations about these parameters:

- $URAM_{mode}$ is not optimized as a target. Because for large-capacity devices, URAM mode always achieves much higher throughput than

Table 7.1: Parameters

Parameter	Type
$URAM_{mode}$	bool
$LITE$	bool
B	int
G	int
DC	int
$DWCCDIV$	int
$SWUMEM$	int
PE_W	int
PE_Q	int
$PEDW_{WQ}$	int
$FCDIV$	int
$MADD_{DSP}$	float
ACC_{DSP}	float
$PE_{WMEM_{BRAM}}$	float
$AMEM_{BRAM}$	bool
SWU_{MEM}	bool
DWC_{MEM}	bool
$PE_{WMEM_{mode}}$	bool

SDRAM mode; for small-capacity devices, SDRAM mode always has higher area efficiency than URAM mode, and devices that are too small may not even accommodate accelerators in URAM mode.

- $LITE$ controls whether the path from PEDW Array to ADATA is enabled. When enabled, it can reduce the minimum weight capacity of the PE Array.
- B represents batch size, ranging from 1 to 16.
- G represents group size, controlling the number of PEs in the PE Array, ranging from 2 to 16, with each 2 being a step. It determines P_{PE} together with DC.
- DC represents the height of DSP cascades.
- $DWCCDIV$ controls the number of PEs in the PEDW Array, where $P_{PEDW} = \frac{DC}{DWCCDIV}$, and its value is a factor of $DC \times 2$.
- $SWUMEM$ controls the number of memories in the SWU, either $K+1$ or $K+2$.
- PE_W controls the depth of weight memory in the PE Array, ranging from 64 to 16,384, with each 64 being a step.
- PE_Q controls the depth of quantization parameter memory in the PE Array, ranging from 64 to 256, with each 64 being a step.
- $PEDW_{WQ}$ controls the depth of weight and quantization parameter memory in the PEDW Array, ranging from 64 to 1,024, with each 64 being a step.
- $FCDIV$ controls how many parts the fully connected layer is divided into, with values of 1, 2, or 4.
- $MADD_{DSP}$ represents the proportion of DSPs used for multiply-accumulate operations in the PE Array.
- ACC_{DSP} represents the proportion of DSPs used for accumulation operations in the PE Array.
- $PE_{WMEM_{BRAM}}$ represents the proportion of weight memory in the PE Array using BRAM, with the remaining part using URAM.
- $AMEM_{BRAM}$ controls whether the AMEM in PE Array 0/1 uses BRAM.
- SWU_{MEM} controls the type of memory in the SWU.
- DWC_{MEM} controls the type of memory used by the channel conversion module in the PEDW Array.
- $PE_{WMEM_{mode}}$ controls the structure of weight memory in the PE Array. When set to MUX, it allows mixing of BRAM and LUTRAM to achieve

higher area efficiency when slightly exceeding the BRAM capacity multiple.

7.3 Optimization Goal

According to the target device's capacity information, our objective is to find the optimal parameters to achieve the highest performance or performance per unit area (efficiency).

7.3.1 Area

Due to the complexity of synthesis tool algorithms, the generated netlist may vary significantly with changes in variable names. Therefore, we use the default synthesis strategy during the evaluation process and enforce hierarchy preservation. Generally, the resource utilization after place-and-route is slightly lower than post-synthesis utilization. Therefore, we use synthesis results as a pessimistic estimate.

Before proceeding, we need to calculate the relationship between SRAM capacity and area. When LUTs in SLICEM are combined into LUTRAMs, only multiples of 64 appear according to the above parameters, so each LUT's effective capacity is fully utilized. When combining simple dual-port SRAMs, with depths in multiples of 64, if using an UltraScale device, each SLICEM contains 8 LUTs, and the simple dual-port mode requires at least one additional LUT per SLICEM. The total number of LUTs used is a power of 2, such as 2, 4, 8. The calculation method is as follows:

$$\begin{cases} \left\lfloor \frac{W}{7} \right\rfloor \times \frac{D}{64} & W \bmod 7 = 0 \\ \left(\left\lfloor \frac{W}{7} \right\rfloor \times 8 + 2^{\lceil log_2(W \bmod 7+1) \rceil} \right) \times \frac{D}{64} & W \bmod 7 \neq 0 \end{cases}, \tag{7.1}$$

where W is the width and D is the depth. For 7-series devices, each SLICEM contains 4 LUTs, with other constraints consistent with UltraScale devices. The calculation method is as follows:

$$\begin{cases} \left\lfloor \frac{W}{3} \right\rfloor \times \frac{D}{64} & W \bmod 3 = 0 \\ \left(\left\lfloor \frac{W}{3} \right\rfloor \times 4 + 2^{\lceil log_2(W \bmod 3+1) \rceil} \right) \times \frac{D}{64} & W \bmod 3 \neq 0 \end{cases}. \tag{7.2}$$

We define this calculation as $LUTRAM_{sdp}(W, D)$.

Table 7.2: $RAMB36_d()$

W	D
$W = 1$	32768
$W = 2$	16384
$3 \leq W \leq 4$	8192
$5 \leq W \leq 9$	4096
$10 \leq W \leq 18$	2048
$19 \leq W \leq 36$	1024
$37 \leq W \leq 72$	512

The relationship between port width W and depth D when using RAMB36 is shown in Table. 7.2.

When $W > 36$, simple dual-port SRAM cannot be formed. We define this calculation as $RAMB36_d(W)$.

Additionally, we need to calculate the relationship between the MUX ratio and the number of LUTs, calculated as follows:

$$LUT4MUX(x) = 2^{\lceil \log_2 \lceil \frac{x}{4} \rceil \rceil} \tag{7.3}$$

7.3.2 PE Array

Let's start by analyzing the PE Array. Due to the large number of configurable parameters, we have thoroughly examined the impact of each variable. Table. 7.3 shows the resource requirements for each part.

$MADD$ represents the resource demand of two multiply-accumulate units when it is implemented with LUT. $MADD_M$ is a correction factor resulting from optimization of the adder when the first stage is fixed at 0 input. ADD indicates additional resource usage due to the need for extra adders because of the bit-width limitations of the 7-series DSP48E1. ACC represents the resources needed for the accumulator. Q denotes the resources required for the control logic in the quantization module, where only DSP48 units are used due to the wide bit-width of quantization calculations. $Ctrl$ represents the control section's resource demand, while $FIFO$ is for the FIFO required to store output payloads. $AMEM$ denotes memory resources for storing input

Table 7.3: Resource Requirements of PE Array

	UltraScale Series		7 Series	
	LUT	FF	LUT	FF
$MADD$	184	120	182	120
$MADD_M$	-38	-4	-36	-4
ADD	0	0	63	48
ACC	87	125	87	125
Q	39	26	39	26
$Ctrl$	213+14	253	213+14	253
$FIFO$	$LUTRAM_{sdp}($ $S_{PE} \times 2$ $\times BW_A,$ $64) + 47$	$S_{PE} \times 2$ $\times BW_A$ $+18$	$LUTRAM_{sdp}($ $S_{PE} \times 2$ $\times BW_A,$ $64) + 74$	$S_{PE} \times 2$ $\times BW_A$ $+18$
$AMEM_{LUTRAM}$	$LUTRAM_{sdp}($ $S_{PE} \times BW_A,$ $320) + 2$ $\times S_{PE} \times$ $BW_A + 4$	$S_{PE} \times$ BW_A $\times 6 + 3$	$LUTRAM_{sdp}($ $S_{PE} \times BW_A,$ $320) + 2$ $\times S_{PE} \times$ $BW_A + 4$	$S_{PE} \times$ BW_A $\times 6 + 3$
$AMEM_{BRAM}$	0	$S_{PE} \times BW_A$	0	$S_{PE} \times BW_A$
SRL	$\sum_{i=3}^{DC}(S_{PE}$ $\times BW_A)$ $+S_{PE} \times BW_A$	$\sum_{i=0}^{3}((i+1)$ $\times S_{PE}$ $\times BW_A)+$ $\sum_{i=3}^{DC}(S_{PE}$ $\times BW_A \times 2)+$ $(DC-1)$	$\sum_{i=3}^{DC}(S_{PE}$ $\times BW_A)$ $+S_{PE} \times BW_A$	$\sum_{i=0}^{3}((i+1)$ $\times S_{PE}$ $\times BW_A)+$ $\sum_{i=3}^{DC}(S_{PE}$ $\times BW_A \times 2)+$ $(DC-1)$

feature maps, configurable as either LUTRAM or BRAM. SRL represents the shift registers used to offset the input feature map. Here, BW_A and BW_W are 8 bits, BW_{ACC} is 24 bits, and BW_Q is 18 bits.

Thus, the required number of LUTs is:

$$
\begin{aligned}
&\lfloor P_{PE} \times (1 - MADD_{DSP}) \rfloor \times (LUT_{MADD} \times DC + LUT_{MADD_M}) && \text{For } MADD \\
&+ \lceil P_{PE} \times MADD_{DSP} \rceil && \text{For DSP Packing Correction} \\
&+ \lceil P_{PE} \times ACC_{DSP} \rceil && \text{For OPMODE of ACC DSP} \\
&+ \lfloor P_{PE} \times (1 - ACC_{DSP}) \rfloor \times LUT_{ACC} && \text{For ACC} \\
&+ (BW_Q + BW_{ACC}) \times S_{PE} \times 2 && \text{For MUX of Quantization Sharing} \\
&+ LUT_Q \times S_{PE} \times 2 && \text{For Quantization} \\
&+ LUT_{FIFO} && \text{For FIFO} \\
&+ LUT_{SRL} \times 2 && \text{For SRL} \\
&+ S_{PE} \times BW_A && \text{For Clock Domain Convert} \\
&+ LUT_{Ctrl} && \text{For Controller}
\end{aligned}
\tag{7.4}
$$

Additionally, if using a 7-series device, we need to add the LUT count for the addition operation in MADD calculations as follows:

$$
\lceil P_{PE} \times MADD_{DSP} \rceil \times (LUT_{ADD} \times (DC - 1)) \,. \tag{7.5}
$$

Moreover, based on the configuration of AMEM, we need to add either $AMEM_{LUTRAM_{LUT}}$ or $AMEM_{BRAM_{LUT}}$.

On the other hand, the required number of FFs is:

$$
\begin{aligned}
&\lfloor P_{PE} \times (1 - MADD_{DSP}) \rfloor \times (FF_{MADD} \times DC + FF_{MADD_M}) && \text{For } MADD \\
&+P_{PE} \times DC \times BW_W && \text{For input W reg} \\
&+P_{PE} \times DC \times (BW_A + 1) \times 2 && \text{For input A reg} \\
&+P_{PE} \times (BW_A \times 4 + 4) \times 2 && \text{For DSP Packing Correction} \\
&+ \lfloor P_{PE} \times (1 - ACC_{DSP}) \rfloor \times FF_{ACC} && \text{For ACC} \\
&+ (BW_Q + BW_{ACC}) \times P_{PE} \times 2 && \text{For MUX of Quantization Sharing} \\
&+FF_Q \times S_{PE} \times 2 && \text{For Quantization} \\
&+FF_{FIFO} && \text{For FIFO} \\
&+FF_{SRL} \times 2 && \text{For SRL} \\
&+S_{PE} \times BW_A \times 3 && \text{For Clock Domain Convert} \\
&+LUT_{Ctrl} && \text{For Controller}
\end{aligned}
\tag{7.6}
$$

Similarly, if using a 7-series device, we also need to add the FF count for the addition operation in MADD calculations:

$$\lceil P_{PE} \times MADD_{DSP} \rceil \times FF_{ADD} \times (DC - 1) \tag{7.7}$$

Likewise, depending on the usage of AMEM, we may need to add the DSP count, adding either $AMEM_{LUTRAM_{FF}}$ or $AMEM_{BRAM_{FF}}$. The required number of DSPs is calculated as follows:

$$\lceil P_{PE} \times MADD_{DSP} \rceil \times DC + \lceil P_{PE} \times ACC_{DSP} \rceil + S_{PE} \times 2. \tag{7.8}$$

The three components of the polynomial are MADD, ACC, and the quantization section. When AMEM uses BRAM, the BRAM usage is 1; otherwise, it is 0.

7.3.3 PEDW Array

Next, we analyze the PEDW Array, where each component's resource requirements are listed in Table. 7.4.

Table 7.4: Resource Requirements of PEDW Array

	UltraScale Series		7 Series	
	LUT	FF	LUT	FF
MUL	148	58	148	58
$ADD25$	422	446	422	446
$ADD9$	151	179	151	179
Q	11	26	11	26
$Ctrl$	113+14	175	113+11	175
$FIFO$	$LUTRAM_{sdp}(\frac{S_{PE}\times 2\times BW_A}{DWCCDIV}, 64)+49$	$\frac{S_{PE}\times 2\times BW_A}{DWCCDIV}$ +18	$LUTRAM_{sdp}(\frac{S_{PE}\times 2\times BW_A}{DWCCDIV}, 64)+74$	$\frac{S_{PE}\times 2\times BW_A}{DWCCDIV}$ +18

MUL represents the resource demand for two multipliers, and ADD indicates the resources needed for the adder tree. Here, two scenarios are considered: when K=5, there are 25 input data; when K=3, there are 9 input data. Q denotes the quantization module's resources, and in the case of the PEDW Array, the quantization is lower because the PE outputs only unsigned fixed-point numbers. $Ctrl$ represents the control section's resources, and $FIFO$ is for storing the output payloads.

Thus, the required number of LUTs is:

$$\begin{aligned} &P_{PEDW}\times(LUT_{MUL}\times K\times K+2\times LUT_{ADD}) && \text{For Madd}\\ &+P_{PEDW}\times 2\times LUT_{Q} && \text{For Quantization}\\ &+LUT_{FIFO} && \text{For FIFO}\\ &+LUT_{Ctrl} && \text{For Controller} \end{aligned} \quad . \tag{7.9}$$

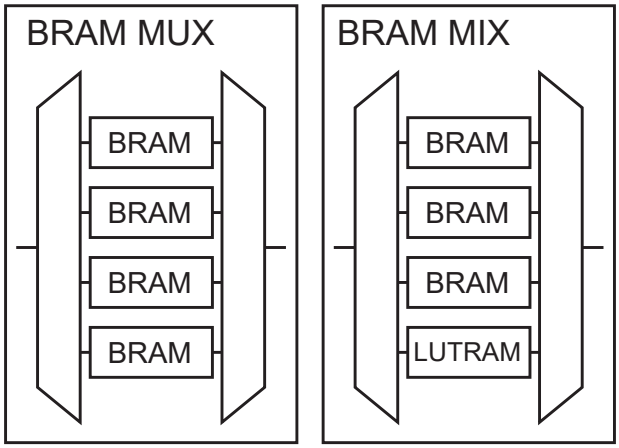


Figure 7.1: Memories with MUX structure.

On the other hand, the required number of FFs is:

$$\begin{aligned} & P_{PEDW} \times (FF_{MUL} \times K \times K + 2 \times FF_{ADD}) && \text{For Madd} \\ & + P_{PEDW} \times 2 \times BW_A \times K \times K \times 2 && \text{For input A reg} \\ & + P_{PEDW} \times 2 \times FF_Q && \text{For Quantization} \\ & + FF_{FIFO} && \text{For FIFO} \\ & + FF_{Ctrl} && \text{For Controller} \end{aligned} . \quad (7.10)$$

The required number of DSP units is $P_{PEDW} \times 2$.

7.3.4 Weight and Quantization Parameter Storage

The next step is weight and quantization parameter memory. There are multiple configurations for weight memory in the PE Array, and the use of MUX mode has a significant impact on the usage of LUTs and FFs. Figure. 7.1 illustrates this method, showing the additional MUX and DEMUX components required. Additionally, when the capacity is not a multiple of BRAM's capacity, a combination of BRAM and LUTRAM can be used.

The required number of LUTs is:

$$\begin{aligned}
&(DC-1)\times B+G && \text{For W}\\
&+LUTRAM_{sdp}\left(BW_Q, PE_Q\right)\times 2\times P_{PE}+\left(BW_Q\times 2+\left\lceil\frac{PE_Q}{64}\right\rceil\right)\times P_{PE}+2\times G && \text{For Q}\\
&+211 && \text{For INIT}
\end{aligned} \tag{7.11}$$

When using the MUX mode, the additional requirement of LUT is:

$$\lceil P_{PE}\times PE_{WMEM_{BRAM}}\rceil\times\left(\left\lceil\frac{PE_W}{RAMB36_d\left(DC\times BW_W\right)}\right\rceil+DC\times BW_W\right). \tag{7.12}$$

If it's not a multiple of the BRAM depth, the additional requirement of LUT is:

$$\begin{aligned}
&\lceil P_{PE}\times PE_{WMEM_{BRAM}}\rceil\times\\
&\left(LUTRAM_{sdp}\left(DC\times BW_W, PE_W \bmod RAMB36_d\left(DC\times BW_W\right)\right)+DC\times BW_W\right).
\end{aligned} \tag{7.13}$$

When mixed with URAM, the additional required quantity of LUT is:

$$\lfloor G\times\left(1-PE_{WMEM_{BRAM}}\right)\rfloor\times S_{PE}. \tag{7.14}$$

On the other hand, the required FF count is:

$$\begin{aligned}
&DC\times P_{PE}\times BW_W+1+14\times DC\times B && \text{For W}\\
&+BW_Q\times\left(\left\lceil\frac{PE_Q}{64}\right\rceil+1\right)\times LUT4MUX\left(\frac{PE_Q}{64}\right)\times 2\times P_{PE} && \text{For Q(1/2)}\\
&+LUT4MUX\left(\frac{PE_Q}{64}\right)\times P_{PE} && \text{For Q(2/2)}\\
&+\left(DC\times DC\times BW_W+DC\times DC\times BW_W\times\frac{BW_Q}{32}\times 2\right)\times 2+162. && \text{For INIT}
\end{aligned} \tag{7.15}$$

When using MUX mode, additional requirements of LUT include:

$$\begin{aligned}
&\lceil P_{PE}\times PE_{WMEM_{BRAM}}\rceil\times\\
&\left(2\times\lceil\log_2\left(PE_W \bmod RAMB36_d\left(DC\times BW_W\right)\right)\rceil+DC\times BW_W\times 3\right).
\end{aligned} \tag{7.16}$$

When mixing with URAM, additional requirements of LUT are:

$$\lfloor G\times\left(1-PE_{WMEM_{BRAM}}\right)\rfloor\times DC\times 2. \tag{7.17}$$

The required BRAM quantity varies depending on the MUX mode:

- MUX mode:

$$\lceil G \times PE_{WMEM_{BRAM}} \rceil \times DC \times \left\lceil DC \times BW_W \times \left\lfloor \frac{PE_W}{RAMB36_d\,(DC \times BW_W)} \right\rfloor \times \frac{RAMB36_d\,(DC \times BW_W)}{36864} \right\rceil . \tag{7.18}$$

- CAS mode:

$$\lceil G \times PE_{WMEM_{BRAM}} \rceil \times DC \times \left\lceil DC \times BW_W \times \left\lceil \frac{PE_W}{RAMB36_d\,(DC \times BW_W)} \right\rceil \times \frac{RAMB36_d\,(DC \times BW_W)}{36864} \right\rceil . \tag{7.19}$$

If using URAM, the required URAM quantity is:

$$\left\lceil \frac{\lfloor G \times (1 - PE_{WMEM_{BRAM}}) \rfloor}{\left\lfloor \frac{72}{DC \times BW_W} \right\rfloor} \right\rceil \times DC \times \left\lceil \frac{PE_W}{4096} \right\rceil . \tag{7.20}$$

