UCLA
UCLA Electronic Theses and Dissertations

Title
Software and Hardware Co-optimization for Deep Learning Algorithms on FPGA

Permalink
https://escholarship.org/uc/item/02t771j7

Author
Wu, Chen

Publication Date
2022

Peer reviewed|Thesis/dissertation

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

UNIVERSITY OF CALIFORNIA

Los Angeles

Software and Hardware Co-optimization for Deep Learning Algorithms on FPGA

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Electrical and Computer Engineering

by

Chen Wu

2022

© Copyright by

Chen Wu

2022

ABSTRACT OF THE DISSERTATION

Software and Hardware Co-optimization for Deep Learning Algorithms on FPGA

by

Chen Wu

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Los Angeles, 2022

Professor Lei He, Chair

Over recent years, deep learning paradigms such as convolutional neural networks (CNNs) have shown great success in various families of tasks including object detection and autonomous driving, etc. To extend such success to non-euclidean data, graph convolutional networks (GCNs) have been introduced, and have quickly attracted industrial and academia attention as a popular solution to real-world problems. However, both CNNs and GCNs often have huge computation and memory complexity, which calls for specific hardware architectures to accelerate these algorithms. In this dissertation, we propose several architectures to accelerate CNNs and GCNs based on FPGA platforms.

We start from the domain-specific FPGA-overlay processor (OPU) on commonly used CNNs, such as VGG, Inception, ResNet, and YoloV2. The data is first quantized to 8-bit fixed-point with little accuracy loss to reduce computation complexity and memory requirement. A fully-pipelined dataflow architecture is proposed to accelerate the typical layers (*i.e., convolutional, pooling, residual, inception, and activation layers*) in CNNs. Experimental results show that OPU is 9.6× faster than GPU Jetson TX2 on a cascaded of three CNNs, which are used for the curbside parking system.

However, 8-bit fixed-point data representation always need re-training to maintain accuracy for *deep* CNNs. In this way, we propose a low precision (8-bit) floating-point (LPFP) quantization method for FPGA-based acceleration to overcome the above limitation. Without any re-training, LPFP finds an optimal 8-bit data representation with negligible top-1/top-5 accuracy loss (within 0.5%/0.3% in our experiments, respectively, and significantly better than existing methods for *deep* CNNs). Furthermore, we implement one 8-bit LPFP multiplication by one 4-bit multiply-adder (MAC) and one 3-bit adder. Therefore, we can implement **four** 8-bit LPFP multiplications using one DSP48E1 of Xilinx Kintex-7 family or one DSP48E2 of Xilinx Ultrascale/Ultrascale Plus family whereas one DSP can only implement **two** 8-bit fixed-point multiplications. Experiments on six typical CNNs for inference show that on average, we improve throughput by 1.5× over existing FPGA accelerators. Particularly for VGG16 and Yolo, compared with seven FPGA accelerators, we improve average throughput by 3.5× and 27.5× and average throughput per DSP by 4.1× and 5×, respectively.

CNNs quantized with mixed precision, on the other hand, benefits from low precision while maintaining accuracy. To better leverage the advantages of mixed precision, we propose a Mixed Precision FPGA-based Overlay Processor (MP-OPU) for both conventional and lightweight CNNs. The micro-architecture of MP-OPU considers sharing of computation core with mixed precision weights and activations to improve computation efficiency. In addition, run-time scheduling of external memory access and data arrangement are optimized to further leverage the advantages of mixed precision data representation. Our experimental results show that MP-OPU reaches 4.92 TOPS peak throughput when implemented on Xilinx VC709 FPGA (with all DSPs configured to support 2-bit multipliers). Moreover, MP-OPU achieves 12.9× latency reduction and 2.2× better throughput per DSP for conventional CNNs, while 7.6× latency reduction and 2.9× better throughput per DSP for lightweight CNNs, all on average compared with existing FPGA accelerators/processors, respectively.

Graph convolutional networks (GCNs) have been introduced to effectively process non-

euclidean graph data. However, GCNs incur large amount of irregularity in computation and memory access, which prevents efficient use of previous CNN accelerators/processors. In this way, we propose a lightweight FPGA-based accelerator, named LW-GCN, to tackle irregularity in computation and memory access in GCN inference. We first decompose the main GCN operations into Sparse Matrix-Matrix Multiplication (SpMM) and Matrix-Matrix Multiplication (MM). Thereafter, we propose a novel compression format to balance workload across PEs and prevent data hazards. In addition, we quantize the data into 16-bit fixed-point and apply workload tiling, and map both SpMM and MM onto a uniform architecture on resource limited devices. Evaluations on GCN and GraphSAGE are performed on Xilinx Kintex-7 FPGA with three popular datasets. Compared with existing CPU, GPU and state-of-the-art FPGA-based accelerator, LW-GCN reduces latency by up to 60×, 12× and 1.7× and increases power efficiency by up to 912×, 511× and 3.87×, respectively. Moreover, compared with Nvidia's latest edge GPU Jetson Xavier NX, LW-GCN achieves speedup and energy savings of 32× and 84×, respectively.

At last, we extend our GCN inference accelerator to a GCN training accelerator, called SkeletonGCN. To better fit the properties of GCN training, we add more software-hardware co-optimizations. First, we simplify the non-linear operations in GCN training to better fit the FPGA computation, and identify reusable intermediate results to eliminate redundant computation. Second, we optimize the previous compression format to further reduce memory bandwidth while allowing efficient decompression on hardware. Finally, we propose a unified architecture to support SpMM, MM and MM with transpose, all on the same group of PEs to increase DSP utilization on FPGA. Evaluations are performed on Xilinx Alveo U200 board. Compared with existing FPGA-based accelerator on the same network architecture, SkeletonGCN can achieve up to 11.3× speedup while maintaining the same training accuracy with 16-bit fixed-point data representation. In addition, SkeletonGCN is 178× and 13.1× faster than state-of-the-art CPU and GPU implementation on popular datasets, respectively.