The PEDW Array's weight and quantization parameter memories are both composed of LUTRAMs. The required LUT count is:

$$\begin{aligned}
&LUTRAM_{sdp}\,(BW_W \times K \times K, PEDW_{WQ}) \times P_{PEDW} + && \text{For W(1/2)} \\
&BW_W \times K \times K \times LUTMUX\left(\tfrac{PEDW_{WQ}}{64}\right) + \left\lceil \tfrac{P_{PEDW}}{64} \right\rceil \times P_{PEDW} + P_{PEDW} && \text{For W(2/2)} \\
&+LUTRAM_{sdp}\,(BW_Q, PEDW_{WQ}) \times 2 \times P_{PEDW} + && \text{For Q(1/2)} \\
&\left(BW_Q \times LUT4MUX\left(\tfrac{PEDW_{WQ}}{64}\right) \times 2 + \left\lceil \tfrac{PEDW_{WQ}}{64} \right\rceil\right) \times P_{PEDW} && \text{For Q(2/2)} \\
&+219 && \text{For INIT}
\end{aligned} \tag{7.21}$$

The required FF count is:

$$\begin{aligned}
&BW_W \times K \times K \times P_{PEDW} && \text{For W} \\
&+BW_Q \times 2 \times P_{PEDW} && \text{For Q} \\
&+\left(BW_W \times K \times K + BW_{AXI} \times \tfrac{BW_Q}{32} \times 2\right) \times 2 + 149 && \text{For INIT}
\end{aligned} \tag{7.22}$$

7.3.4.1 SWU

The remaining modules require relatively fewer parameter modifications, and we have fitted them accordingly. The SWU module requires the following number of LUTs:

$$\begin{cases} 392 \times P_{PEDW} + (462 \times DC + 1096) + SWUMEM & \text{K=5} \\ 141 \times P_{PEDW} + (307 \times DC + 596) + SWUMEM & \text{K=3} \end{cases}. \tag{7.23}$$

If using BRAM, the additional required LUT count is:

$$\left(DC \times BW_A \times \left(2 + 2 \times LUT4MUX \left(\frac{3072}{RAMB36_d\,(DC \times BW_A)} \right) \right) \right. \\ \left. +3 + \frac{3072}{RAMB36_d\,(DC \times BW_A)} \times 2 \right) \times SWUMEM. \tag{7.24}$$

The required number of FFs is:

$$\begin{cases} 319 \times P_{PEDW} + (135 \times DC + 487) + SWUMEM & \text{K=5} \\ 167 \times P_{PEDW} + (130 \times DC + 397) + SWUMEM & \text{K=3} \end{cases}. \tag{7.25}$$

If using BRAM, the additional required FF count is:

$$(DC \times BW_A \times 7 \times 2 + 40) \times SWUMEM. \tag{7.26}$$

If URAM is used, the required quantity of URAM is equal to SWUMEM. On the other hand, if BRAM is used, the requirement is as follows:

$$SWUMEM \times \\ \left\lceil DC \times BW_A \times \left\lfloor \frac{3072}{RAMB36_d\,(DC \times BW_A)} \right\rfloor \times \frac{RAMB36_d\,(DC \times BW_A)}{36864} \right\rceil. \tag{7.27}$$

The number of DSPs used is fixed at 2.

7.3.4.2 DWC

The number of LUTs required for the DWC (Depth-Wise channel Converter) module is:

$$15 \times P_{PEDW} + (65 \times DC + 400) + 2. \tag{7.28}$$

If using BRAM, the additional required LUT count is:

$$\left(DC \times BW_A \times LUT4MUX\left(\frac{3072}{RAMB36_d\left(DC \times BW_A\right)}\right) \times 2 \right. \\ \left. +3 + \frac{3072}{RAMB36_d\left(DC \times BW_A\right)} \times 2\right) \times 2. \quad (7.29)$$

The required number of FFs is:

$$29 \times P_{PEDW} + (16 \times DC + 325) + 2. \quad (7.30)$$

If using BRAM, the additional required FF count is:

$$(DC \times BW_A \times 6 \times 2 + 40) \times 2. \quad (7.31)$$

If URAM is used, the required quantity of URAM is equal to SWUMEM. On the other hand, if BRAM is used, the requirement is as follows:

$$2 \times \left\lceil DC \times BW_A \times \left\lfloor \frac{3072}{RAMB36_d\left(DC \times BW_A\right)} \right\rfloor \times \frac{RAMB36_d\left(DC \times BW_A\right)}{36864} \right\rceil. \quad (7.32)$$

The number of DSPs used is fixed at 2.

7.3.4.3 GAP and ADD

The number of LUTs required for the GAP module is:

$$90 \times DC + 47 + DC \times 2 \times LUTRAM_{sdp}\left(14, 2^8\right). \quad (7.33)$$

The required number of FFs is:

$$88 \times DC + 41. \quad (7.34)$$

The number of DSPs required is 2.

The number of LUTs required for the ADD module is:

$$104 \times DC + 59. \quad (7.35)$$

The required number of FFs is:

$$69 \times DC + 34. \quad (7.36)$$

The number of DSPs required is 2.

(7.37)

7.3.5 Miscellaneous Modules

To facilitate the explanation of the remaining modules, we first define the total bus width between the modules processing an image as:

$$BW_{BUS} = S_{PE} \times BW_A \times PP. \tag{7.38}$$

The resource requirements for other miscellaneous modules are listed in Table. 7.5. Based on the instance count of each module and the batch size B, we can estimate the resources required for the entire accelerator.

Table 7.5: Resource Requirements of Miscellaneous Modules

	LUT	FF
HsReg	$BW_{BUS} + 3$	$2 \times BW_{BUS} + 2$
Broadcase	5	2
Combine	$BW_{BUS} + 4$	1
MUXn	$BW_{BUS} \times LUT4MUX(n)+$ $LUT4MUX(n) + n$	$\lceil \log_2 n \rceil$
DEMUXn	$LUT4MUX(n) + n$	$\lceil \log_2 n \rceil$

7.4 Cost Function

First, we need to define the capacity of the target device, including LUT, FF, DSP, BRAM, and URAM. This is defined as a hard constraint. However, high utilization of these resources often leads to routing difficulties and a reduced achievable maximum frequency. Therefore, we need an additional constraint to ensure that hardware resource utilization remains within a reasonable range. We referenced Xilinx's recommended values to design soft limits in Table. 7.6. When the solution's area exceeds this soft constraint, a penalty is applied to the throughput to discourage the optimization algorithm from generating solutions with excessive area.

The constraints are set to 80% for LUT, 50% for FF, and 80% each for DSP, BRAM, and URAM. Compared to Xilinx's recommendations, we do not limit the total DSP, BRAM, and URAM usage to below 70% because

Table 7.6: Utilization limits

	Xilinx Guideline	Soft Limit in Exp.
LUT	70%	80%
FF	50%	50%
DSP,BRAM,URAM	80%	80%
DSP+BRAM+URAM	70%	N/A
LUTRAM	30%	N/A
CARRY8	25%	N/A

we aim to utilize dedicated cascade routing resources to reduce the impact of these IO-intensive hardware resources on general routing resources. We set the LUT utilization constraint slightly higher than Xilinx's recommended values because the post-place-and-route utilization is typically lower than the synthesis value, along with effective use of the dedicated routing resources mentioned earlier. We do not constrain LUTRAM and CARRY8 usage, as LUTRAM in medium to large devices is far from reaching this threshold, and it will inevitably exceed this threshold in smaller devices, while CARRY8 will undoubtedly exceed it significantly.

Thus, our cost function is as follows:

$$Cost = \begin{cases} 1_{\times 10^{-8}} & \text{exceed} \\ p \times FPS + (1-p) \times LIM & \text{not exceed} \end{cases}, \tag{7.39}$$

where

$$FPS = \frac{3_{\times 10^{8}}}{CC_{Est}} \times B, \tag{7.40}$$

$$LIM = \frac{FPS}{\frac{LUT}{LUT_{LIM}} + \frac{FF}{FF_{LIM}} + \frac{DSP}{DSP_{LIM}} + \frac{BRAM+8\times URAM}{BRAM_{LIM}+8\times URAM_{LIM}}}. \tag{7.41}$$

If any resource utilization exceeds the soft limit, we need an additional division by $\frac{\text{resource utilization}}{\text{soft limit}}$. Here, BRAM and URAM are calculated together after weighting their actual capacities. This is because URAM quantities are usually low, and using even one can significantly increase utilization, which might cause the optimization algorithm to avoid using URAM entirely. Combining them in the calculation helps to alleviate this issue to some extent.

Additionally, the formula includes a coefficient p, which controls the bias of the optimization objective. When p approaches 1, absolute throughput is prioritized, whereas when p is closer to 0, throughput per unit area is emphasized.

7.5 Optimization Algorithm

We experimented with several optimization algorithms to solve this problem. However, due to constraints like data types, particularly nonlinear parameters such as $DWCCDIV$, as well as conditional branches and higher-order functions, many optimization algorithms face implementation challenges. Therefore, we primarily rely on Simulated Annealing and Genetic Algorithms, both of which allow the cost function to be treated as a black box.

7.5.1 Simulated Annealing

Let's start with Simulated Annealing. Simulated Annealing is a classical heuristic algorithm used for optimization problems. It simulates the annealing process in metallurgy, where decisions to adopt new solutions follow the Metropolis criterion based on the current temperature. When searching for a maximum, the Metropolis criterion uses the following formula to determine the probability of accepting a new solution:

$$\begin{cases} 1 & \Delta > 0 \\ e^{\frac{\Delta}{t}} & \Delta \leq 0 \end{cases}. \tag{7.42}$$

As the formula indicates, the higher the temperature, the greater the probability of accepting a solution worse than the original one. The computation process of Simulated Annealing is illustrated in Figure. 7.2.

The specific steps are as follows:

1. Generate an initial solution and calculate its cost function value.
2. Randomly modify this solution and calculate the cost function value of the modified solution.
3. Compare the two cost function values. If the new solution is better, update it directly; otherwise, decide whether to accept it based on the Metropolis rule.

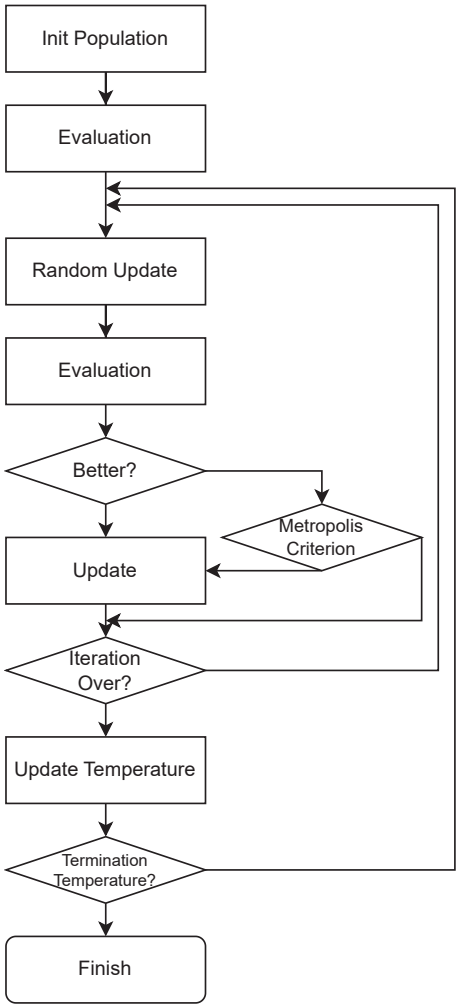


Figure 7.2: Simulated Annealing.

4. Check whether the iteration count at the current temperature has been reached. If not, return to Step 2.
5. If the maximum iteration count at the current temperature has been reached, lower the temperature. Check if the temperature has reached the final temperature. If not, return to Step 2.

We have three main types of parameters: Boolean, integer, and floating-point. Each type generates new solutions in different ways:

- For **Boolean** values, we first generate a random value, then calculate the probability of adopting the new value based on the current tem-

perature. This probability is calculated using the formula $log(\frac{T_{max}}{T})$. $SWUMEM$ is mapped as a Boolean, where $K+1$ represents false and $K+2$ represents true.

- For **integer** parameters, such as memory depths PE_W, PE_Q, and $PEDW_{WQ}$, we first generate a Gaussian-distributed random number with a mean of 0 and a variance of 1. Then, we multiply this value by a scaling factor calculated based on the current temperature, which is $log_{10}(T)$. The resulting value is rounded to the nearest integer, multiplied by the parameter's minimum interval (in this case, 64), and added to the original value. If this value is out of range, it is clipped to stay within the range.
- For parameters B and DC, a Gaussian-distributed random number with a variance of 0.2 is generated. The remaining steps are similar to the above. For parameter G, the variance of the Gaussian distribution is 0.1, and the rounded value is multiplied by the minimum interval (in this case, 2).
- A special case is $DWCCDIV$: first, we generate a list of factors. If DC changes, a value is randomly selected from this list. If DC remains unchanged, the value is randomly adopted based on a probability similar to the Boolean process.
- For **floating-point** values, we generate a random number with a mean of 0 and a variance of 0.1. After multiplying this value by the temperature scaling factor and clipping, it is added to the original value.

7.5.2 Genetic Algorithm

Next is the Genetic Algorithm, another classic general-purpose heuristic algorithm inspired by biological evolution. Compared to Simulated Annealing, Genetic Algorithms come with some additional constraints. All parameters need to be represented in binary, which makes encoding certain parameters more complex:

- **Boolean** variables can be directly represented with binary values.
- **Integer** variables like PE_W, PE_Q, and $PEDW_{WQ}$ have specific ranges and step sizes, so directly using their binary values may leave some values unreachable. To address this, we create a mapping:

$$y = clamp\left(MIN + x \times \Delta, MIN, MAX\right). \tag{7.43}$$

The range of acceptable values x is rounded up to the nearest power of 2, so it slightly exceeds the maximum.

- For $DWCCDIV$, when the maximum value of DC is set to 4, the values range from 1, 2, 4, to 8, resulting in four possible values. We encode this as 2 bits, representing the position of the parameter value in the list.
- For other parameters that need to vary, like $FCDIV$ and G, we take the original values and convert them during decoding to the actual required values.
- **Floating-point** values use a fixed quantization adjustment. These parameters are finally encoded as a 75-bit gene in the Genetic Algorithm.

Figure. 7.3 illustrates the process of the Genetic Algorithm. Initially, a number of individuals are generated with randomly created genes. These individuals are then evaluated based on their cost function values, and each individual is assigned a selection probability. Here, rather than a traditional linear probability distribution, we want the better results to have a higher likelihood of selection. Therefore, we use a `softmax` function to handle their probability distribution, calculated as follows:

$$y = \frac{e^x}{\sum e^{x_i}}. \tag{7.44}$$

Traditional `softmax` works well when processing data with a mean of 0 and a variance of 1, using e as the base. However, in our case, values are scaled to a sequence in the range $[0, 1]$. Using e as the base may not be optimal, so we allow different bases for `softmax`.

The main loop then begins. First, a random subset of individuals equal to the total population size is selected based on the calculated probabilities. Next, based on crossover probability, some individuals undergo crossover. There are several crossover methods available; we choose single-point crossover and uniform crossover. In **single-point** crossover, a random crossover point is selected within the gene, where one offspring inherits the first half of the gene from one parent and the second half from the other, while the remaining genes form another offspring. In **uniform** crossover, bit positions are randomly selected for swapping, with each bit having an equal chance of being chosen.

Then, the new population undergoes mutation, where random gene positions experience bit-flipping. Afterward, all individuals in the new population

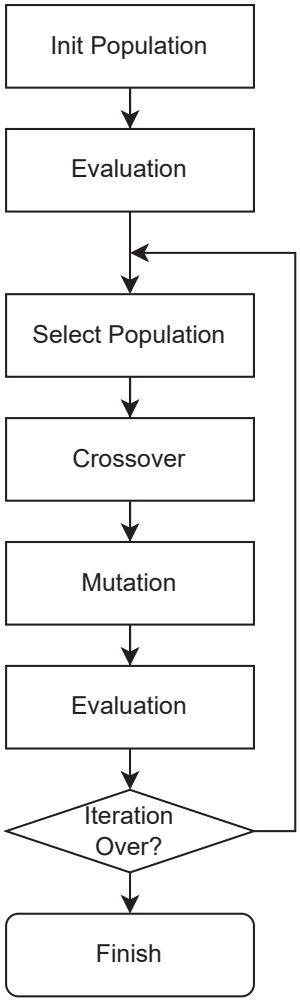


Figure 7.3: Genetic Algorithm.

are evaluated. If the iterations are not yet complete, the process returns to the loop entrance to simulate the next generation.

7.6 Experiment

We designed experiments with various parameter combinations, running each scenario 128 times to eliminate randomness and analyze the probability of generating high-quality solutions.

For Simulated Annealing, we adopted two scenarios: standard Simulated

Annealing (marked as SA), where all parameters are randomly updated at each step, and specialized Simulated Annealing (marked as SA1), where only one parameter is updated randomly per iteration. Besides the final output of the algorithm, we also recorded the highest cost function value during the computation (marked as Hi-SA and Hi-SA1). The initial temperature was set to 10,000, the final temperature to 1, and the temperature update factor to 0.95, with each temperature cycle containing 1,000 iterations, totaling 180k operations.

Table 7.7: Scenarios of Genetic Algorithm

Scenario	Pop.	Gen.	`softmax` Base	Crossover	Calculation
GAp20i10	20	10k	-	One	200k
GAp20i10s1	20	10k	10	One	200k
GAp20i10s2	20	10k	100	One	200k
GAp20i10s3	20	10k	1000	One	200k
GAp20i10s4	20	10k	10000	One	200k
GAp100i2	100	2k	-	One	200k
GAp100i2s1	100	2k	10	One	200k
GAp100i2s2	100	2k	100	One	200k
GAp100i2s3	100	2k	1000	One	200k
GAp100i2s4	100	2k	10000	One	200k
GAp100i5	100	5k	-	One	500k
GAp100i5s1	100	5k	10	One	500k
GAp100i5s2	100	5k	100	One	500k
GAp100i5s3	100	5k	1000	One	500k
GAp100i5s4	100	5k	10000	One	500k
GAp20i10u	20	10k	-	Uniform	200k
GAp20i10s1u	20	10k	10	Uniform	200k
GAp20i10s2u	20	10k	100	Uniform	200k
GAp20i10s3u	20	10k	1000	Uniform	200k
GAp20i10s4u	20	10k	10000	Uniform	200k

In the Genetic Algorithm (marked as GA) experiments, due to the large number of parameters and their significant impact, we configured a series of comprehensive scenarios as shown in Table. 7.7.

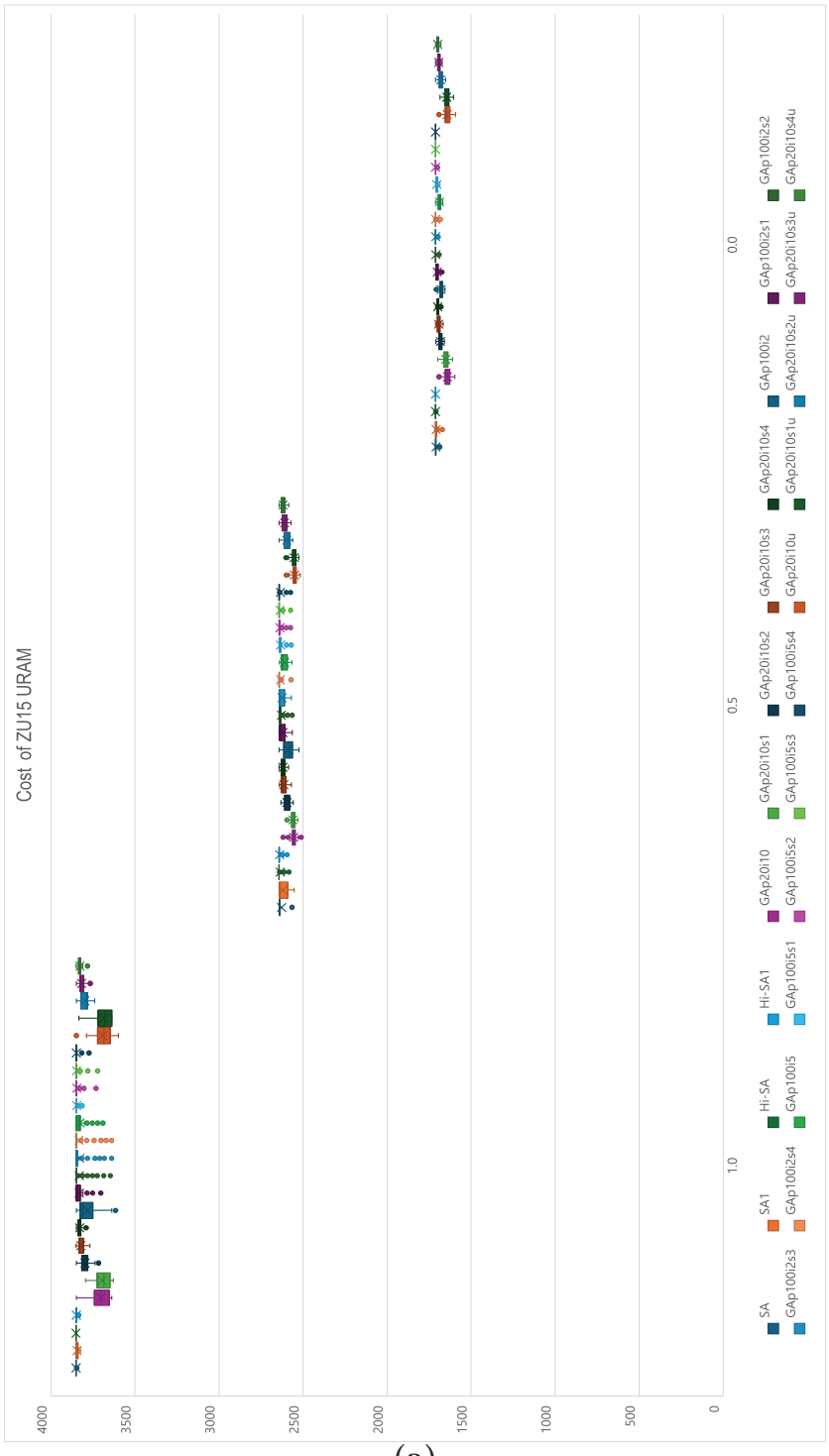
The population and generation settings consisted of three configurations,

with two having computational loads similar to Simulated Annealing: p20i10 (smaller population, more generations) and p100i2 (larger population, fewer generations). The third configuration, p100i5, had a significantly higher computational load. The `softmax` base had four options, ranging from 10^1 to 10^4, with an additional version that didn't use `softmax` for comparison. Scenarios were also set for uniform crossover. The crossover probability was set to 0.8 and the mutation probability to 0.1.

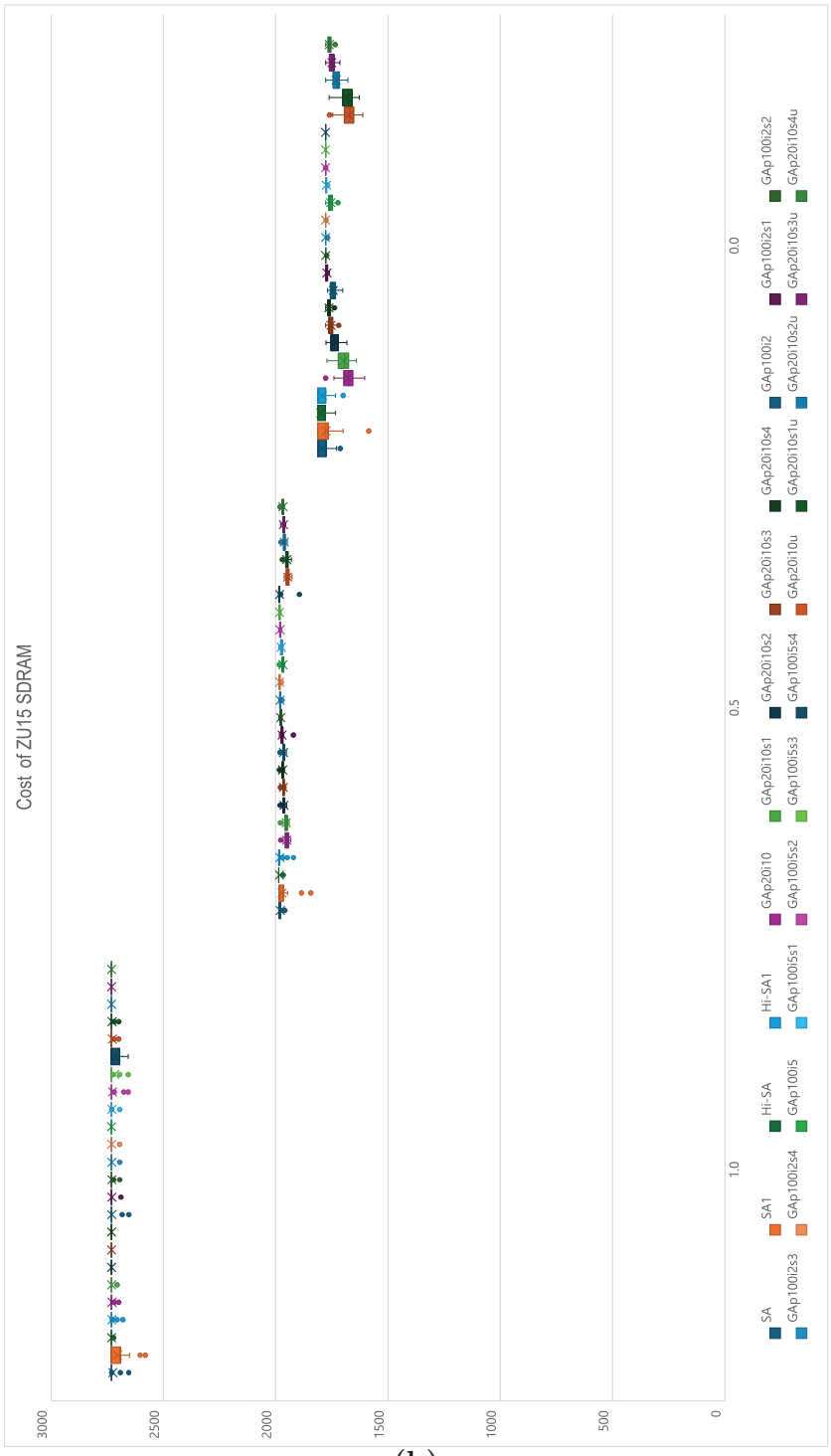
Table 7.8: FPGA List

	LUT(k)	FF(k)	BRAM	URAM	RAM	DSP
ZU1	37	74	108		108	216
ZU2	47	84	150		150	240
ZU3	71	141	216		216	360
ZU3T	72	144	146	50	546	576
ZU4	88	176	128	48	512	728
ZU5	117	234	144	64	656	1248
ZU6	217	429	714		714	1973
ZU7	230	461	312	96	1080	1728
ZU9	274	548	912		912	2520
ZU11	299	597	600	80	1240	2928
ZU15	341	682	744	112	1640	3528
ZU17	423	847	796	102	1612	1590
ZU19	523	1045	984	128	2008	1968
VU13P(1SLR)	432	864	672	320	3232	3072

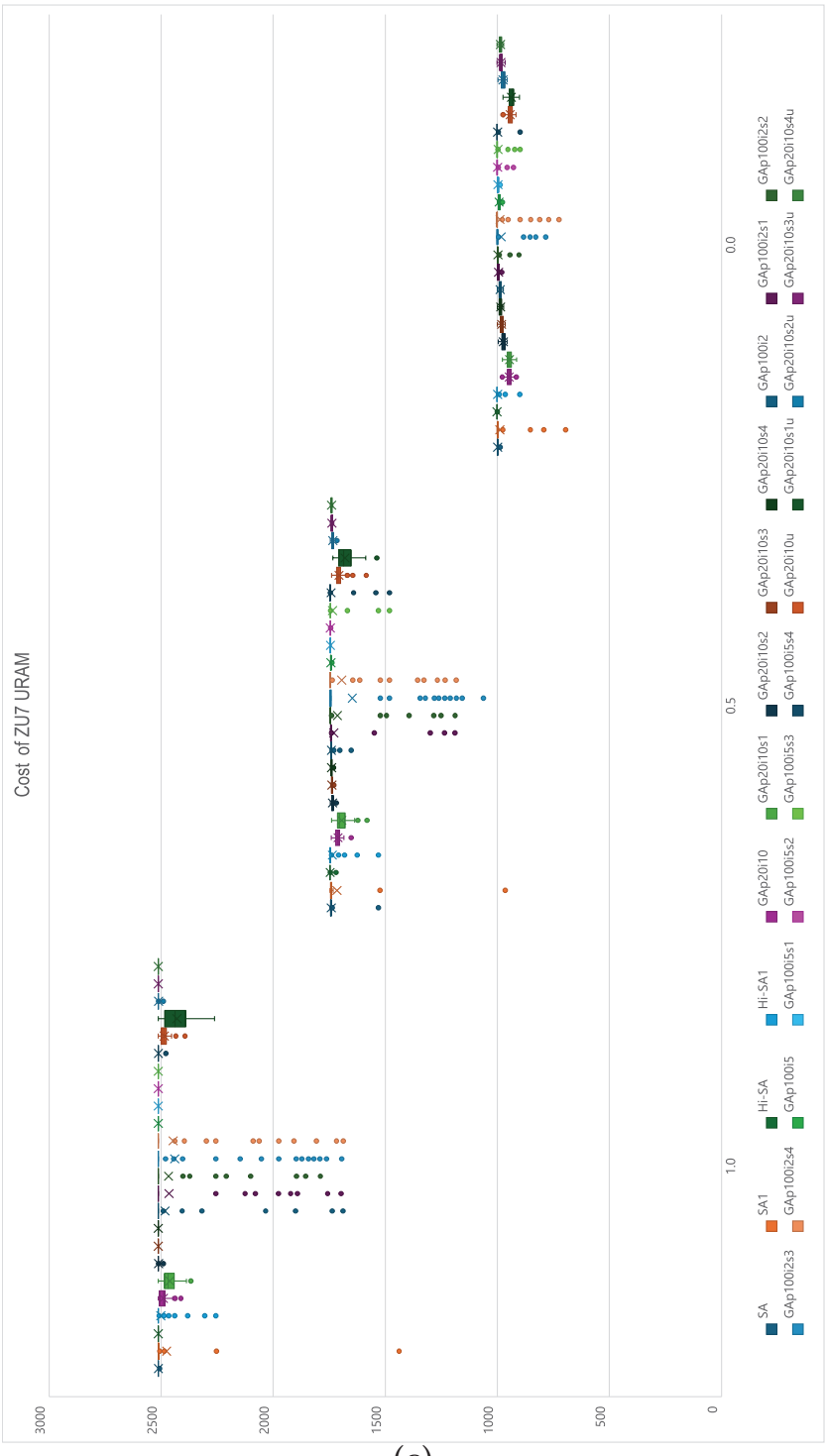
We selected FPGA targets with different capacities, including ZU15, ZU7, ZU5, and ZU3, as listed in Table. 7.8. These FPGAs span various device sizes, with a preference for devices containing URAM to enable SDRAM and URAM versions for comparison.


Cost of ZU15 URAM
4000
3500
3000
2500
2000
1500
1000
500
0
1.0
0.5
0.0
SA
SA1
Hi-SA
Hi-SA1
GAp20i10
GAp20i10s1
GAp20i10s2
GAp20i10s3
GAp20i10s4
GAp20i10u
GAp20i10s1u
GAp20i10s2u
GAp20i10s3u
GAp20i10s4u
GAp100i2
GAp100i2s1
GAp100i2s2
GAp100i2s3
GAp100i2s4
GAp100i5
GAp100i5s1
GAp100i5s2
GAp100i5s3
GAp100i5s4

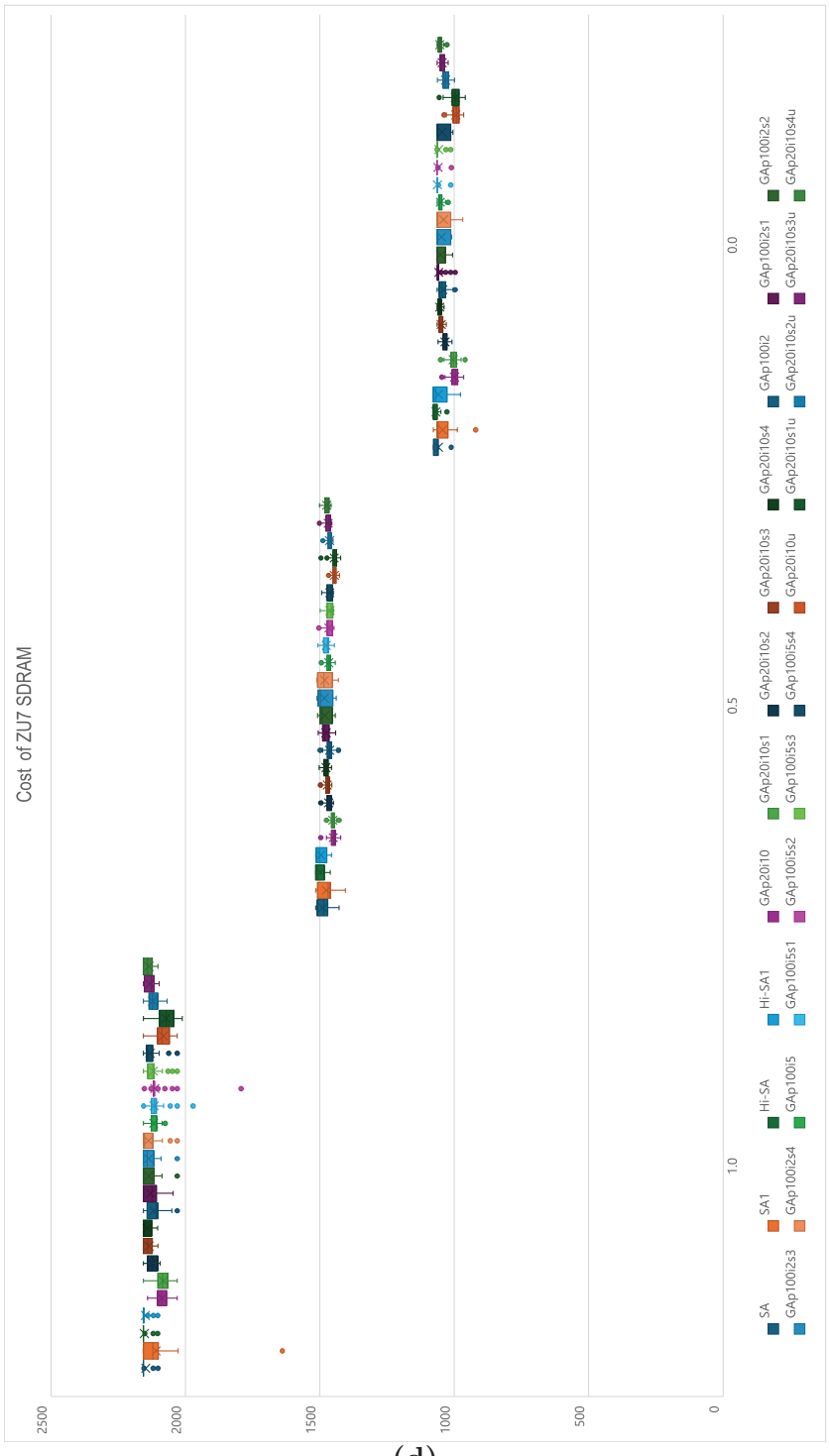
(a)


Cost of ZU15 SDRAM
3000
2500
2000
1500
1000
500
0
1.0
0.5
0.0
SA
SA1
Hi-SA
Hi-SA1
GAp20i10
GAp20i10s1
GAp20i10s2
GAp20i10s3
GAp20i10u
GAp20i10s4
GAp20i10s1u
GAp20i10s2u
GAp20i10s3u
GAp20i10s4u
GAp100i2s3
GAp100i2s4
GAp100i5
GAp100i5s1
GAp100i5s2
GAp100i5s3
GAp100i5s4
GAp100i2
GAp100i2s1
GAp100i2s2

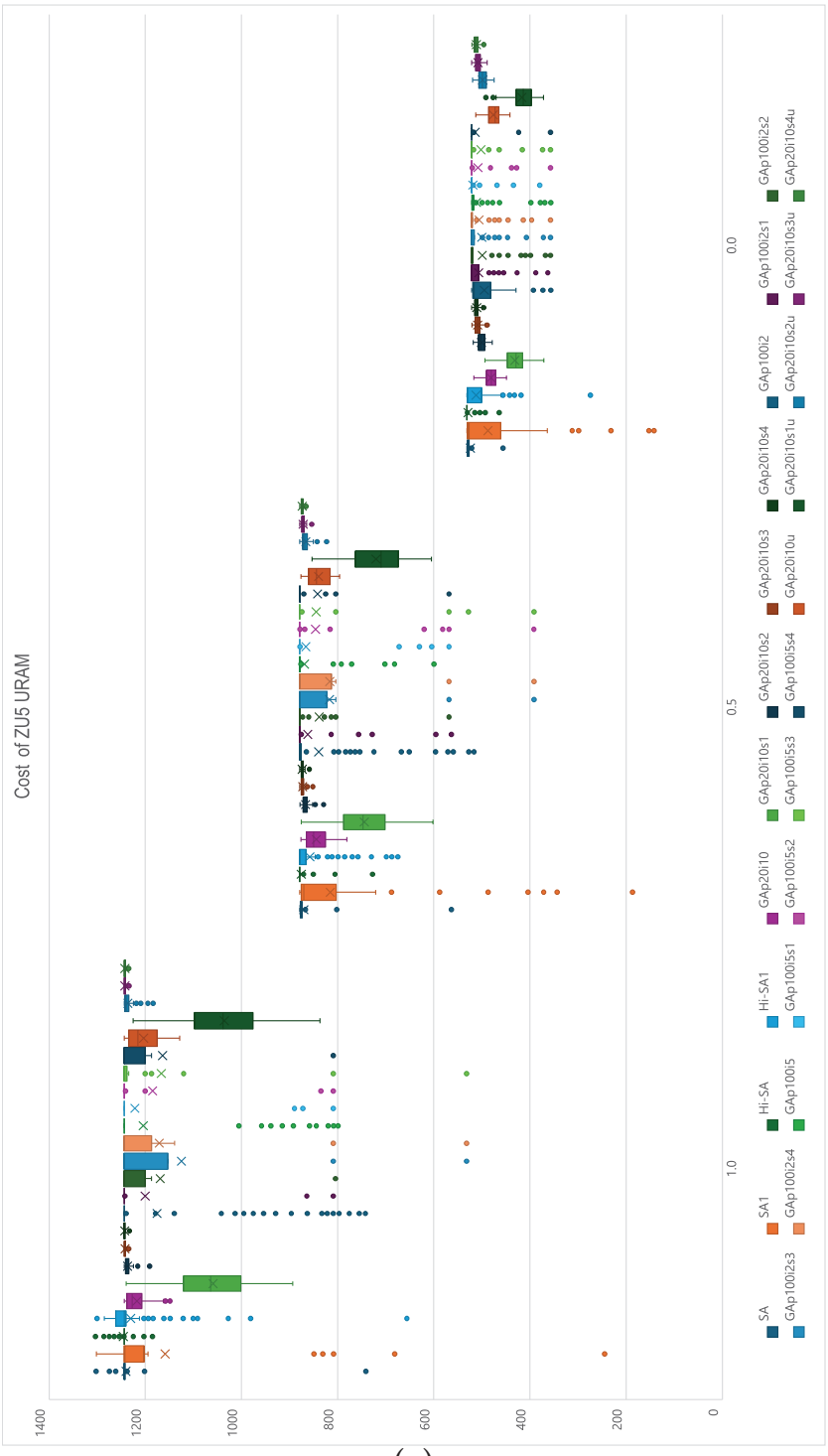
(b)