To summarize, we have been working on FPGA-based acceleration for deep learning algorithms of CNNs and GCNs in both inference and training process. All the accelerators/processors were hand-coded and have been fully verified. In addition, the related tool chains for generating golden results and running instructions for the accelerators/processors have also been finished.

The dissertation of Chen Wu is approved.

Sudhakar Pamarti

Puneet Gupta

Yong Chen

Lei He, Committee Chair

University of California, Los Angeles

2022

To my family

TABLE OF CONTENTS

1 Introduction **1**

1.1 Background 1

1.1.1 The Rapid Pace of Deep Learning 1

1.1.2 Challenges of Deploying Deep Learning Algorithms 1

1.2 Motivation 2

1.2.1 Data Compression 2

1.2.2 FPGA-based Acceleration 3

1.3 Organization 3

2 OPU: An FPGA-based Overlay Processor for Convolutional Neural Networks **6**

2.1 Introduction 6

2.2 Background and Related Work 8

2.2.1 CNN 8

2.2.2 Quantization 8

2.2.3 Related Works 9

2.3 Micro-architecture 10

2.3.1 Computation Core 10

2.3.2 Post Process 13

2.3.3 Memory Management 13

2.3.4 Instruction Control 14

2.4 Experimental Results 16

2.4.1 Experiment Setup 16
2.4.2 Runtime MAC Efficiency 17
2.4.3 Comparison with Existing FPGA Accelerators 18
2.4.4 Case Study of Cascaded CNNs 18
2.5 Conclusions and Discussions 19
3 LPFP: Low Precision Floating-point Arithmetic for High Performance FPGA-based CNN Acceleration 20
3.1 Introduction 20
3.2 Background and Motivation 23
3.2.1 Background: Low Precision Floating-point 23
3.2.2 Motivation 24
3.3 Low Precision Floating-point Quantization 25
3.3.1 Quantization Process 25
3.3.2 Data Flow in Processor 27
3.4 Processor Architecture 29
3.4.1 Overview 29
3.4.2 Floating-point Function Unit 30
3.4.3 Memory System 33
3.4.4 Central Control 35
3.5 Evaluation 35
3.5.1 Evaluation of Quantization Method 35
3.5.2 Evaluation of Hardware Implementation 38
3.6 Related Work 46

3.7 Conclusion 48

4 MP-OPU: A Mixed Precision FPGA-based Overlay Processor for Convolutional Neural Networks 49

4.1 Introduction 49

4.2 Background and Related Work 51

4.2.1 Low Precision Quantization 51

4.2.2 Fixed Precision Processors/Accelerators 52

4.2.3 Mixed Precision Processors/Accelerators 52

4.3 Micro-Architecture of **MP-OPU** 52

4.3.1 Computation Core 53

4.3.2 Data Pre-fetch and Placement for on-chip Memory 56

4.4 Experiments 57

4.4.1 Experimental Setup 57

4.4.2 Hardware Implementation 58

4.4.3 Comparison with Customized FPGA Accelerators 58

4.4.4 Discussion 59

4.5 Conclusions 61

5 LW-GCN: A Lightweight FPGA-based Graph Convolutional Network Accelerator 62

5.1 Introduction 62

5.2 Challenges and Motivations 66

5.2.1 GCN Background 66

5.2.2 Challenges 67

5.2.3 Motivation 70
5.3 Software Preprocessing 72
5.3.1 Data compression 72
5.3.2 Assignment and Scheduling 75
5.4 Micro-architecture of LW-GCN 78
5.4.1 Overall Workflow 79
5.4.2 Multi-Bank Dense Data Memory (DDM) 80
5.4.3 Unified PE architecture for MM and SpMM 81
5.5 Evaluation 82
5.5.1 Experiment Design 83
5.5.2 Hyper Parameter Impact 83
5.5.3 Latency Breakdown 87
5.5.4 Overall Comparison 88
5.5.5 Extending LW-GCN to Other Algorithms 90
5.6 Conclusions and Future Work 91

6 SkeletonGCN: A Simple Yet Effective Accelerator for GCN Training . . 92
6.1 Introduction 92
6.2 Background and Related Work 95
6.2.1 Workload Breakdown 95
6.2.2 Low Precision Training 97
6.2.3 FPGA-based Accelerators 98
6.3 Optimized Training 99
6.3.1 Training Simplifications 99

6.3.2 Hardware-aware Compression . . . 100
6.4 Hardware Architecture . . . 102
6.4.1 Overall Architecture . . . 102
6.4.2 Unified PE Architecture . . . 103
6.4.3 Weight Update . . . 106
6.4.4 Data Communication . . . 107
6.4.5 Allocation and Scheduling . . . 107
6.5 Experimental Results . . . 109
6.5.1 Experimental Setup . . . 109
6.5.2 Training Accuracy and Latency . . . 110
6.5.3 Comparison with State-of-the-art . . . 111
6.5.4 Discussion . . . 112
6.6 Conclusion and Future Work . . . 114
7 Summary . . . 115
References . . . 118

LIST OF FIGURES

2.1 Overall micro-architecture of **OPU**. 11

2.2 (a) Conventional intra-kernel based parallelism. (b) **OPU** channel based parallelism. (c) Data fetch pattern in **OPU**. 12

2.3 Data arrangement in off-chip memory and on-chip memory. 14

3.1 Computation complexity and memory requirement with respect to different CNNs. 21

3.2 The data flow in our processor with $M4E3$ data format as an example (FP: floating-point, Mult: LPFP multiplier, AM: alignment module, Acc: accumulator, DC: data converter). 28

3.3 The overall architecture of proposed processor. 30

3.4 The architecture of a PE and the parallel computation pattern in a PE. MUL: LPFP multiplier, AM: alignment module, ACC: accumulator, Act: activation. . 32

3.5 Top-1/Top-5 accuracy for different (mantissa, exponent) combinations with respect to different CNNs. 36

3.6 Top-1 and top-5 accuracies for different bit width with respect to different CNNs. 38

3.7 Data format of the DSP to implement four 4-bit MACs. M_a, M_b, M_c and M_d: the mantissas of LPFP data a, b, c and d, respectively; $Ex_{M_aM_c}, Ex_{M_aM_d}, Ex_{M_bM_c}$ and $Ex_{M_bM_d}$: the extra term expressed as $Ex_{M_aM_c} = 1.M_a + 0.M_c$; P_{ac}, P_{ad}, P_{bc} and P_{bd}: the mantissas of the product of two LPFP data expressed as $P_{ac} = 1.M_a \times 1.M_c$. 39

3.8 Parallel exploration with respect to throughput and bandwidth requirement. . . 42

4.1 The micro-architecture of **MP-OPU**. The computation core has multiple PEs for convolutional layers followed by a post process module for other types of layers. The memory system is designed with on-chip buffers in ping-pong manner to save communication time with the external memory. 53

4.2 The architecture of one PE. 54

4.3 Four levels of parallelism in **MP-OPU**. 56

4.4 Inference latency of Tiny-Yolo-V3 with respect to different combinations of bit width. In the legends, "w=2" means the bit width of weights is 2. 60

5.1 Packet-level column-only coordinate list format 72

5.2 Outer product matrix multiplication 75

5.3 Round-robin assignment of non-zero elements to four PEs 77

5.4 The overall micro-architecture and workflow of **LW-GCN** 79

5.5 (a) The architecture of PE Array; (b) Detailed architecture of a PE. 81

5.6 Impact of (a) dense data replication with 512-row tiles and (b) tile size with 1 replica 84

5.7 Resource consumption of (a) replication and (b) tile size 86

5.8 Latency breakdown for (a) full execution and (b) SpMM 88

5.9 PE utilization during SpMM for Cora: (a) first combination tile and (b) first aggregation tile 88

6.1 Packet-level column-only coordinate list format 100

6.2 Overall architecture of **SkeletonGCN**. 102

6.3 (a) The architecture of the unified PE. (b) The architecture of the data distribution module. (c) The architecture of the L2 normalization and gradients module. 104

6.4 An example of allocating SpMM, MM or MM with transpose onto 2 *MACC Arrays*. The notion "L_R#" indicates the row index of the left matrix while the notion "R_R#" indicates the row index of the right matrix. 106

6.5 Scheduling between CPU and FPGA. 108

6.6 Training Accuracy Comparison 110

6.7 The DSP efficiency of computing SpMM, MM, and MM with transpose in training GCN on reddit. The MM with transpose is marked as "TMM" in the figure. . . 113

LIST OF TABLES

2.1 Accuracy evaluation for quantized networks. We report validation accuracy (%) and mean average precision (%) for classification and detection networks, respectively. 9

2.2 Resource Utilization on KC705 16

2.3 Network Configurations 16

2.4 RME of **OPU** for different CNNs, (B) indicates evaluated with batch size of 8. . 17

2.5 Comparison with FPGA accelerators, C: convolutional layer only. 18

2.6 Latency comparison with Jetson TX2 GPU on cascaded CNNs. 19

3.1 Resource utilization of multipliers on FPGA for different data representations. DSP: digital signal processing, LUT: look-up table, FF: flip-flop. $M4E3$: 1-bit sign, 4-bit mantissa and 3-bit exponent. 22

3.2 Accuracy comparison between $M4E3$, $M5E2$, references and FP32. "-" means no reported results. "R" means the method with retraining. 37

3.3 Resource Utilization on XC7K325T. 41

3.4 Resource Utilization on Ultrascale+ 7EV. 42

3.5 Comparison between Intel I7-8700T CPU, Nvidia Xavier NX GPU, existing accelerators and our processor with respect to different CNNs. "-" means no reported results. 43

3.6 Comparison with prior accelerators on VGG16. "-" means no reported results. . 44

3.7 Comparison with prior accelerators on YOLO. "-" means no reported results. . . 46

4.1 Comparison with customized FPGA accelerators/processors on conventional CNNs. 57

4.2 Resource Utilization of **MP-OPU** on XC7VX690T. 58

4.3 Comparison with customized FPGA accelerators/processors on MobileNetV1/V2. 59

5.1 Dimensions and densities of widely-used datasets. 65

5.2 Required computation and storage under different computation orders. 67

5.3 Comparison of Storage Requirement between CSR, CSC, COO and PCOO. . . . 73

5.4 Preprocessing time for different datasets. 84

5.5 Resource utilization on Kintex-7 325T FPGA. 86

5.6 Comparison with CPU, edge GPU, general GPU and existing FPGA accelerator on GCN and GraphSAGE 89

6.1 Dimensions, Densities and Workloads Across Datasets. 100

6.2 Compressed matrix sizes across datasets and algorithms (random-walk sampled subgraphs [ZZS19b]) 101

6.3 Resource Utilization on Alveo U200 Board. 110

6.4 Training Latency Comparison. 111

6.5 Comparison with GraphACT. 112

ACKNOWLEDGMENTS

I would like to thank my supervisor Lei He for all his continuous support and patience during my PhD study. Also thanks to my committee members, Yong Chen, Puneet Gupta and Sudhakar Pamarti, who offered guidance and support. I would also like to thank Kun Wang for his help on my papers and thanks to Zhuofu Tao for the help on the work of GCN. Finally, many thanks to my wife, daughter, parents and numerous friends who endured this long process with me, always offering support and love.

VITA

2008–2012	B.S., Electronic Science and Technology, School of Electronic Science and Engineering, University of Electronic Science and Technology of China, Chengdu, Sichuan, China.
2012–2015	M.S., Electronic Science and Technology, School of Integrated Circuits, Tsinghua University, Beijing, China.
2015–2016	Software Engineer, Shanghai Huawei Technologies Co. Ltd, Shanghai, China.
2016–present	PhD program, Department of Electrical and Computer Engineering, University of California, Los Angeles, California, USA.
2017–2019	Teaching Assistant.
2017	Internship, Fudan Microelectronics Group, Shanghai, China.
2018	Internship, Fudan Microelectronics Group, Shanghai, China.
2019–2021	Internship, Nunova Technologies Co. Ltd, Chengdu, Sichuan, China.