Cost of ZU7 URAM
3000
2500
2000
1500
1000
500
0
1.0
0.5
0.0
SA
SA1
Hi-SA
Hi-SA1
GAp20l10
GAp20l10s1
GAp20l10s2
GAp20l10s3
GAp20l10s4
GAp20l10u
GAp20l10s1u
GAp20l10s2u
GAp20l10s3u
GAp20l10s4u
GAp100i2s3
GAp100i2s4
GAp100i5
GAp100i5s1
GAp100i5s2
GAp100i5s3
GAp100i5s4
GAp100i2
GAp100i2s1
GAp100i2s2

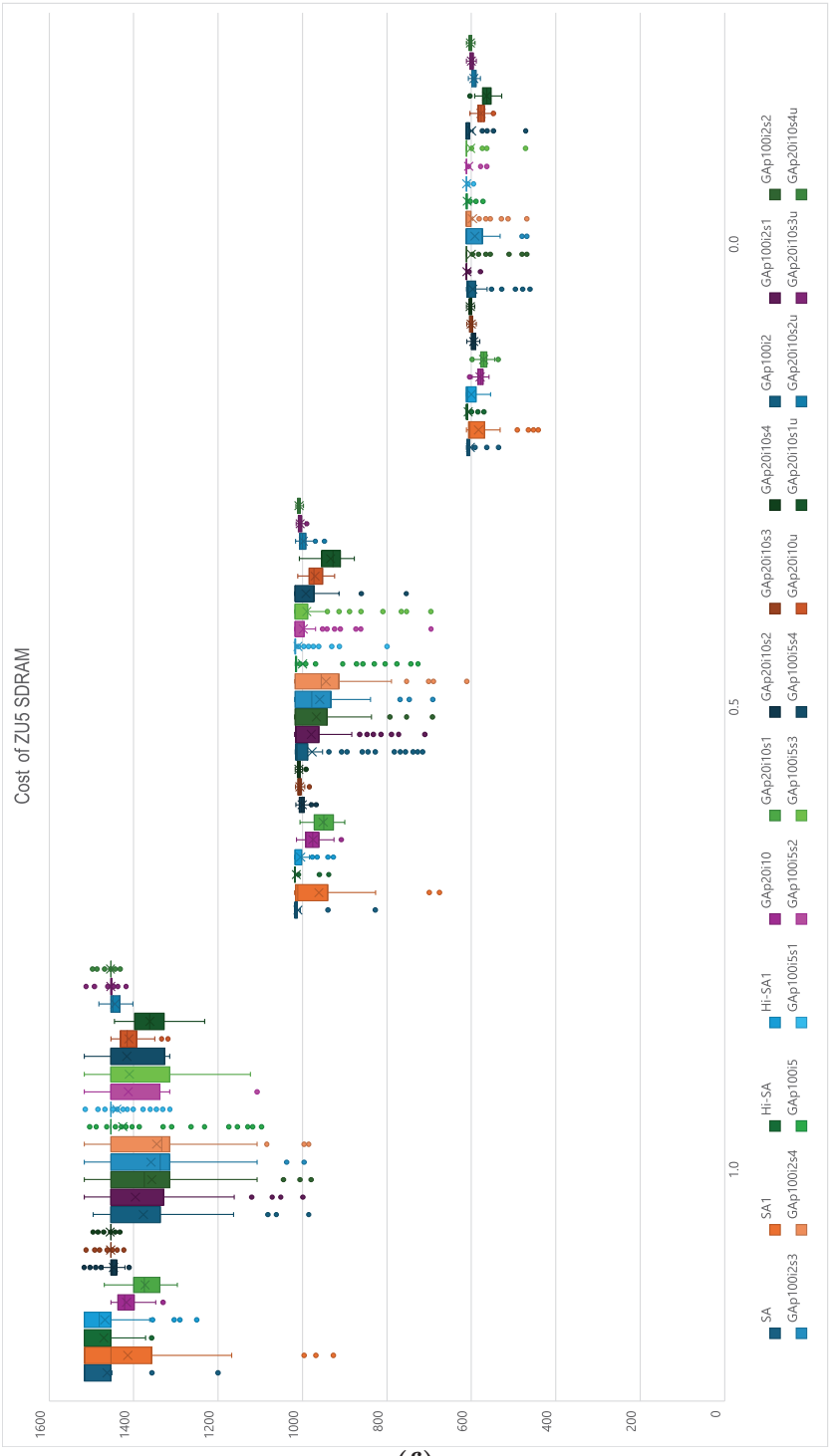
(c)


Cost of ZU7 SDRAM
2500
2000
1500
1000
500
0
1.0
0.5
0.0
SA
SA1
Hi-SA
Hi-SA1
GAp20i10
GAp20i10s1
GAp20i10s2
GAp20i10s3
GAp20i10s4
GAp20i10u
GAp20i10s1u
GAp20i10s2u
GAp20i10s3u
GAp20i10s4u
GAp100i2s3
GAp100i2s4
GAp100i5
GAp100i5s1
GAp100i5s2
GAp100i5s3
GAp100i5s4
GAp100i2
GAp100i2s1
GAp100i2s2

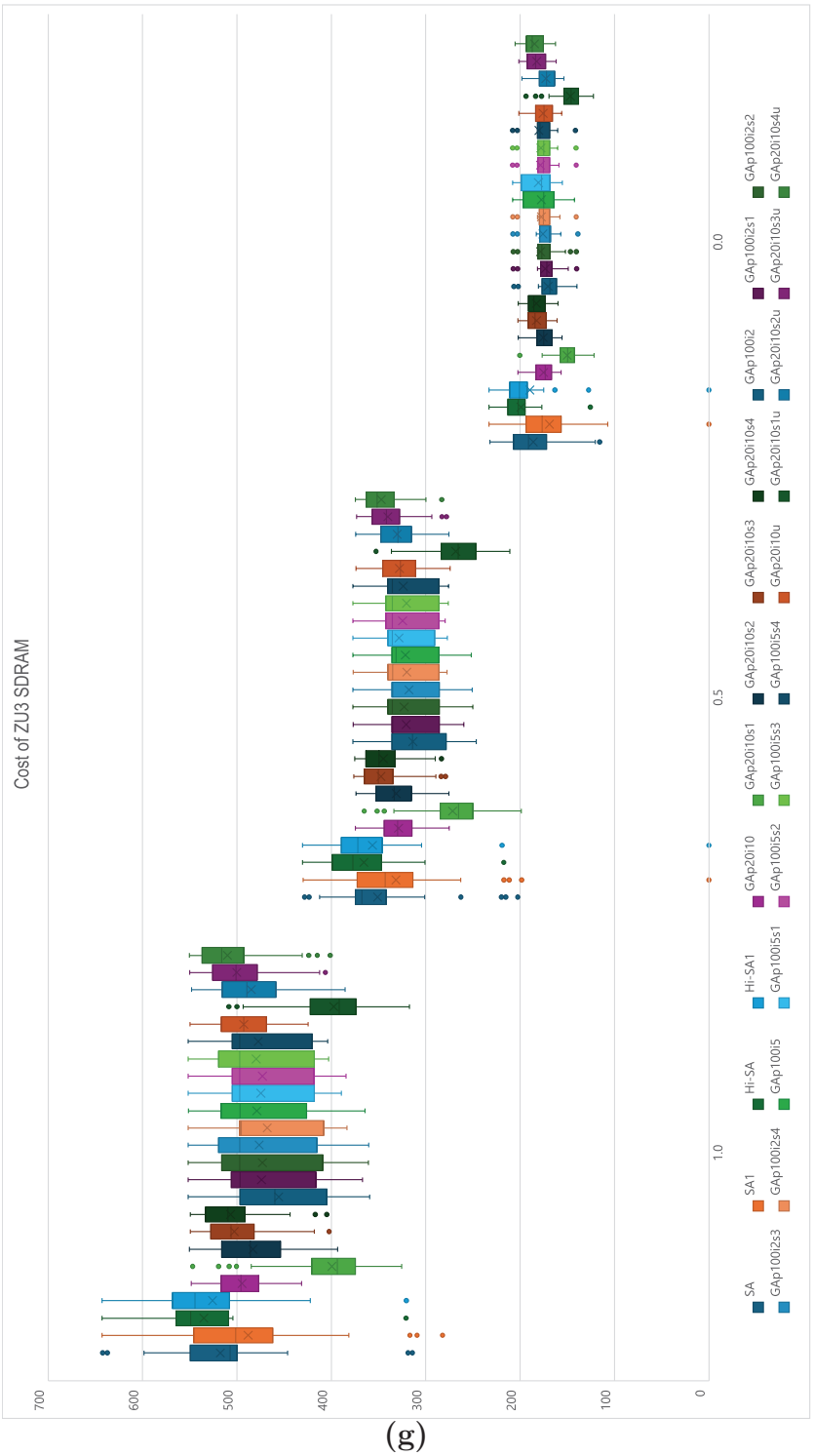
(d)


Cost of ZU5 URAM
1400
1200
1000
800
600
400
200
0
1.0
0.5
0.0
SA
SA1
Hi-SA
Hi-SA1
GAp20l10
GAp20l10s1
GAp20l10s2
GAp20l10s3
GAp20l10s4
GAp20l10u
GAp100i2s3
GAp100i2s4
GAp100i5
GAp100i5s1
GAp100i5s2
GAp100i5s3
GAp100i5s4
GAp20l10s1u
GAp20l10s2u
GAp20l10s3u
GAp20l10s4u
GAp100i2
GAp100i2s1
GAp100i2s2

(e)


Cost of ZU5 SDRAM
1600
1400
1200
1000
800
600
400
200
0
1.0
0.5
0.0
SA
SA1
Hi-SA
Hi-SA1
GAp20i10
GAp20i10u
GAp20i10s1
GAp20i10s2
GAp20i10s3
GAp20i10s4
GAp20i10s1u
GAp20i10s2u
GAp20i10s3u
GAp20i10s4u
GAp100i2s3
GAp100i2s4
GAp100i5
GAp100i5s1
GAp100i5s2
GAp100i5s3
GAp100i5s4
GAp100i2
GAp100i2s1
GAp100i2s2

(f)



(g)

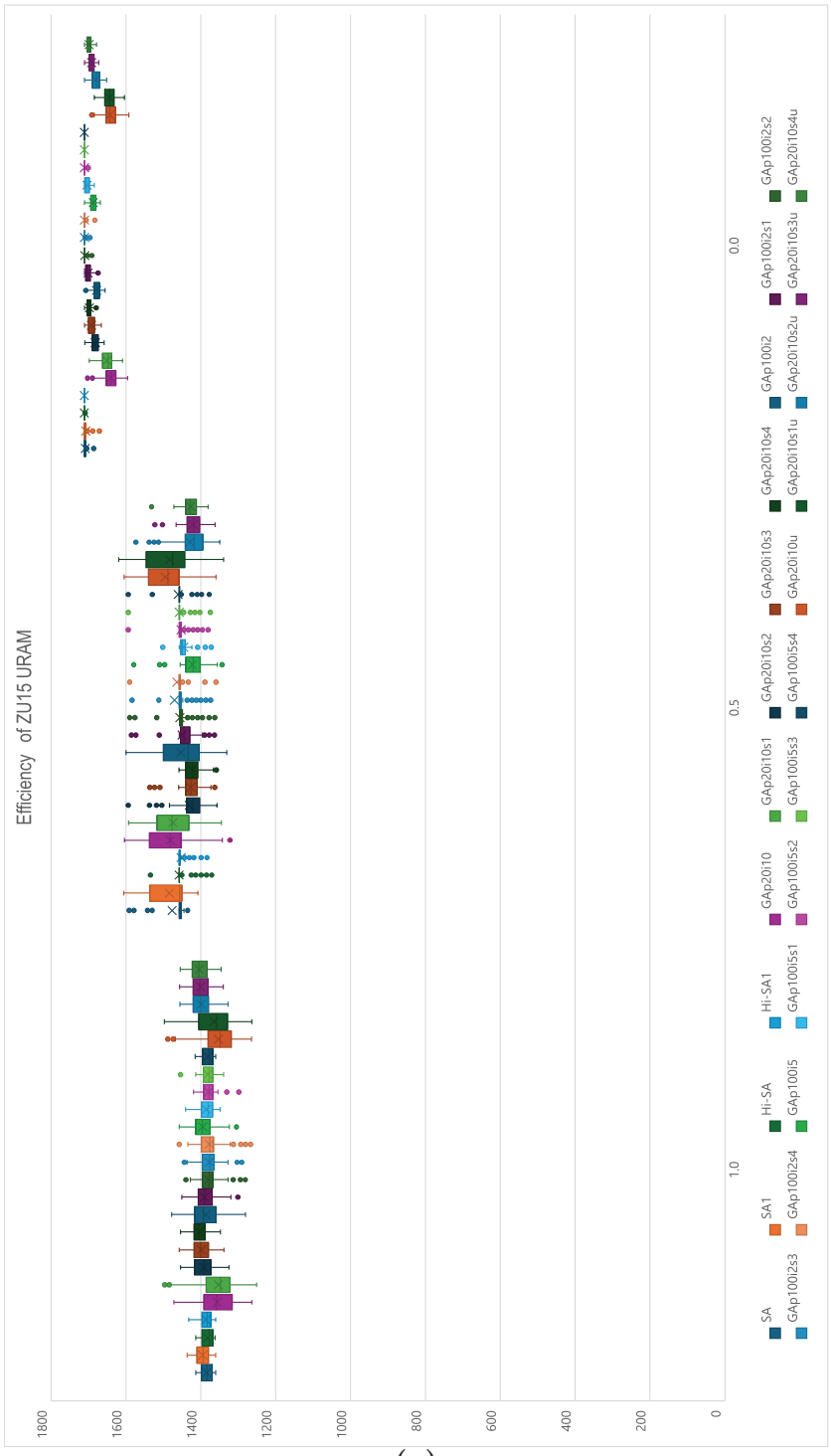
Figure 7.4: Cost function distribution

Figure. 7.4 shows the distribution of 128 cost function values for each experimental scenario. These are presented as box plots, where each box represents the distribution of results from the 1st to 3rd quartiles. The "×" inside the box indicates the mean, and the horizontal line represents the median. The lines extending upward or downward indicate values within 1.5 times the box length from the upper or lower quartile. Outliers beyond this range are shown as points. The series in the figure is divided into three sections, with the horizontal axis showing the value of p.

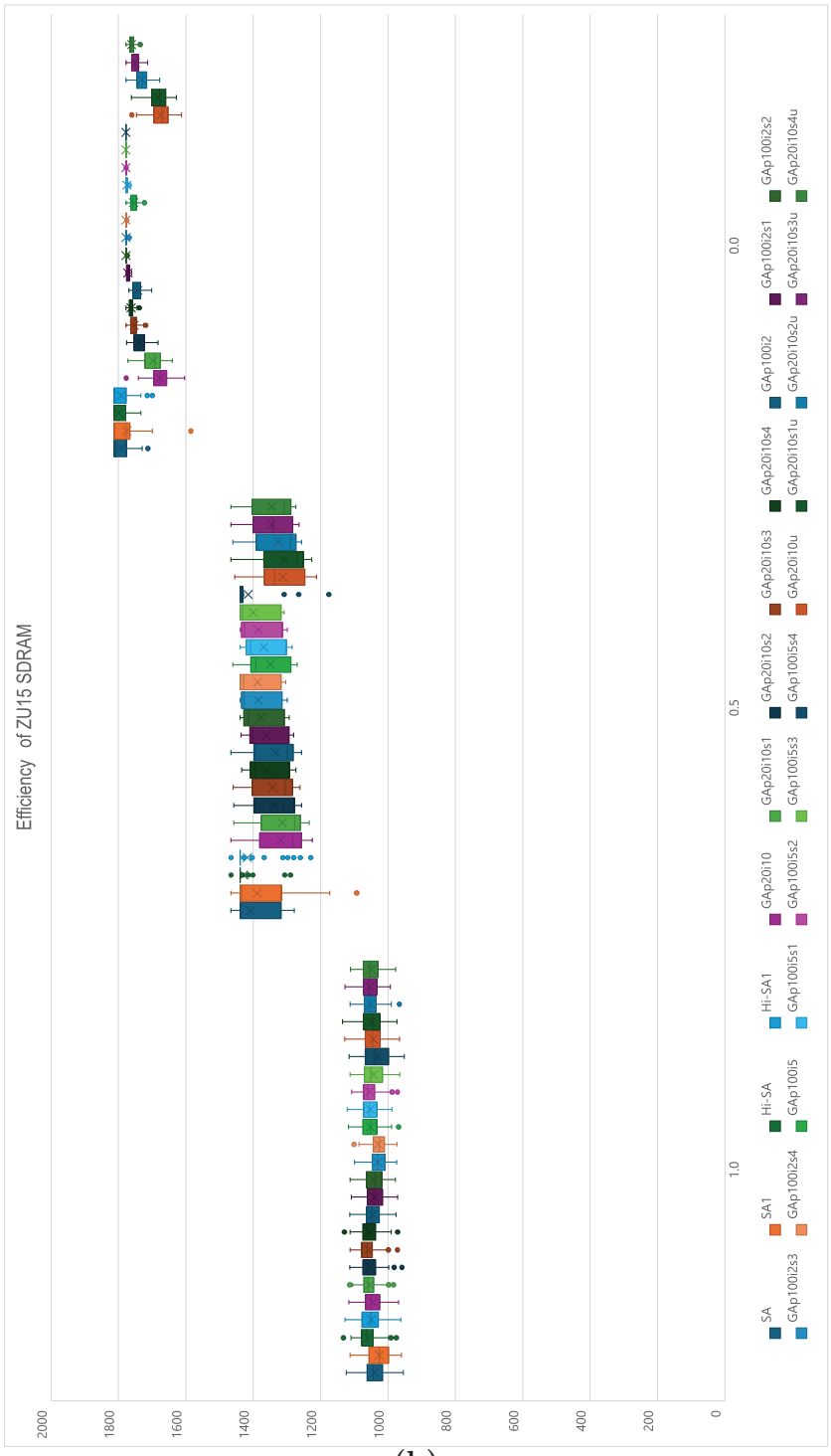
As the efficiency ratio increases, the cost function values tend to decrease since throughput per area (efficiency) is lower than throughput. The highest cost function values found by the two algorithms are close. From Figure. 7.4, we can observe that Simulated Annealing consistently finds the optimal solution, with SA generally more stable than SA1 in most cases. In Figure. 7.4(e), Hi-SA1 slightly outperforms Hi-SA. When computational complexity is similar, Simulated Annealing typically achieves better and more stable solutions than the Genetic Algorithm.

For the Genetic Algorithm, with equivalent computational complexity, results with smaller populations and more generations are more stable, though slightly lower in quality compared to larger populations with fewer generations. In most cases, higher `softmax` base values result in better solution quality. However, on ZU5, the results were completely opposite. Additionally, in some cases, using a base of 10^4 led to minimal improvement over 10^3, as the larger base reduced the search space. In most cases, using a base of 10 yielded worse results than not using `softmax` at all. Increasing the Genetic Algorithm's computational complexity allows it to reach performance similar to Simulated Annealing. On smaller devices, uniform crossover shows significant improvement over single-point crossover, though no significant difference is observed on larger devices.

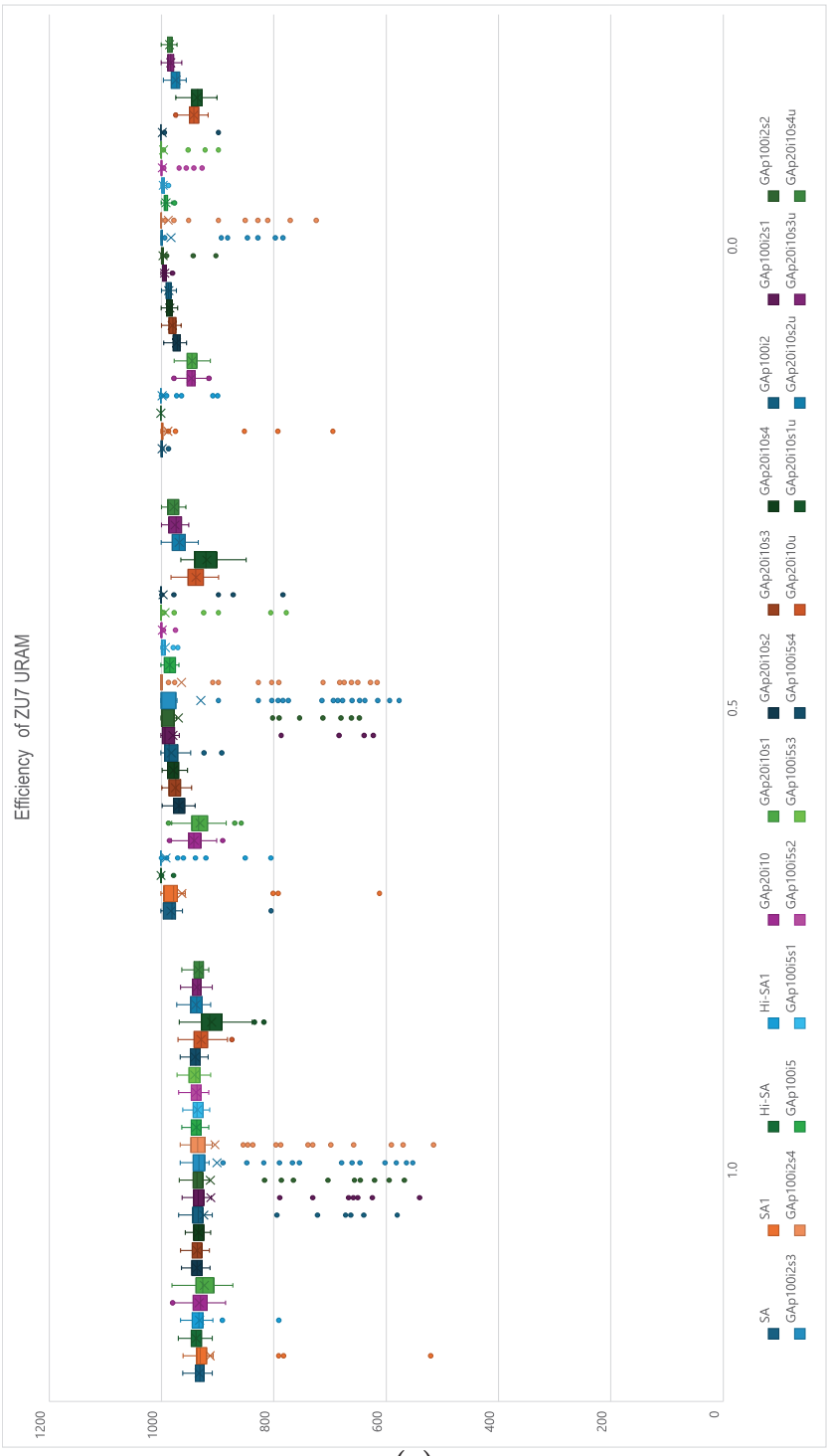
Detailed efficiency and throughput distributions are shown in Figures. 7.5 and 7.6.


Efficiency of ZU15 URAM
1800
1600
1400
1200
1000
800
600
400
200
0
1.0
0.5
0.0
SA
SA1
Hi-SA
Hi-SA1
GAp20i10
GAp20i10s1
GAp20i10s2
GAp20i10s3
GAp20i10u
GAp20i10s4
GAp20i10s1u
GAp20i10s2u
GAp20i10s3u
GAp20i10s4u
GAp100i2
GAp100i2s1
GAp100i2s2
GAp100i2s3
GAp100i2s4
GAp100i5
GAp100i5s1
GAp100i5s2
GAp100i5s3
GAp100i5s4

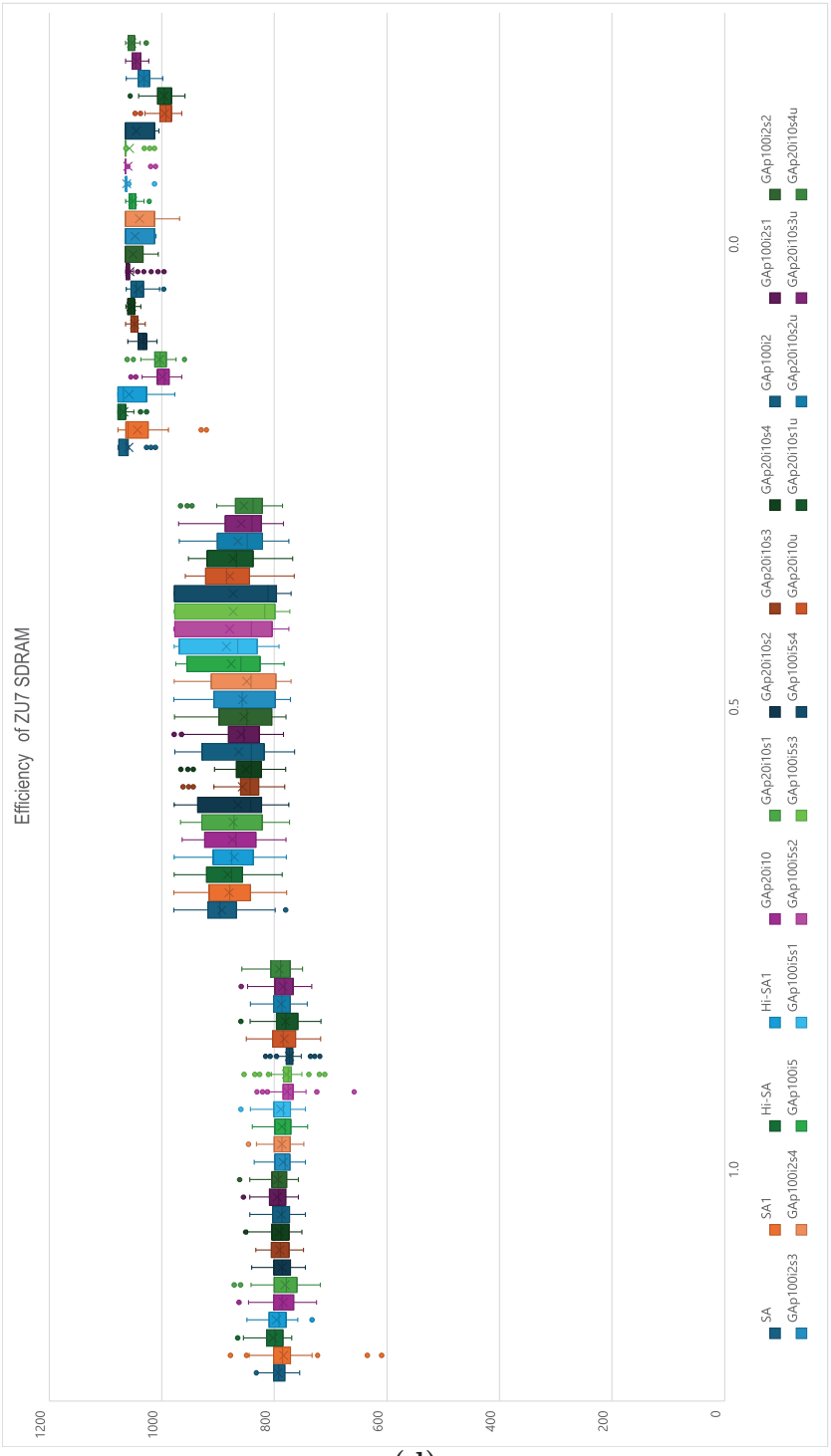
(a)


Efficiency of ZU15 SDRAM
2000
1800
1600
1400
1200
1000
800
600
400
200
0
1.0
0.5
0.0
SA
SA1
Hi-SA
Hi-SA1
GAp20i10
GAp20i10s1
GAp20i10s2
GAp20i10s3
GAp20i10u
GAp20i10s4
GAp20i10s1u
GAp20i10s2u
GAp20i10s3u
GAp20i10s4u
GAp100i2
GAp100i2s1
GAp100i2s2
GAp100i2s3
GAp100i2s4
GAp100i5
GAp100i5s1
GAp100i5s2
GAp100i5s3
GAp100i5s4

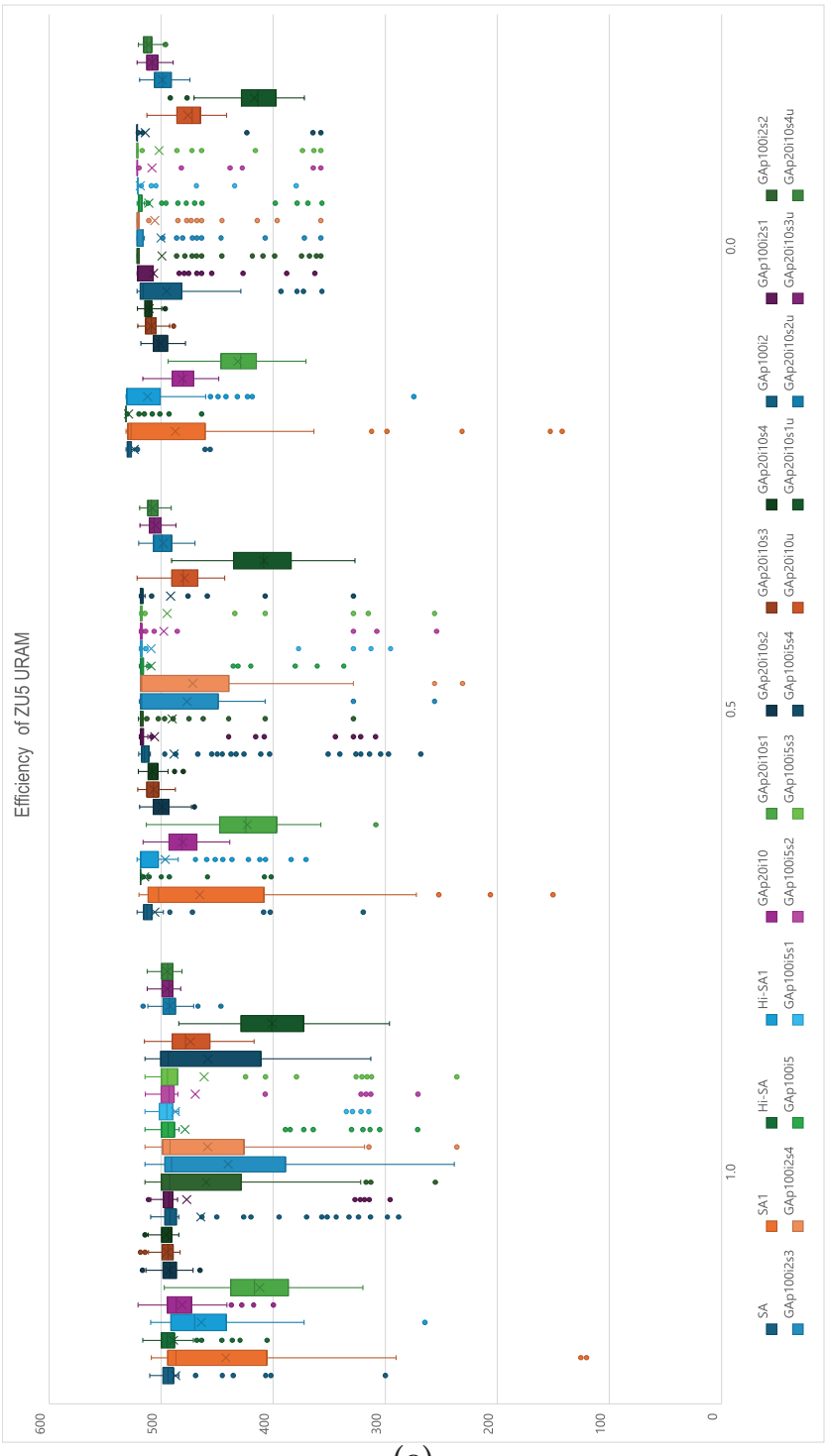
(b)


Efficiency of ZU7 URAM
1200
1000
800
600
400
200
0
1.0
0.5
0.0
SA
SA1
Hi-SA
Hi-SA1
GAp20l10
GAp20l10s1
GAp20l10s2
GAp20l10s3
GAp20l10u
GAp20l10s4
GAp20l10s1u
GAp20l10s2u
GAp20l10s3u
GAp20l10s4u
GAp100i2s3
GAp100i2s4
GAp100i5
GAp100i5s1
GAp100i5s2
GAp100i5s3
GAp100i5s4
GAp100i2
GAp100i2s1
GAp100i2s2

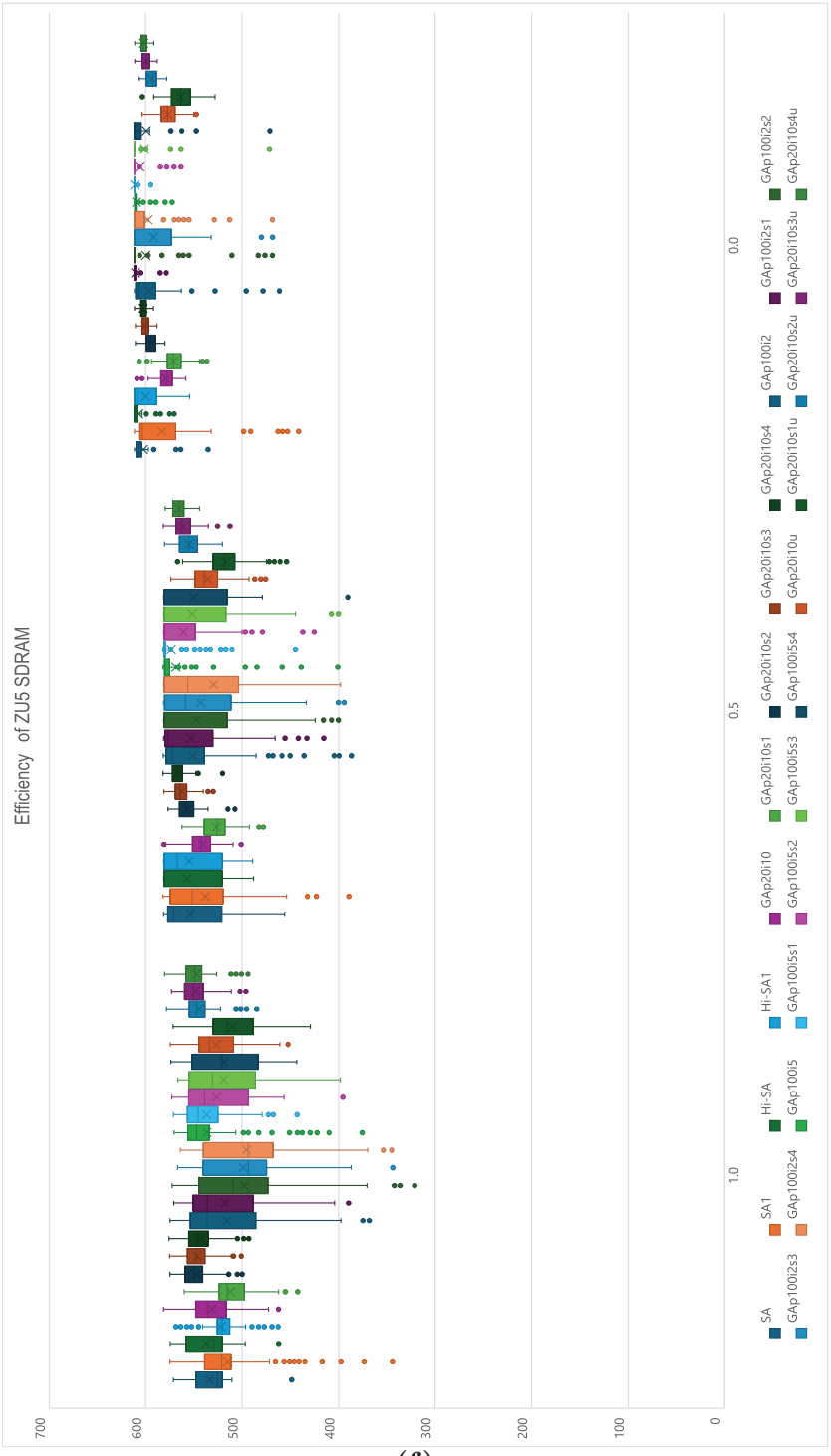
(c)


Efficiency of ZU7 SDRAM
1200
1000
800
600
400
200
0
1.0
0.5
0.0
SA
SA1
Hi-SA
Hi-SA1
GAp20i10
GAp20i10s1
GAp20i10s2
GAp20i10s3
GAp20i10s4
GAp20i10u
GAp20i10s1u
GAp20i10s2u
GAp20i10s3u
GAp20i10s4u
GAp100i2
GAp100i2s1
GAp100i2s2
GAp100i2s3
GAp100i2s4
GAp100i5
GAp100i5s1
GAp100i5s2
GAp100i5s3
GAp100i5s4