PUBLICATIONS

Yunxuan Yu, **Chen Wu,** et. al., "OPU: An fpga-based overlay processor for convolutional neural networks." *IEEE Transactions on Very Large Scale Integration(VLSI) Systems,* 2019, 28(1), pp. 35-47.

Yunxuan Yu, **Chen Wu,** et. al., “Overview of a FPGA-based overlay processor.” *China Semiconductor Technology International Conference (CSTIC), IEEE*, 2019.

Chen Wu, Mingyu Wang, et. al., “Low precision floating-point arithmetic for high performance FPGA-based CNN acceleration.” *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* 2020, pp. 318-318.

Chen Wu, Mingyu Wang, et. al., “Low precision floating-point arithmetic for high performance FPGA-based CNN acceleration.” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2021, 15(1), pp. 1-21.

Chen Wu, Jinming Zhuang, et. al., “MP-OPU: A mixed precision FPGA-based overlay processor for convolutional neural networks.” *2021 31st International Conference on Field-Programmable Logic and Applications (FPL).*, IEEE, 2021, pp. 33-37.

Zhuofu Tao, **Chen Wu,** et. al., “LW-GCN: A lightweight FPGA-based graph convolutional network accelerator.” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2021.

Chen Wu, Zhuofu Tao, et. al., “SkeletonGCN: A simple yet effective accelerator for GCN training.” *2022 32rd International Conference on Field-Programmable Logic and Applications (FPL).* IEEE, 2022.

CHAPTER 1

Introduction

1.1 Background

1.1.1 The Rapid Pace of Deep Learning

Deep learning algorithms have been demonstrated a breakthrough in performance for a broad range of applications and have brought artificial intelligence (AI) into our daily life. The smart phones, autonomous driving cars, web-scale recommenders have made our life easier.

Among all the deep learning algorithms, convolutional neural networks (CNNs) have shown great success in various families of tasks in the euclidean data domain, such as object classification, object detection and segmentation tasks. Starting from 2012, CNNs have reduced the top-5 error for ImageNet classification from 17% (AlexNet in 2012) to 2.9% (EfficientNet-B7 in 2019). In the non-euclidean data domain, such as graphs, graph neural networks (GNNs) have been introduced and have demonstrated the ability to accurately process complex graph data. Among numerous GNNs, graph convolutional networks (GCNs), which borrows ideas from CNNs to aggregate neighbor data, have quickly attracted industrial attention as a popular solution to real-world problems.

1.1.2 Challenges of Deploying Deep Learning Algorithms

CNNs and GCNs have great success on the prediction accuracy front, however, they always require a large amount of memory and computation. Although algorithm designers have

been reducing parameters in CNNs, the number of parameters of a CNN still stays large at 264 MB. Moreover, the requirement of storage goes higher for GCNs, as the datasets can be as large as gigabytes. Such great storage requirements makes fitting CNNs or GCNs onto hardware challenging, especially for resource-limited devices.

On the other hand, the computation complexity keeps increasing to achieve better prediction accuracy. For example, the computation complexity of a feed-forward process of a 224×224 RGB image increases from 2.27 GOP of AlexNet in 2012 to 74 GOP of EfficientNet-B7 in 2019. This calls for specific hardwares to accelerate computation because general CPUs may take seconds or even minutes to run one network, which is difficult to meet the constraints in real scenarios. In addition, the irregular computation patterns caused by Sparse Matrix-Matrix Multiplication (SpMM) in GCNs also requires specific hardwares as general purpose hardwares are not efficient.

1.2 Motivation

1.2.1 Data Compression

The huge amount of data is difficult to be fitted into hardwares, therefore, data compression is needed. Previous quantization schemes have demonstrated that 8-bit fixed-point data representation is effective for CNNs [ORK15a]. However, they are not effective for *deep* CNNs or need re-training to compensate for the quantization error. In this way, we explore further for different data representations and quantization schemes to compress CNNs while maintaining accuracy. We first propose a low precision floating-point data representation to quantize *deep* CNNs, which will be discussed in detail in Chapter 3. In addition, we explore more on the properties of different layers and use mixed precision for different layers, so that we can further compress the CNNs and keep accuracy. Details can be found in Chapter 4.

We also consider quantization for GCNs, both in inference phase and training phase. However, the sparse matrices in GCNs are always as large as megabytes but only less than

0.1% of which are valuable (non-zeros) elements. Therefore, we propose an effective compression method to compress such sparse matrices to only store and compute the non-zero elements. Details will be discussed in Chapter 5 and 6.

1.2.2 FPGA-based Acceleration

FPGA-based accelerators for CNNs have been widely explored [ZSF16], showing a promising solution to CNN acceleration. However, these accelerators only target on a certain network and it is difficult to expand to other networks. In this way, we first propose an FPGA-overlay processor to support a wide range of CNNs without changing the hardware design, as discussed in Chapter 2. Based on the low precision floating-point and mixed precision data representation for CNNs, we also develop specific FPGA-based processors to further leverage the advantages of different data representation, as explained in Chapter 3 and 4.

CNN accelerators/processors is not effective for GCNs, especially for SpMM used in GCNs. Although previous work tries to overcome the sparseness challenges [LWL20b, GLS20], they either require large on-chip buffers or use independent hardware modules for different operations, which are not efficient for hardwares, especially for resource-limited devices. To this end, we propose a lightweight GCN inference accelerator to tackle the above challenges, as discussed in detail in Chapter 5. We also extend our GCN inference accelerator to support GCN training to better resolve the challenges in GCNs, which will be discussed in Chapter 6.