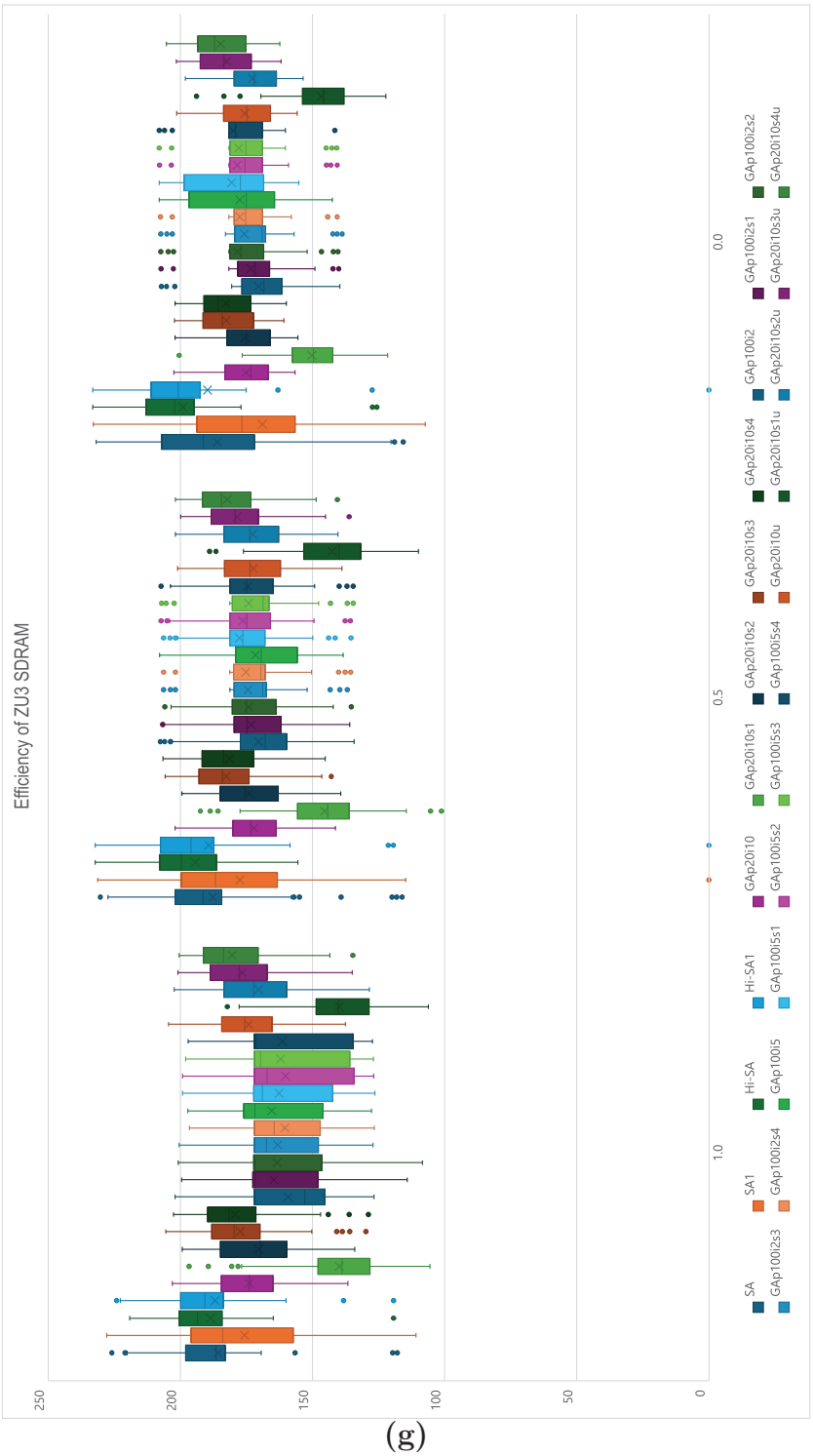
(d)


Efficiency of ZU5 URAM
600
500
400
300
200
100
0
1.0
0.5
0.0
SA
SA1
Hi-SA
Hi-SA1
GAp20l10
GAp20l10s1
GAp20l10s2
GAp20l10s3
GAp20l10s4
GAp20l10u
GAp100l2
GAp100l2s1
GAp100l2s2
GAp100i2s3
GAp100i2s4
GAp100i5
GAp100i5s1
GAp100i5s2
GAp100i5s3
GAp100i5s4
GAp20i10s1u
GAp20i10s2u
GAp20i10s3u
GAp20i10s4u

(e)

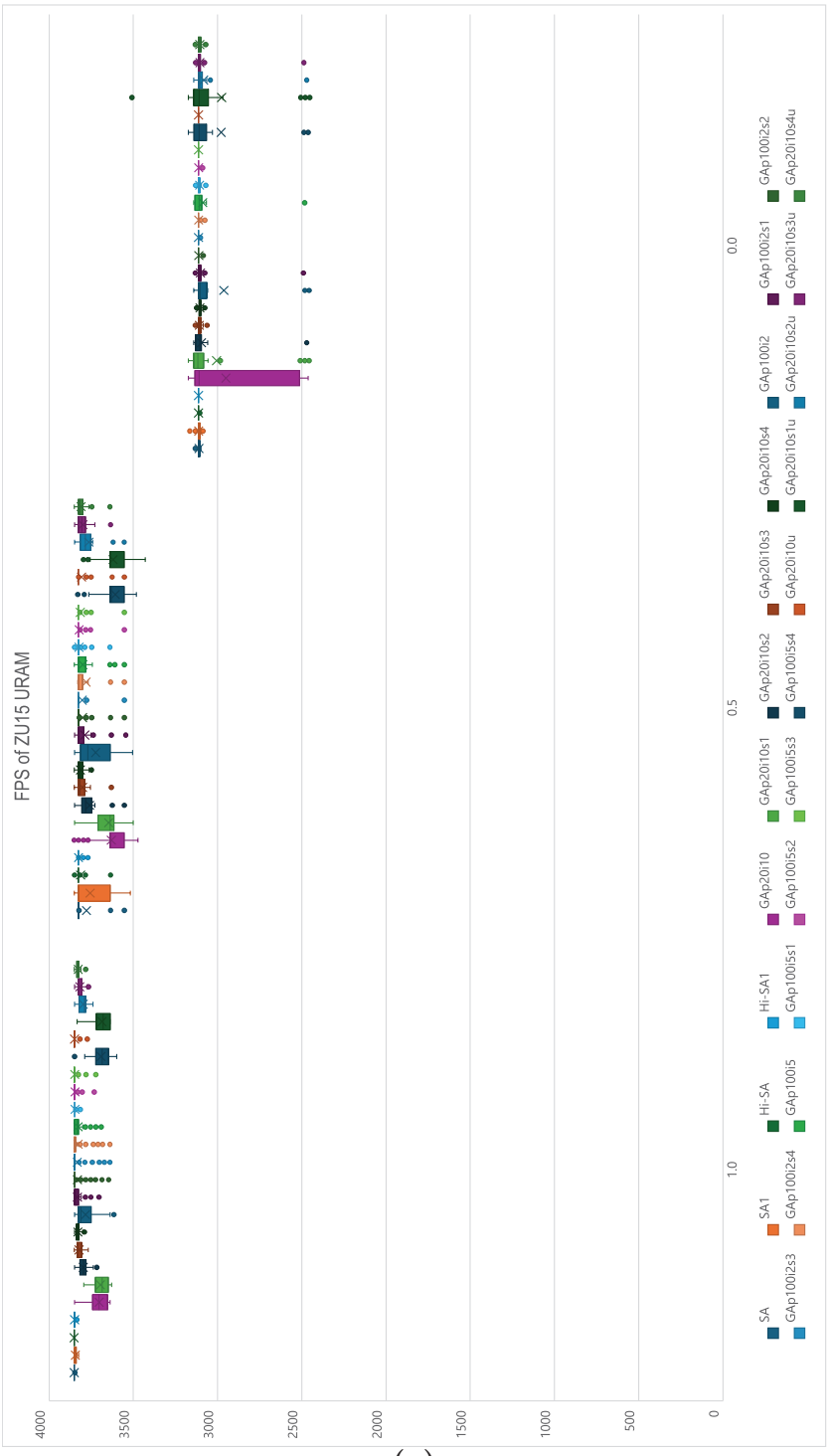

Efficiency of ZU5 SDRAM
700
600
500
400
300
200
100
0
0.0
0.5
1.0
SA
SA1
Hi-SA
Hi-SA1
GAp20i10
GAp20i10s1
GAp20i10s2
GAp20i10s3
GAp20i10u
GAp20i10s4
GAp20i10s1u
GAp20i10s2u
GAp20i10s3u
GAp20i10s4u
GAp100i2s3
GAp100i2s4
GAp100i5
GAp100i5s1
GAp100i5s2
GAp100i5s3
GAp100i5s4
GAp100i2
GAp100i2s1
GAp100i2s2

(f)

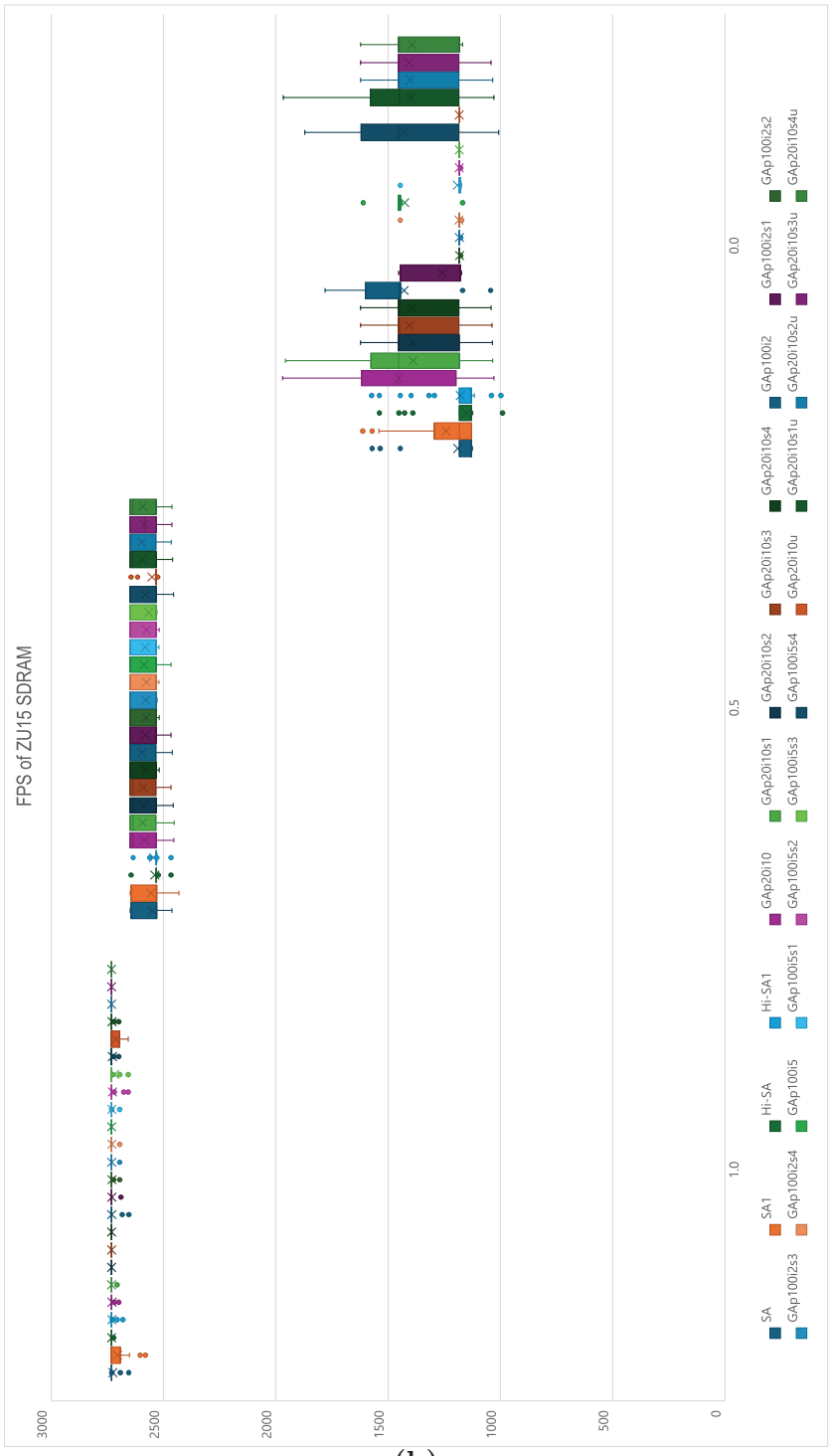


(g)

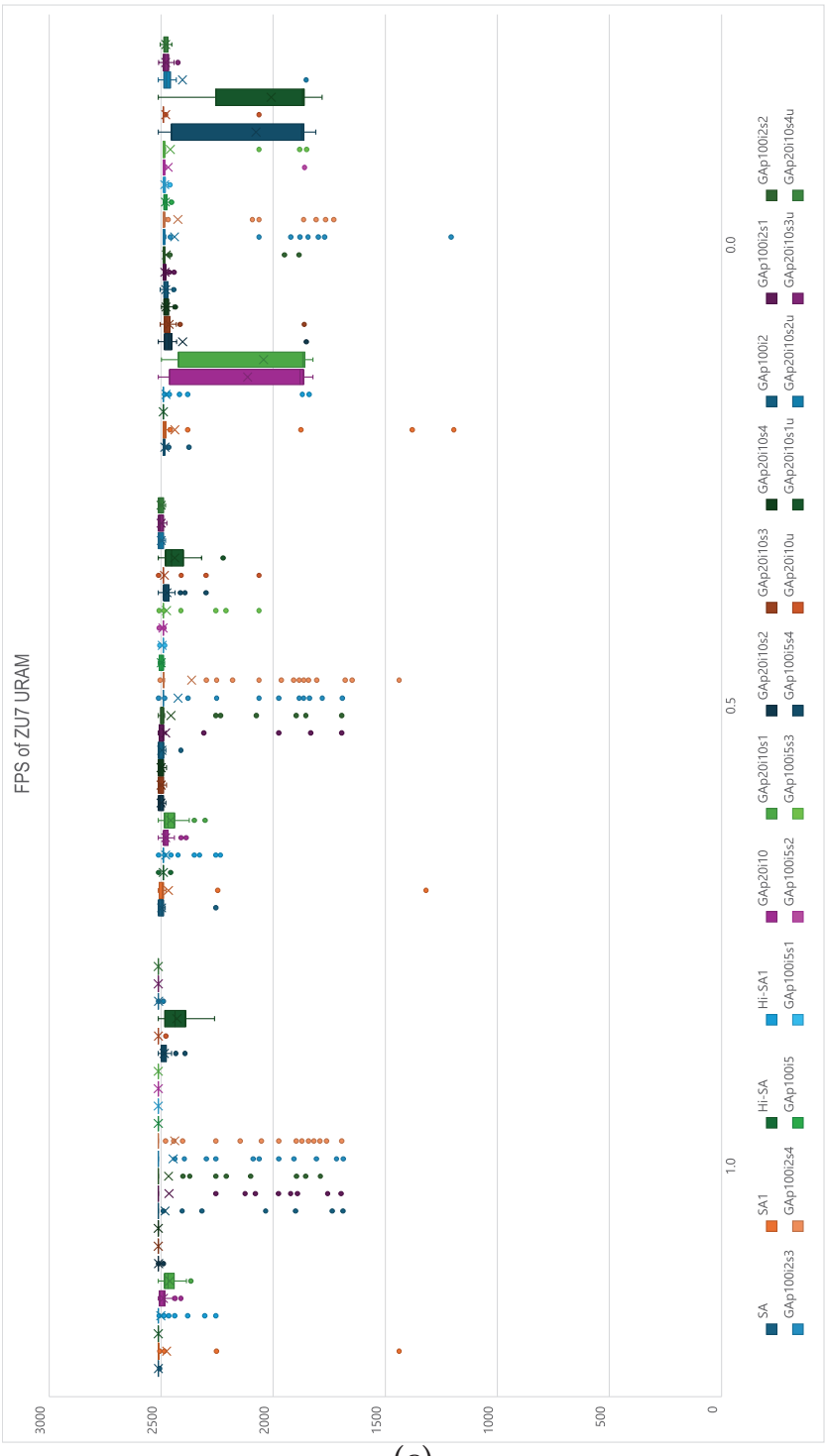
Figure 7.5: Efficiency distribution


FPS of ZU15 URAM
4000
3500
3000
2500
2000
1500
1000
500
0
1.0
0.5
0.0
SA
SA1
Hi-SA
Hi-SA1
GAp20i10
GAp20i10s1
GAp20i10s2
GAp20i10s3
GAp20i10s4
GAp20i10u
GAp20i10s1u
GAp20i10s2u
GAp20i10s3u
GAp20i10s4u
GAp100i2
GAp100i2s1
GAp100i2s2
GAp100i2s3
GAp100i2s4
GAp100i5
GAp100i5s1
GAp100i5s2
GAp100i5s3
GAp100i5s4

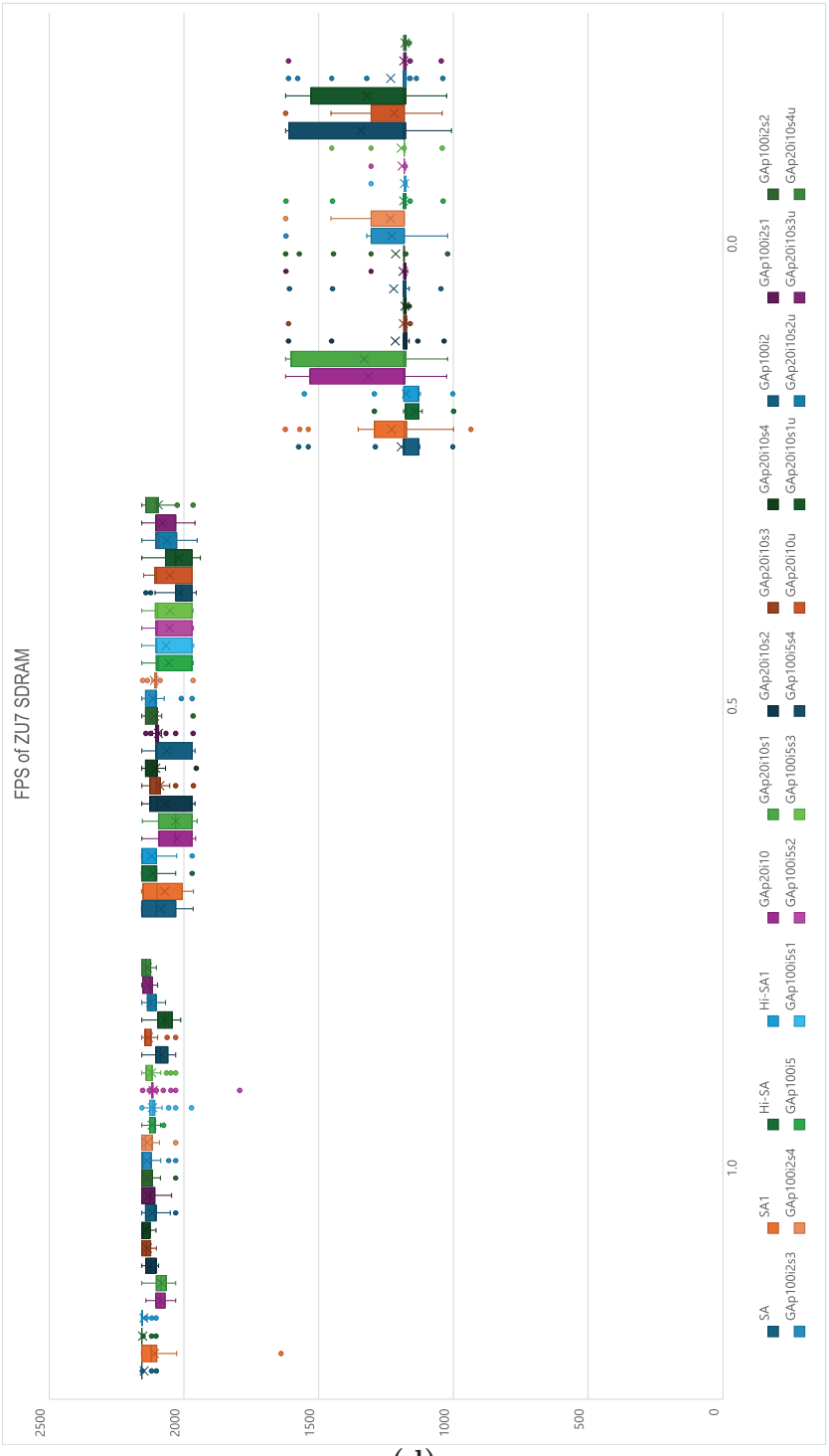
(a)