1.3 Organization

This dissertation mainly focuses on the research on architecture level and algorithm level optimizations to accelerate deep learning algorithms on FPGA. The remaining parts of this dissertation are organized as follows:

- **Chapter 2: OPU: An FPGA-based overlay processor for convolutional neural networks.**

  A domain-specific FPGA-based overlay processor, called **OPU**, is first proposed to accelerate inference for various CNNs. 8-bit fixed-point data representation is utilized in **OPU** to reduce computation complexity and memory requirement. Based on this data representation, a fully-pipelined architecture is developed to accelerate the inference phase of CNNs. We also implement **OPU** on a real world application with cascaded CNNs to show the effectiveness of **OPU**.

- **Chapter 3: Low precision floating-point arithmetic for high performance FPGA-based CNN acceleration.**

  We study further into the quantization algorithm, and propose a low precision floating-point (LPFP) quantization method to quantize *depp* CNNs without re-training while maintaining accuracy. We further develop an FPGA-based processor, where one DSP slice along with several LUTs are decomposed into four LPFP multipliers, to further leverage the advantages of LPFP data representation.

- **Chapter 4: MP-OPU: A mixed precision FPGA-based overlay processor for convolutional neural networks.**

  Since mixed precision CNNs can further reduce memory requirement while maintaining accuracy, we propose a mixed precision FPGA-based overlay processor (**MP-OPU**) to leverage the advantages brought by mixed precision. We focus on the micro-architecture design to support mixed precision weights and activations during runtime. In addition, we explore the parallel computation pattern of different layers in CNNs.

- **Chapter 5: LW-GCN: A lightweight FPGA-based graph convolutional network accelerator.**

  CNNs work efficiently for euclidean data while GCNs process non-euclidean data efficiently. However, GCNs incur large amount of irregular computation and memory

access because of sparse matrix multiplication, which is not efficient for CNN processors. In this way, we propose a lightweight FPGA-based (**LW-GCN**) accelerator for GCNs to tackle the irregularity in computation and memory access. We first develop a compression format to efficiently compress the sparse matrix while effectively decompress on hardware. Moreover, a unified architecture, which support both SpMM and MM, is designed to accelerate the inference phase of GCNs.

- **Chapter 6: SkeletonGCN: A simple yet effective accelerator for GCN training.**

  We extend our GCN inference accelerator to support GCN training. The compression method is first improved to further reduce storage. In addition, non-linear functions are simplified in algorithm level to better fit the FPGA computation, and intermediate results are identified to be reused to reduce redundancy computation. Finally, the unified architecture is extend to support MM with transpose on the same group of PEs so that the DSP utilization is optimized.

- **Chapter 7: Summary.**

  The summary of this dissertation is discussed in this chapter.

CHAPTER 2

OPU: An FPGA-based Overlay Processor for Convolutional Neural Networks

2.1 Introduction

Convolutional neural networks (CNNs) have demonstrated a breakthrough in performance for a broad range of applications including object recognition [SLJ15c], object detection [RDG16b] and speech recognition [AAA16]. However, CNNs often have huge computation complexity. This motivates accelerating CNNs by CPU/GPU clusters [DCM12b], FPGAs [ORK15b] and ASICs [CDS14]. Customized accelerators on FPGAs [ORK15a, ZLS15b], which leverage full capacity of parallelism, have shown more promising throughput and power efficiency than traditional CPU/GPU clusters [ZSF16, WYZ17b].

However, implementing a high-performance FPGA accelerator can be time-consuming as it always involves parallel exploration, bandwidth optimization, area and timing tuning. This leads to the development of accelerator generators, where hardware description codes of the target accelerators are generated automatically [ZWZ18a, WYZ17a]. In these designs, they simplify the design space exploration to parameter optimization for existing modules by designing a parameterized template architecture.

However, the above accelerator generators still have challenges in design CNN accelerators. Since the outputs are always RTL codes, they still need synthesis, placement and routing to obtain the final bitstream, which always takes hours to finish and may face timing violation problems. Instead of fixing violating paths as in regular FPGA design, user can

only adjust module parameters or alleviate timing constraints at the expense of performance degradation in these automatic compilers. Moreover, nowadays complex deep learning tasks usually involve cascaded networks, which is inefficient to constantly re-burn FPGA for running different networks in series.

To this end, we propose an FPGA-based overlay processor (**OPU**) for general CNN acceleration. **OPU** has fine-grained pipeline and explores parallelism of different CNN architectures, which ensures 91% runtime utilization of computation resources on average across 9 commonly used CNNs. Moreover, we provide a domain-specific instruction set to support different CNN configurations. In this way, once a new network configuration is given, instead of re-generating a new accelerator on FPGA, we just compile the network into instructions to be executed on **OPU**. Experiments also show that **OPU** has **OPU** is 9.6× faster than GPU Jetson TX2 with similar amount of computing resources for cascaded networks.

To conclude, the main contributions can be summarized as follows:

- **High flexibility.** Controlled by parameter registers, **OPU** is flexible to run different CNNs without any change of the hardware.

- **Fully pipelined micro-architecture.** The micro-architecture of **OPU** is optimized for computation, communication and data reorganization. Moreover, the micro-architecture is designed fully pipelined to improve performance.

- **High performance.** Comprehensive experiments are performed on **OPU**. It shows on average 91% runtime MAC efficiency across 9 different CNNs. Moreover, **OPU** shows 9.6× speedup than Jetson TX2 GPU on realtime cascade CNNs.

2.2 Background and Related Work

2.2.1 CNN

CNNs are used to classify or recognize objects by passing the inputs through multiple types of layers. In each layer, multiple neurons are constructed to process different inputs and pass the outputs to the next layer through connections, and the connections are used to store the weights for the network. Based on different processing procedures, the layers are typically divided into *convolutional, pooling, activation, normalization, fully-connected, residual* and *inception* layers. Among them, *convolutional/fully-connected* layers consume most portions of computation while *fully-connected* layers require largest memory to store weights. In this way, we treat *convolutional/fully-connected* layers as major layers while other layers as minor layers. In most CNNs, one major layer is always followed by several minor layers.