FPS of ZU15 SDRAM
3000
2500
2000
1500
1000
500
0
1.0
0.5
0.0
SA
SA1
Hi-SA
Hi-SA1
GAp20i10
GAp20i10s1
GAp20i10s2
GAp20i10s3
GAp20i10s4
GAp20i10u
GAp20i10s1u
GAp20i10s2u
GAp20i10s3u
GAp20i10s4u
GAp100i2
GAp100i2s1
GAp100i2s2
GAp100i2s3
GAp100i2s4
GAp100i5
GAp100i5s1
GAp100i5s2
GAp100i5s3
GAp100i5s4

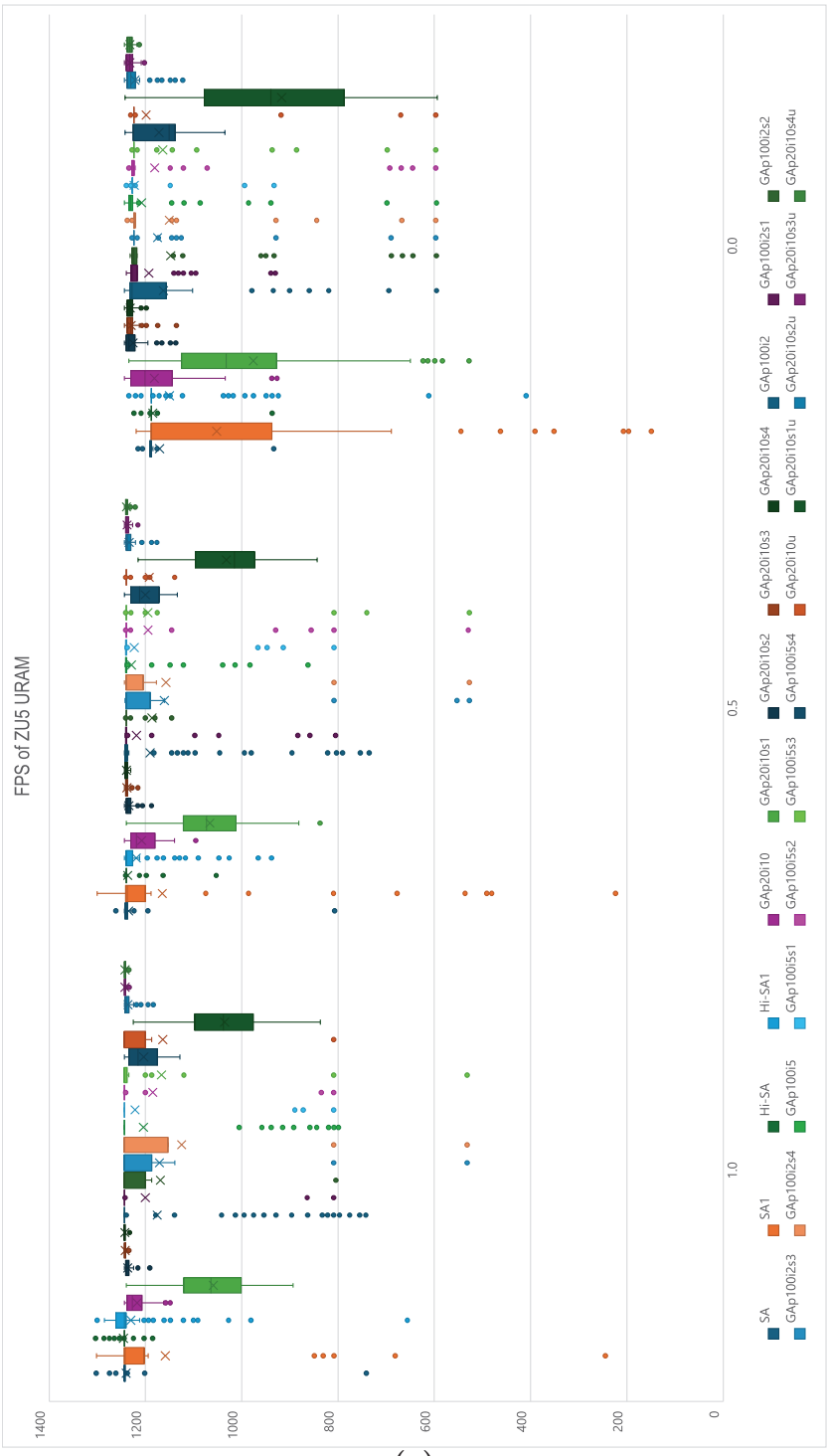
(b)


FPS of ZU7 URAM
3000
2500
2000
1500
1000
500
0
1.0
0.5
0.0
SA
SA1
Hi-SA
Hi-SA1
GAp20l10
GAp20l10s1
GAp20l10s2
GAp20l10s3
GAp20l10s4
GAp20l10u
GAp20l10s1u
GAp20l10s2u
GAp20l10s3u
GAp20l10s4u
GAp100i2s3
GAp100i2s4
GAp100i5
GAp100i5s1
GAp100i5s2
GAp100i5s3
GAp100i5s4
GAp100i2
GAp100i2s1
GAp100i2s2

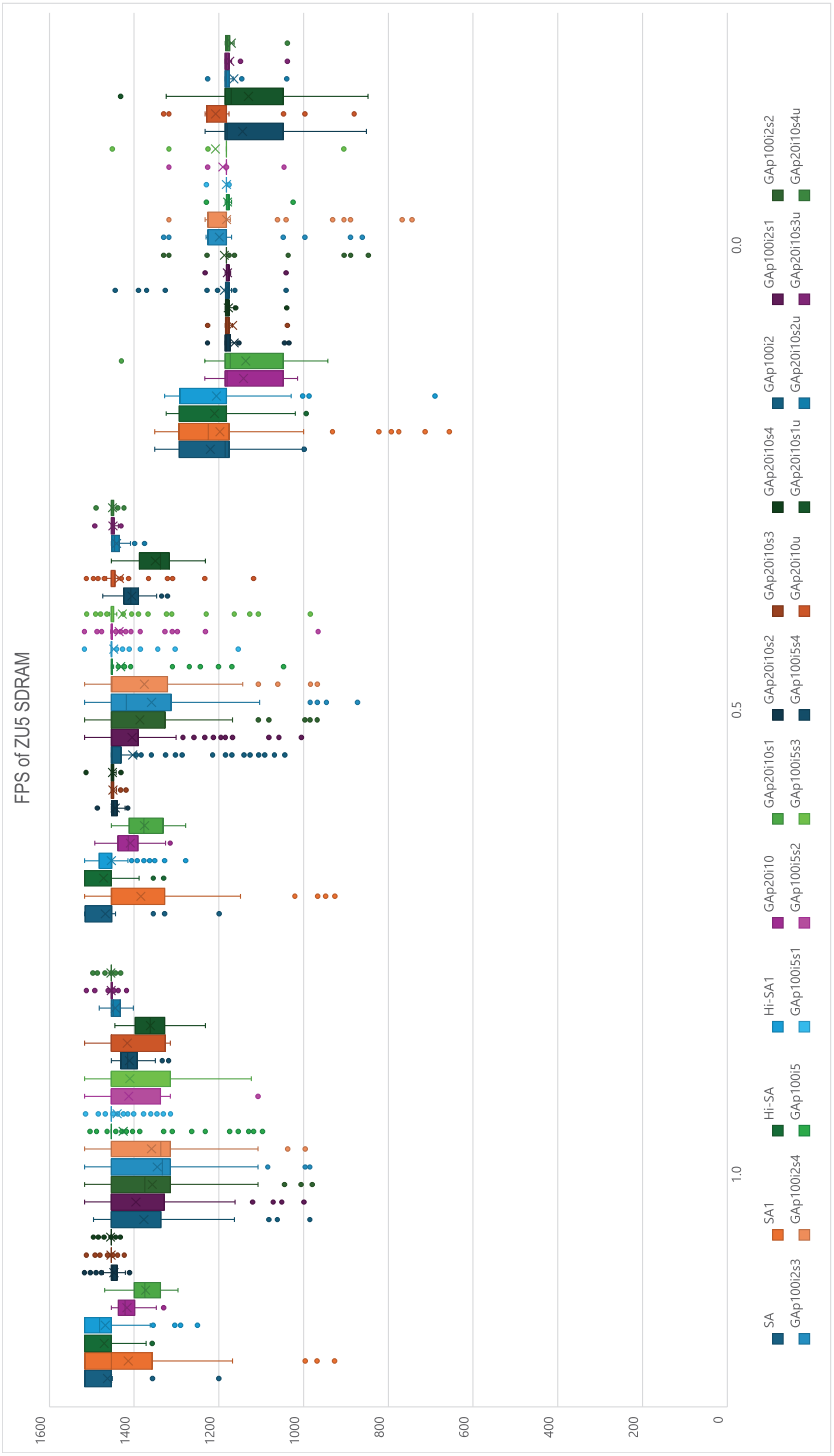
(c)


FPS of ZU7 SDRAM
2500
2000
1500
1000
500
0
1.0
0.5
0.0
SA
SA1
Hi-SA
Hi-SA1
GAp20i10
GAp20i10s1
GAp20i10s2
GAp20i10s3
GAp20i10s4
GAp20i10u
GAp20i10s1u
GAp20i10s2u
GAp20i10s3u
GAp20i10s4u
GAp100i2s3
GAp100i2s4
GAp100i5
GAp100i5s1
GAp100i5s2
GAp100i5s3
GAp100i5s4
GAp100i2
GAp100i2s1
GAp100i2s2

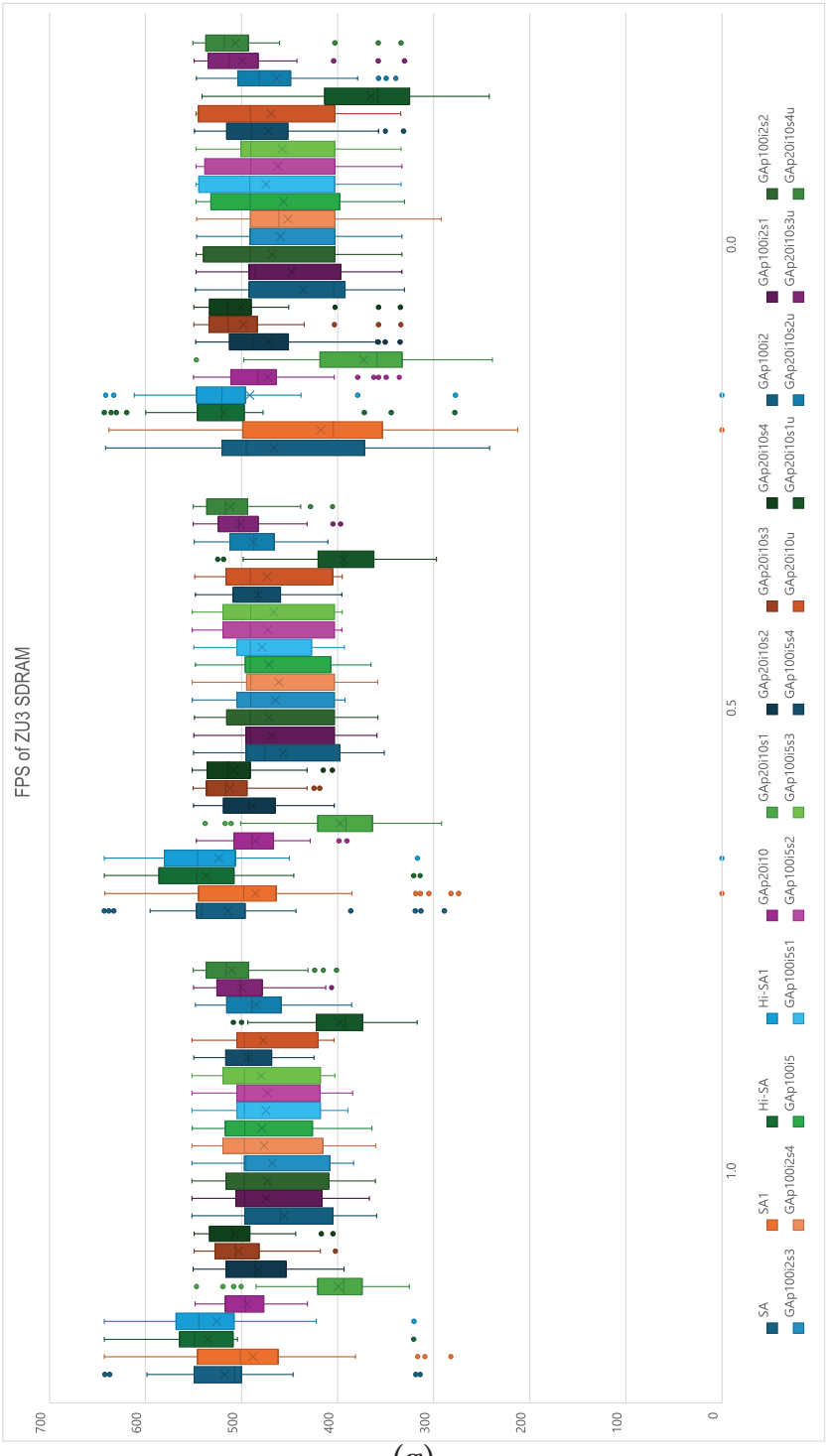
(d)


FPS of ZU5 URAM
1400
1200
1000
800
600
400
200
0
0.0
0.5
1.0
SA
SA1
Hi-SA
Hi-SA1
GAp20l10
GAp20l10s1
GAp20l10s2
GAp20l10s3
GAp20l10s4
GAp20l10u
GAp100l2
GAp100l2s1
GAp100l2s2
GAp100l2s3
GAp100l2s4
GAp100l5
GAp100l5s1
GAp100l5s2
GAp100l5s3
GAp100l5s4
GAp20l10s1u
GAp20l10s2u
GAp20l10s3u
GAp20l10s4u

(e)


FPS of ZU5 SDRAM
1600
1400
1200
1000
800
600
400
200
0
1.0
0.5
0.0
SA
SA1
Hi-SA
GAp20i10
GAp20i10s1
GAp20i10s2
GAp20i10s3
GAp20i10s4
GAp20i10u
GAp20i10s1u
GAp20i10s2u
GAp20i10s3u
GAp20i10s4u
GAp100i2s3
GAp100i2s4
GAp100i5
GAp100i5s1
GAp100i5s2
GAp100i5s3
GAp100i5s4
GAp100i2
GAp100i2s1
GAp100i2s2

(f)



(g)

Figure 7.6: Throughput distribution

In terms of efficiency, as p decreases, the optimization objective shifts toward higher efficiency, which is evident in Fig. 7.5. In most cases, Simulated Annealing continues to show more stable results and better solutions than the Genetic Algorithm. Figure. 7.5(g) clearly shows that Simulated Annealing outperforms the Genetic Algorithm.

Regarding throughput, as p decreases, throughput gradually decreases slightly. However, from Figure. 7.6, we can see that in most experimental scenarios, the difference in throughput between $p = 1.0$ and $p = 0.5$ is minor. However, the efficiency difference in Figure. 7.5 is significant. Therefore, in practical applications, it may be preferable to add a slight efficiency weighting in the cost function evaluation rather than solely focusing on throughput, even when pursuing high throughput.

Solutions obtained with Simulated Annealing at $p = 0.5$ are listed in Table. 7.9. Compared to the previous manually specified configuration, the solutions found were significantly better. For ZU7 URAM, the improvement was less noticeable as the differences weren't as pronounced, but for ZU7 SDRAM, there was a 21% improvement. Similarly, for ZU3 SDRAM, the improvement was nearly 15%. Not only was throughput improved, but the performance per area was significantly enhanced, with a more balanced utilization across various resources.

Table 7.9: Solutions Obtained by Simulated Annealing

	ZU15 URAM	ZU15 SDRAM	ZU7 URAM	ZU7 SDRAM	ZU5 URAM	ZU5 SDRAM	ZU3 SDRAM
$LITE$	1	1	1	1	1	1	1
B	6	6	4	6	2	5	2
G	12	8	8	6	8	4	4
DC	4	4	4	4	4	4	4
$DWCCDIV$	1	1	1	2	1	2	2
$SWUMEM$	$K+2$	$K+2$	$K+2$	$K+2$	$K+2$	$K+2$	$K+2$
PE_W	2624	5120	3072	7936	2816	6080	4096
PE_Q	64	64	64	64	64	64	64
$PEDW_{WQ}$	256	256	256	448	256	256	256
FCDIV	2	1	2	1	2	2	4
$MADD_{DSP}$	0.997	1	1	0.878	1	1	0.719
ACC_{DSP}	0.334	1	0.434	0.022	1	0.797	0
$PE_{WMEM_{BRAM}}$	1	1	0.683	0	0.405	0.362	0.919
$AMEM_{BRAM}$	1	1	1	1	1	1	1
SWU_{MEM}	BRAM	BRAM	BRAM	BRAM	BRAM	URAM	BRAM
DWC_{MEM}	BRAM	BRAM	BRAM	BRAM	BRAM	URAM	BRAM
$PE_{WMEM_{mode}}$	CAS	CAS	CAS	MUX	MUX	CAS	CAS
LUT	199137	164017	125383	158541	76203	94695	58477
FF	341743	241963	1820633	226543	88763	116307	69829
DSP	2868	2280	1376	1380	760	1030	296
BRAM	426	458	234	138	110	106	174
URAM	96	0	72	48	48	51	0
FPS	3826.83	2532.76	2490.29	2155.67	1239.87	1517.22	642.94
Efficiency	1457.75	1438.56	1001.14	875.69	518.43	520.68	217.85
Cost	2642.29	1985.66	1745.71	1515.68	879.15	1018.95	630.39

7.7 Conclusion

This chapter focuses on optimizing accelerator design by exploring various optimization algorithms due to the complexity of the problem. The main algorithms used were Simulated Annealing and Genetic Algorithm. Simulated Annealing was explored in two scenarios: standard Simulated Annealing (SA) and Simulated Annealing with single-parameter updates (SA1). Results indicated that SA generally outperformed SA1 in terms of stability and solution quality. The Genetic Algorithm was extensively explored under different population and generation configurations, using various `softmax` bases and comparing with a version that did not use softmax.

In most cases, Simulated Annealing consistently demonstrated superior stability and solution quality compared to the Genetic Algorithm. Specific solutions obtained with Simulated Annealing, particularly at $p = 0.5$, showed significant improvements over manually specified configurations. Notably, throughput for ZU7 and ZU3 SDRAM targets increased considerably, along with substantial improvements in performance per unit area and resource utilization ratios.

Chapter 8

Conclusion

8.1 Conclusion

This dissertation set out to address the pressing challenges posed by modern CNNs, particularly lightweight models like MobileNetV2, on CNN accelerators. The primary goal was to find a balanced solution that combines the scalability and flexibility of Overlay architecture accelerators with the performance advantages of Dataflow architecture accelerators, all while minimizing off-chip memory dependency and maximizing resource efficiency.

The research presented herein made significant strides in the domain of CNN accelerators through the development of a novel block-based execution model. This model effectively mitigates the memory bandwidth bottlenecks that are inherent in Overlay architecture accelerators and overcomes the resource-intensive nature of Dataflow architecture accelerators. The following key findings emerged from the study:

Block-based Execution: The proposed block-based architecture processes entire inverted residual blocks as single units, which proved to be a strategic approach for reducing data transfer overhead. This design enables the accelerator to leverage the efficient handling of data within a block and minimizes reliance on high-bandwidth off-chip memory. By processing data through optimized on-chip pathways, the accelerator can sustain higher throughput compared to traditional Overlay architecture accelerators, which require frequent data storage and retrieval from off-chip memory.