2.2.2 Quantization

It has been proven that CNNs are robust against precision reduction [QWY16, GSQ18]. To reduce memory footprint and save computational resources, we select 8-bit fixed-point data representation for both weights and feature maps in **OPU**. We employ a fast yet effective stationary quantization method which does not require any calibration, fine-tuning or re-training. Similar to that in [QWY16, MCV17b], we utilize dynamic quantization to maintain accuracy. In this way, the process of finding the best range for each trunk of data is described as follows:

$$\underset{floc}{\operatorname{argmin}} \sum (V_{fp} - V_{fix}^{flen})^2, \tag{2.1}$$

where V_{fp} is the original floating-point data, while V_{fix}^{flen} is the fixed-point data quantized based on the fraction length $flen$. Based on the dynamic quantization scheme, we can maintain the accuracy loss within 1% averagely when evaluated on 9 commonly used networks,

Table 2.1: Accuracy evaluation for quantized networks. We report validation accuracy (%) and mean average precision (%) for classification and detection networks, respectively.

	Classification Networks							Detection Networks	
	VGG16	VGG19	InceptionV1	InceptionV2	InceptionV3	ResNet50	ResNet101	YoloV2	TinyYolo
Float32	89.8	85.2	87.3	90.3	93.2	92.9	93.7	85.9	90.8
INt8	89.4	84.3	85.5	89.9	91.4	92.3	93.2	86.2	89.5

as shown in Table 2.1.

2.2.3 Related Works

FPGA-based acceleration for deep learning algorithms has been extensive studied, starting with customized accelerators for specific network. Farabet [FPH09] uses FPGA as a vectorial arithmetic unit, and implements CNN mainly on a 32 bit soft processor for flexibility. Accelerators for each layer of CNN are proposed in [ORK15a, CMB10, CSJ10]. The whole network AlexNet is implemented on a VC707 board with high throughput by using HLS [ZLS15b], and hand-coded RTL accelerator for VGG16 is proposed in [QWY16]. The aforementioned work demonstrates the capability of developing high performance accelerator on FPGA, however, manually designing accelerator for each CNN is inefficient.

More recent studies focus on automatic compiler to generate CNN accelerators on FPGA. Deepweaver [SPM16] is proposed to map CNNs into hand-optimized design templates to achieve comparable performance with hand-crafted accelerators. The Caffeine is proposed as a high level synthesis (HLS) based compiler which optimizes bandwidth by memory access reorganization [ZFZ16]. However, the above work aims at generating specific accelerators for a specific CNN, which comes with high efforts of re-engineering hardware when the target CNN updates.

The work in [AHB18] explores FPGA overlay to implement CNN accelerators. They use instructions to decrease the overhead of control logic and reconfigure the overlay architecture

to maximize performance for different CNNs. Moreover, they use coarse grained instruction sets, for example, one block of 10 instructions are used for a single convolutional layer and an optional pooling layer. In contrast, **OPU** uses fine grained instruction sets to represent each typical operation with one specific instruction. In addition, we design a FPGA-based processor for different CNNs without FPGA reconfiguration.

2.3 Micro-architecture

The challenge for designing the micro-architecture of **OPU** is to incur small control overhead while maintain easily runtime adjustable and functionality. In this way, we design parameter registers, which can be configured directly by instructions, to customize and reconfigure the modules during runtime. Moreover, the computation core explores multiple levels of parallelism that generalize well among different network configurations. At the same time, the minor operations are combined so that they can be accomplished by the same module to reduce overhead.

As shown in Fig. 2.1, the overall micro-architecture of **OPU** can be decomposed into six main modules. Each module is controlled by one corresponding instruction (see details in 2.3.4) to accomplish the functionality. Besides, the memory system, which includes four on-chip buffers and one off-chip memory, are optimized to overcome the bandwidth constraint issue. With most of the control flow embedded in instruction, **OPU** only handles the computation of one tile. If the input size of one layer is larger than the tile size, the layer will be sliced into different tiles to fit into the hardware.

2.3.1 Computation Core

There are several parallelism levels (*i.e.,* parallel in input channel, output channel, kernel) for computing convolutional layer. In this way, how to use the same architecture of the computation core to accommodate parallel computing for layers with different configurations

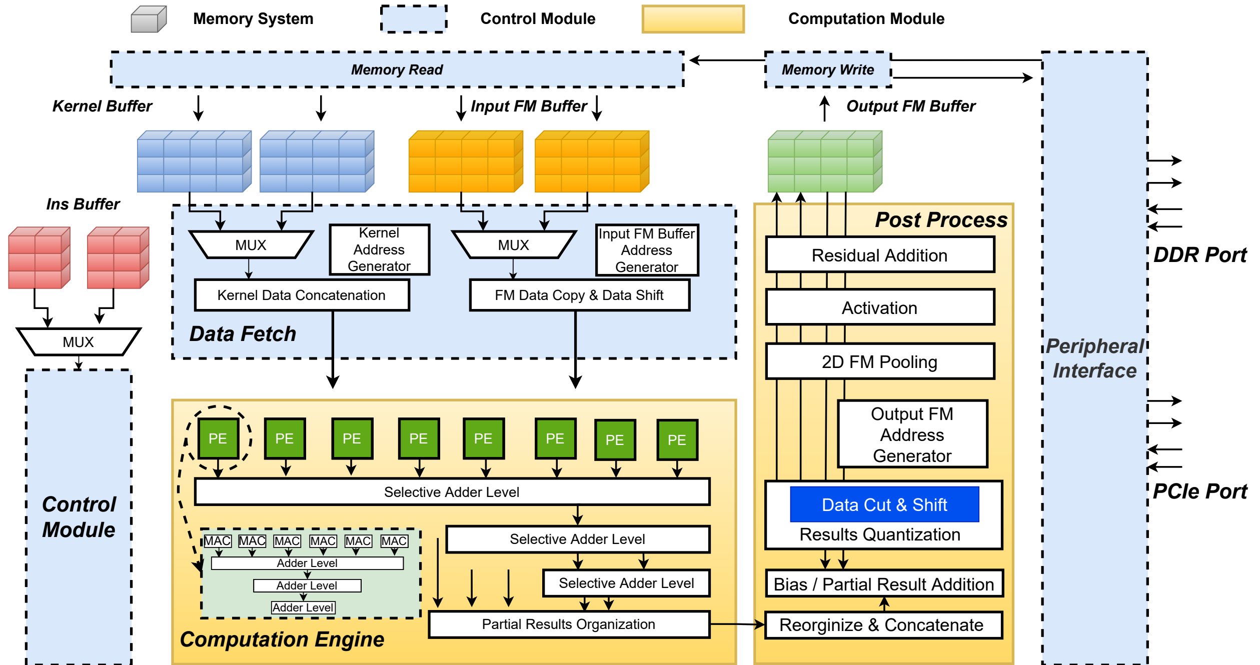


Figure 2.1: Overall micro-architecture of **OPU**.

is the biggest challenge in **OPU**. Conventional designs explore parallelism within the 2-dimension (2D) kernel, which is straightforward but comes with the disadvantages of complex data management of feature map and poor generalization among various kernel sizes. As shown in Fig. 2.2(a), expanding a $k_x * k_y$ kernel sized window of feature map requires row and column direction data fetch in single clock cycle (step ①). This poses challenges on data arrangement in on-chip memories and generally requires extra resources for data reuse (*i.e.*, line buffer). Moreover, customized data management logic for one kernel size cannot be efficient for other kernel sizes. Similarly, the computation core optimized for one kernel size may not fit other sizes efficiently. This is why many conventional FPGA accelerators, which is optimized for 3×3 kernels, perform the best only on networks with pure 3×3 kernel size.

To address this issue, we explore channel based parallelism and leave the 2D kernel being computed sequentially, which makes **OPU** fit well for any kernel size. Moreover, both the sizes of input channel and output channel in CNNs are always multiples of 16. This feature

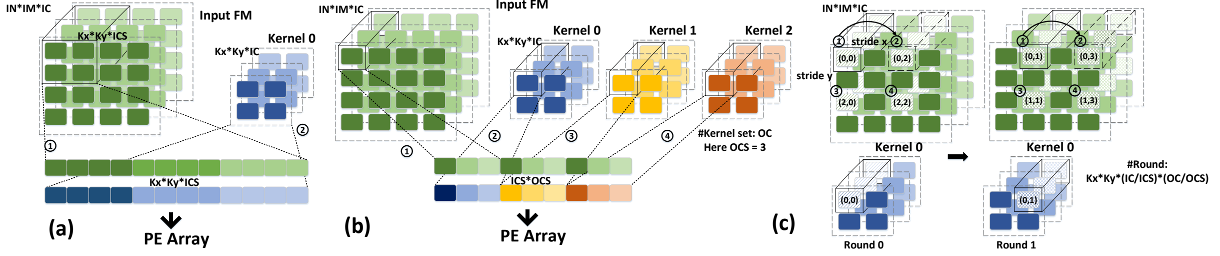


Figure 2.2: (a) Conventional intra-kernel based parallelism. (b) **OPU** channel based parallelism. (c) Data fetch pattern in **OPU**.

makes **OPU** suitable for different network configurations with different input/output channel sizes. The working flow in **OPU** is explained in Fig. 2.2(b) with an example. At each clock cycle, a slice of input channel of 1, 1, IC_p^i in width, height and depth, respectively, is read. In addition, the corresponding kernel elements in OC_p^i output channels are read so that parallelism is implemented within input channel slice IC_p^i and output channel slice OC_p^i. This fits natural data storage pattern and requires much smaller bandwidth. Fig. 2.2(c) further shows the computation process. For round 0 cycle 0, input feature map channel slice from position $(0, 0)$ is read. We then jump stride x ($x = 2$ in this example) and read position $(0, 2)$ in next cycle. Read operation continues until all pixels corresponding to kernel position $(0, 0)$ is fetched out and computed. After that we enter round 1 and read from position $(0, 1)$ to get all pixels corresponding to kernel position $(0, 1)$.

With fixed number of multiply-adders (MACs) in the computation core, we implement parallelism with different input and output channel combinations. In this way, **OPU** can fit flexibly into different convolutional layers with different input/output channels combinations. In our current implementation with totally 1024 MACs, we can support 6 $[in_c, out_c]$ pairs: [512, 2], [256, 4], [128, 8], [64, 16], [32, 32] and [16, 64].

Our computing pattern guarantees uniform data fetching pattern for any kernel size or stride, which greatly simplifies data management and enables higher working frequency with less resource consumption. Moreover, we leverage both input and output channel level

parallelisms. This provides higher flexibility for resource utilization and promises reasonable generalization performance.

2.3.2 Post Process

We perform the major layers in the computation core and all the minor layers in the post process module. Since one major layer is always followed by a set of minor layers in series, we design the fully pipelined post process module to go through all the operations in minor layers so that extra on-chip data movement is reduced. In addition, data quantization, concatenation in output channel and partial sum addition are performed in this module, as only one tile is computed in the computation core. Since the different computation pattern will produce [2, 4, 8, 16, 32, 64] output channels, we set the parallelism in the post process module 64 to later process. To conclude, the detailed architecture of the post process module is shown in the right part of Fig. 2.1.

2.3.3 Memory Management

Another crucial issue for CNN acceleration on FPGA is the bandwidth constraints of the off-chip memory. Roof-line model [ZLS15a] reveals the relationship between bandwidth utilization and computational roof performance and the bandwidth can easily become the bottle neck of performance. In this way, we utilize a ping-pong structured on-chip buffer to hide the off-chip communication latency under computation. While data is fetched from one buffer for computation, the other buffer can get refilled and updated, which maintains the maximum bandwidth utilization.