Dual Configuration Strategy: The design supports two configurations: a URAM-based configuration that capitalizes on the high on-chip memory available in mid-range to high-end FPGA devices, and an SDRAM-based configuration tailored for more constrained devices. This dual configuration ensures that the architecture is adaptable and scalable across different FPGA platforms, making it suitable for a range of practical applications, from edge

devices to powerful data center accelerators.

Parallelism and Resource Utilization: The architecture's ability to process multiple images in parallel, akin to the batch processing seen in GPUs, was shown to significantly enhance resource utilization and overall throughput. This strategy mitigates the diminishing returns of increasing parallelism within a single PE Array.

Two Execution Modes: The accelerator supports two execution modes and allows dynamic switching at runtime based on the workload. The serial execution mode can run blocks efficiently while the parallel execution mode is critical for efficiently executing individual layers and reducing the on-chip weight memory capacity.

Efficiency with Lightweight Models: The architecture was particularly effective when deployed on lightweight models such as MobileNetV2. The block-based execution reduced the high inter-layer activation transfer, a common bottleneck in depthwise convolution layers. This feature ensures that the proposed design not only achieves high throughput but also maintains efficient DSP and LUT utilization, critical for achieving high area efficiency on FPGA devices.

Extensive experiments validated the performance gains offered by the proposed architecture. The accelerator achieved notable throughput improvements across various FPGA platforms: ZU3 (cost-optimized): Achieved 586 FPS, demonstrating a significant performance increase over traditional Overlay architecture accelerators with lower memory bandwidth. ZU7 (mid-range): Reached 2,350 FPS, leveraging enhanced on-chip memory for efficient inter-block data storage. VU13P (high-end): Realized a peak performance of 11,821 FPS, thanks to the extensive use of URAM for storing inter-block data and maintaining high parallelism.

These results underscore the architecture's scalability and suitability for various hardware configurations. The use of on-chip URAM for inter-block buffering was particularly beneficial in high-end devices, where ample memory resources allowed the accelerator to operate with minimal off-chip data transfer, maintaining high throughput.

The proposed architecture offers a new pathway for developing efficient, scalable CNN accelerators capable of handling modern lightweight and complex CNNs. The reduction in off-chip memory access translates directly to lower energy consumption and higher data processing rates, which are critical for applications in edge computing, where power and memory bandwidth are

often limited. Furthermore, the architecture's modular design allows for flexibility in adapting to different CNN structures and requirements without significant redesign efforts. For other network structures, such as RNNs and transformer models, their specialized accelerators could be designed at the block level. These structures typically consist of repeated blocks or repeatedly feeding a single block with data. This characteristic suggests that our accelerator architecture may effectively compute these models for other tasks.

By supporting a range of FPGA platforms, the architecture ensures that even devices with modest on-chip memory can benefit from a version of the design (SDRAM-based configuration), making it a versatile solution for diverse deployment scenarios.

Minor research concludes with a detailed evaluation of optimization strategies for designing efficient CNN accelerators on FPGA platforms. The findings confirm that choosing the right combination of resource allocation and parallelism configuration can dramatically impact the performance of CNN accelerators. Among the techniques studied, Simulated Annealing proved to be particularly effective for complex, parameter-rich optimization problems. Its iterative approach, coupled with controlled randomness, allowed it to find solutions that balance throughput and resource use better than static configurations.

The Genetic Algorithm, while showing promise with larger populations and varied crossover methods, struggled to match the stability and performance of SA under equivalent computational loads. Nonetheless, the GA's adaptability and the ability to explore diverse solution spaces make it a valuable tool for multi-objective optimization scenarios.

Key experimental results highlighted substantial throughput and efficiency gains. For instance, tests on the ZU7 FPGA using SA yielded configurations that outperformed manually tuned designs by up to 21%, particularly when utilizing SDRAM-based memory setups. The chapter also noted that using URAM configurations where feasible provided significant improvements in larger FPGA devices, while smaller devices benefited from the strategic use of BRAM.

Minor research's insights into optimization reveal that practical accelerator designs require careful balancing of on-chip memory, DSP usage, and algorithmic complexity. Future research could expand on dynamic optimization strategies that adapt to changing workloads and explore hybrid

approaches combining SA and GA elements. These extensions could further enhance efficiency and make CNN accelerators more competitive for diverse applications, from edge devices to data centers.

In conclusion, the block-based accelerator presented in this dissertation represents a significant advancement in the field of CNN acceleration. By strategically combining the strengths of Overlay and Dataflow architecture accelerators while addressing their weaknesses, this architecture provides a robust solution for high-throughput, memory-efficient CNN processing. Compared to typical Overlay architecture accelerators, this design reduces overall off-chip memory transfer volume by 93%, eliminating the need for expensive high-speed memory and thereby lowering the cost of large-scale data center deployments. Compared to typical Dataflow architecture accelerators, this design reduces the on-chip weight storage requirement by 88%, making it deployable on cost-optimized small devices and significantly reducing the deployment cost for edge computing. The demonstrated performance gains, coupled with its adaptability and scalability, position this design as a compelling option for future CNN accelerator development. The results of this study pave the way for more sophisticated and efficient hardware solutions that cater to the ever-growing demands of machine learning and AI applications.

8.2 Future Works

Despite the advancements made with the proposed accelerator design, there remain several challenges that must be addressed to improve its overall performance and versatility. One significant issue is that the current interconnection design for internal accelerator modules is still reliant on manual processes. This approach is not only highly labor-intensive but also greatly limits the architecture's adaptability to diverse network structures, hindering its potential for rapid deployment across different CNN models.

Furthermore, the use of handshake bus connections between modules imposes stringent timing requirements for path switching. This constraint allows for the insertion of handshake registers only at the outputs of specific modules, which can restrict flexibility and extensibility. The accumulation of too many paths within the architecture exacerbates this issue, potentially leading to long "ready" path delays and complications in signal synchronization. These timing challenges can impact data throughput and overall

system stability, particularly as network complexity increases.

Another critical aspect to consider is the performance of the accelerator when running computationally intensive models. The current design does not show significant advantages in these scenarios, highlighting the need for additional optimization techniques such as pruning. Integrating pruning support into the accelerator could help reduce the computational load and enhance efficiency by eliminating unnecessary operations and weights. However, implementing pruning effectively within a Dataflow architecture poses its own set of challenges, particularly in ensuring that the architecture can dynamically adapt to pruned network structures without sacrificing data consistency or processing speed.

Addressing these limitations—automating the interconnection design, managing handshake timing more efficiently, and integrating pruning capabilities—will be essential for the continued evolution of this accelerator. These enhancements would not only boost adaptability and performance but also make the architecture more competitive for a wider range of CNN applications, from lightweight models to those with heavier computational demands.

During the design phase, we considered dynamically allocating PEs to different PE Arrays based on workload. However, after evaluation, it was determined that such dynamic allocation would require the relevant PE Array to remain idle during reallocation. For the inverted residual blocks accelerated in this dissertation, PE Array 0 would need to wait for PE Array 1 to complete its work before dynamic reallocation could occur, resulting in additional waiting time as shown in Figure. 3.15. The benefits of dynamic allocation may not outweigh this extra waiting time.

However, considering the potential for ASIC implementation, this feature could contribute to building a general-purpose CNN accelerator. By assigning PEs to different PE Arrays before execution and not changing this allocation during operation, it is possible to leverage the advantages of this feature while avoiding potential issues. Additionally, implementing the PE Arrays using Versal's AI Engine(-ML) Tiles is feasible, provided the number of parallel pixels is increased to suit the systolic array, or utilizing multiplier columns from the systolic array to process data from different images.

More complex on-chip interconnects need to be introduced, allowing connections between all computation engines and establishing independent pathways between any two engines. This would make it possible to create a

more general-purpose accelerator. Such a general-purpose accelerator could be valuable if implemented as an ASIC. The equivalent number of 8×8 multipliers in the implementation on the VU13P FPGA already exceeds 10,000, but still fewer than that of modern ASIC accelerators. On the other hand, the maximum frequency is limited by the FPGA platform, operating at approximately 600 MHz, which is significantly lower than the 1.5 GHz to 2 GHz typical of ASIC-based parallel accelerators.

Implementing this as an ASIC could significantly narrow the computational power gap with GPUs and other ASIC implementations. It would allow high-throughput execution of models like ResNet50 and YOLO while requiring minimal off-chip memory bandwidth. It may also handle models such as RNNs and transformers with high efficiency.

With the rapid advancement of AI-related technologies, their applications are gradually integrating into everyday life. Due to the limitations of edge computing devices, most reliance still falls on high-performance computing in data centers. If the technologies related to this research achieve further development and widespread application, both edge computing and high-performance computing can benefit from the extremely low bandwidth requirements.

Firstly, compared to current computing products that depend on costly HBM or relatively less expensive GDDR memory, our architecture can meet bandwidth demands using much cheaper DDR memory, directly influencing the final product pricing. For edge computing products, a more affordable price allows more people to enjoy the convenience brought by technological advancements. For data centers, this reduces procurement costs and impacts the pricing of computing services. Reduced memory access means less power is used for data transfer, enhancing the battery life of edge devices and improving the user experience when leveraging AI-related technologies on edge computing devices.

Lower power consumption also leads to reduced heat generation. For data centers, this translates to lower operational costs due to reduced power consumption and cooling needs, which also affects the pricing of related services. Currently, data centers are major consumers of electricity and water (for cooling purposes), so this research contributes to energy conservation and emissions reduction. More affordable edge computing products and data center services will encourage these technologies to penetrate every corner of daily life, improving convenience and productivity.

Just as ultra-low-cost MCUs like the ESP8266, ESP32 and RP2040 have driven IoT-related products and services into every aspect of life, we believe our outcomes will also promote AI's integration into all facets of daily life.

Bibliography

[1] D. Wu, Y. Zhang, X. Jia, L. Tian, T. Li, L. Sui, D. Xie, and Y. Shan, "A high-performance cnn processor based on fpga for mobilenets," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 136–143.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. Red Hook, NY, USA: Curran Associates Inc., 2012, p. 1097–1105.

[3] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[4] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal *et al.*, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 1–12, Jun 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080246

[5] Xilinx. (2023, Jan) Dpuczdx8g for zynq ultrascale+ mpsocs product guide (pg338) v4.1. [Online]. Available: https://docs.xilinx.com/r/en-US/pg338-dpu/Xilinx-Model-Zoo-Performance

[6] Nvidia. (2018) Nvdla primer. [Online]. Available: http://nvdla.org/primer.html

[7] S. Yan, Z. Liu, Y. Wang, C. Zeng, Q. Liu, B. Cheng, and R. C. Cheung, "An fpga-based mobilenet accelerator considering network structure characteristics," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 17–23.

[8] M. Sun, Z. Li, A. Lu, Y. Li, S.-E. Chang, X. Ma, X. Lin, and Z. Fang, "Film-qnn: Efficient fpga acceleration of deep neural networks with intra-layer, mixed-precision quantization," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 134–145. [Online]. Available: https://doi.org/10.1145/3490422.3502364

[9] S. I. Venieris and C.-S. Bouganis, "fpgaconvnet: A framework for mapping convolutional neural networks on fpgas," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 40–47.

[10] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre *et al.*, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. ACM, 2017, pp. 65–74.

[11] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, Dec 2018. [Online]. Available: https://doi.org/10.1145/3242897

[12] M. Ghasemzadeh, M. Samragh, and F. Koushanfar, "Rebnet: Residual binarized neural network," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 57–64.

[13] Z. Dong, Y. Gao, Q. Huang, J. Wawrzynek, H. H. So, and K. Keutzer, "Hao: Hardware-aware neural architecture optimization for efficient inference," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2021, pp. 50–59. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/FCCM51124.2021.00014

[14] W. Jiang, H. Yu, and Y. Ha, "A high-throughput full-dataflow mobilenetv2 accelerator on edge fpga," *IEEE Transactions on Computer-*

Aided Design of Integrated Circuits and Systems, vol. 42, no. 5, pp. 1532–1545, 2023.

[15] L. Petrica, T. Alonso, M. Kroes, N. J. Fraser, S. D. Cotofana, and M. Blott, "Memory-efficient dataflow inference for deep cnns on fpga," *2020 International Conference on Field-Programmable Technology (ICFPT)*, pp. 48–55, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:226965433

[16] M. Kroes, L. Petrica, S. Cotofana, and M. Blott, "Evolutionary bin packing for memory-efficient dataflow inference acceleration on fpga," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, ser. GECCO '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1125–1133. [Online]. Available: https://doi.org/10.1145/3377930.3389808

[17] Xilinx. (2020) Resnet50-pynq. [Online]. Available: https://github.com/Xilinx/ResNet50-PYNQ/blob/master/link/README.md

[18] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[19] NVIDIA. Nvidia a100 tensor core gpu architecture. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[20] ——. Nvidia a100 80gb pcie gpu product brief. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/PB-10577-001_v02.pdf

[21] ——. Nvidia h100 tensor core gpu architecture. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core

[22] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[24] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, Jun 2018, pp. 4510–4520. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2018.00474

[25] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 2019, pp. 6105–6114. [Online]. Available: http://proceedings.mlr.press/v97/tan19a.html

[26] A. Howard, M. Sandler, B. Chen, W. Wang, L.-C. Chen, M. Tan, G. Chu, V. Vasudevan, Y. Zhu, R. Pang, H. Adam, and Q. Le, "Searching for mobilenetv3," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019, pp. 1314–1324.

[27] M. Tan and Q. Le, "Efficientnetv2: Smaller models and faster training," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 10 096–10 106. [Online]. Available: https://proceedings.mlr.press/v139/tan21a.html

[28] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, p. 448–456.

[29] Xilinx. (2018, Mar) 7 series dsp48e1 slice user guide (ug479) v1.10. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug479_7Series_DSP48E1

[30] ——. (2021, Aug) Ultrascale architecture dsp slice user guide (ug579) v1.11. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp

[31] ——. (2022, Sept) Versal acap dsp engine architecture manual (am004). [Online]. Available: https://docs.amd.com/r/en-US/am004-versal-dsp-engine

[32] S. Lee, D. Kim, D. Nguyen, and J. Lee, "Double mac on a dsp: Boosting the performance of convolutional neural networks on fpgas," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 888–897, 2019.

[33] J. Sommer, M. A. Özkan, O. Keszocze, and J. Teich, "Dsp-packing: Squeezing low-precision arithmetic into fpga dsp blocks," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022, pp. 160–166.

[34] Xilinx. (2020, Jun) Convolutional neural network with int4 optimization on xilinx devices (wp521) v1.0.1. [Online]. Available: https://docs.xilinx.com/v/u/en-US/wp521-4bit-optimization

[35] ——. (2017, Apr) Deep learning with int8 optimization on xilinx devices white paper (wp486) v1.0.1. [Online]. Available: https://docs.xilinx.com/v/u/en-US/wp486-deep-learning-int8

[36] Y. Chen and K. Tanaka, "High throughput and low bandwidth demand: Accelerating cnn inference block-by-block on fpgas," in *2024 27th Euromicro Conference on Digital System Design (DSD)*, 2024, pp. 503–511.

[37] Xilinx. (2021) System performance modeling project. [Online]. Available: https://xilinx.github.io/Embedded-Design-Tutorials/docs/2021.2/build/html/docs/User_Guides/SPA-UG/docs/2-system-performance-modeling-project.html

[38] ——. Logicore axi traffic generator. [Online]. Available: https://www.xilinx.com/products/intellectual-property/axi_tg.html

[39] ——. Axi performance monitor. [Online]. Available: https://www.xilinx.com/products/intellectual-property/axi_perf_mon.html

[40] ——. (2023, Dec) Zynq ultrascale+ mpsoc data sheet: Dc and ac switching characteristics (ds925). [Online]. Available: https://docs.amd.com/r/en-US/ds925-zynq-ultrascale-plus

[41] Veripool. (1994, Jul) Verilator. [Online]. Available: https://veripool.org/verilator/

[42] Y. Chen and K. Tanaka, "Better scalability: Improvement of block-based cnn accelerator for fpgas," *IEEE Access*, vol. 12, pp. 187 587–187 603, 2024.

[43] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2017.

[44] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7132–7141.

[45] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha, "Learned step size quantization," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=rkgO66VKDS

[46] Xilinx. (2012) Vivado design suite. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html

[47] ——. (2024, Mar) Ultrascale architecture and product data sheet: Overview (ds890). [Online]. Available: https://docs.amd.com/v/u/en-US/ds890-ultrascale-overview

[48] ——. (2024, Jun) Versal architecture and product data sheet: Overview (ds950). [Online]. Available: https://docs.amd.com/v/u/en-US/ds950-versal-overview

[49] ——. (2022, Apr) Ultrascale architecture-based fpgas memory ip product guide (pg150). [Online]. Available: https://docs.amd.com/v/u/en-US/pg150-ultrascale-memory-ip

[50] ——. (2023, Jul) Vitis ai 3.5 model zoo details and performance. [Online]. Available: https://xilinx.github.io/Vitis-AI/3.5/html/docs/workflow-model-zoo.html#model-zoo-details-and-performance

[51] Y. Yu, T. Zhao, K. Wang, and L. He, "Light-opu: An fpga-based overlay processor for lightweight convolutional neural networks," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 122–132. [Online]. Available: https://doi.org/10.1145/3373087.3375311

[52] B. Li, H. Wang, X. Zhang, J. Ren, L. Liu, H. Sun, and N. Zheng, "Dynamic dataflow scheduling and computation mapping techniques for efficient depthwise separable convolution acceleration," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 8, pp. 3279–3292, 2021.

[53] Xilinx. (2020, Dec) Finn v0.5b (beta) and finn-examples released. [Online]. Available: https://xilinx.github.io/finn/2020/12/17/finn-v05b-beta-is-released.html

[54] Y. Yang, Q. Huang, B. Wu, T. Zhang, L. Ma, G. Gambardella, M. Blott, L. Lavagno, K. Vissers, J. Wawrzynek, and K. Keutzer, "Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded fpgas," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 23–32. [Online]. Available: https://doi.org/10.1145/3289602.3293902

[55] Y. Zhang, J. Pan, X. Liu, H. Chen, D. Chen, and Z. Zhang, "Fracbnn: Accurate and fpga-efficient binary neural networks with fractional activations," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 171–182. [Online]. Available: https://doi.org/10.1145/3431920.3439296

[56] TensorFlow. (2020) Mobilenet. [Online]. Available: https://github.com/tensorflow/models/blob/master/research/slim/nets/mobilenet/README.md

[57] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 21–37.

[58] Tensorflow. (2018, Jul) ssd_mobilenet_v2_coco. [Online]. Available: https://github.com/tensorflow/models/blob/master/research/object_detection/samples/configs/ssd_mobilenet_v2_coco.config

[59] ——. (2018, Jul) ssdlite_mobilenet_v2_coco. [Online]. Available: https://github.com/tensorflow/models/blob/master/research/object_detection/samples/configs/ssdlite_mobilenet_v2_coco.config

[60] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *CoRR*, vol. abs/1804.02767, 2018. [Online]. Available: http://arxiv.org/abs/1804.02767

[61] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

Publications

[1] Y. Chen and K. Tanaka. "A Flexible, Fast, Low Bandwidth Block-based Acceleration Architecture for CNN Inference on FPGAs," in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 183. [Online]. Available: https://doi.org/10.1145/3626202.3637582 (Poster)

[2] Y. Chen and K. Tanaka, "High Throughput and Low Bandwidth Demand: Accelerating CNN Inference Block-by-block on FPGAs," in *27th Euromicro Conference on Digital System Design* (DSD), 2024, pp. 503-511.

[3] Y. Chen and K. Tanaka, "Better Scalability: Improvement of Block-Based CNN Accelerator for FPGAs," in *IEEE Access*, vol. 12, pp. 187587-187603, 2024.