The data arrangement in both the off-chip memory and on-chip buffers also influence the communication latency greatly. Considering that the most efficient way for accessing off-chip memory is to fetch data in continuous addresses, we arrange as many data of one tile in continuous addresses as possible. Moreover, a slice of input channel is computed in

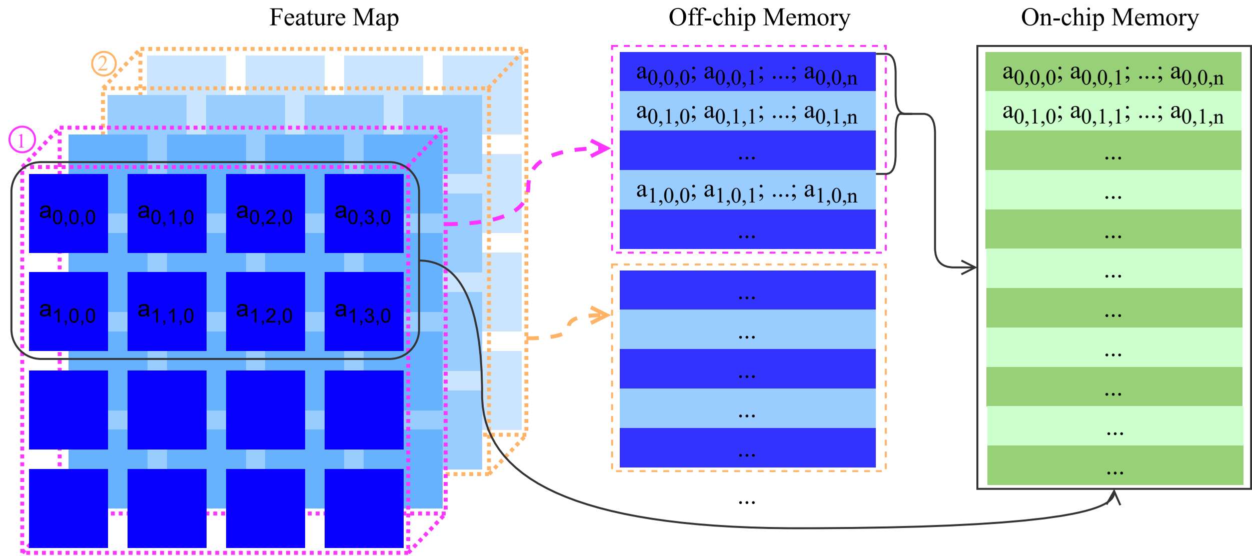


Figure 2.3: Data arrangement in off-chip memory and on-chip memory.

parallel in the computation core, we need to store them under the same address so that they can be accessed in one clock cycle. To this end, we arrange the data of one feature map in channel-column-row pattern in the off-chip memory, as shown in Fig. 2.3. We store each $W_{fm} \times H_{fm} \times IC^i$ ($IC^i = DW/8$ indicates the bit width of off-chip memory divided by data width used in **OPU**) feature map in continuous addresses in the off-chip memory, shown as ① $->$ ② in Fig. 2.3. To simplify the off-chip communication, we set the bit width of the on-chip buffer the same as that of the off-chip memory. In this way, the IC^i input channel data is stored into one address of the on-chip buffer, and the other data is also stored in a continuous way. Moreover, our data arrangement scheme can support any tile size.

2.3.4 Instruction Control

We propose a domain-specific instruction set for **OPU** to support CNN inference. After identifying all the necessary operations during CNN inference, we design different types of instructions with adjustable parameters for flexibility. The instructions are designed with 32-bit uniform length and variant runtime (up to hundreds of cycles) to process major

operations. To summarize, we have 7 types of instructions:

- *Memory Read* transforms data from external memory to on-chip buffers. It operates in two modes to accommodate for different data read patterns. Received data will be distributed to different destination buffers corresponding to the feature map, kernel weighs and instruction respectively.
- *Memory Write* sends the block of computational results back to external memory.
- *Data Fetch* reads data from on-chip buffers and feed to computation engine. For feature map, we fetch in a rectangular way by setting corresponding parameters, while for kernels, we fetch continuous address.
- *Compute* controls the computation core to work on different computation pattern.
- *Post Process* includes pooling, activation, data quantization, partial sum addition as well as residual operations. Selected combination of before-mentioned operations are executed when *post process* is triggered.
- *Instruction Read* reads a new block of instructions from instruction buffer and sends it to target operation modules.
- *Parameter Setup* follows one of the above 6 instructions to set the corresponding parameter registers.

All these instructions are first fetched and decoded, and control signals for other modules are generated according to the instructions in this module.

Table 2.2: Resource Utilization on KC705

Resource	LUT	FF	BRAM	DSP
Used	94763	150848	165	516
Available	203800	407600	445	840
Utilization	46.50%	37.01%	37.08%	61.43%

Table 2.3: Network Configurations

	VGG16	VGG19	InceptionV1	InceptionV2	InceptionV3	ResNet50	ResNet101	YoloV2	TinyYolo
Input size	224x224	224x224	224x224	224x224	299x299	224x224	299x299	608x608	416x416
Kernel size	3x3	3x3	1x1 3x3 5x5,7x7	1x1,3x3	1x1,3x3 5x5,1x3 3x1,1x7,7x1	1x1 3x3 7x7	1x1 3x3 7x7	1x1,3x3	1x1,3x3
Pool size	2x2	2x2	3x2,3x1,7x1	3x2,3x1,7x2	3x2,3x1,8x2	3x2,1x2	3x2,1x2	2x2	2x2
# Conv layer	13	16	57	69	90	53	53	21	9
Activation	ReLU							Leaky ReLU	
Operations(GOP)	30.92	39.24	2.99	3.83	11.25	6.65	12.65	54.07	5.36

2.4 Experimental Results

2.4.1 Experiment Setup

We implement **OPU** with 1024 MACs on Xilinx KC705 evaluation board, and the resource utilization under 200MHz working frequency is shown in Table 2.2. To verify the effectiveness of **OPU**, 9 commonly used CNNs of different architectures are mapped, with detailed configurations shown in Table 2.3. Among all the networks, YoloV2 and TinyYolo are used for object detection while others are used for image classification. Different kernel sizes from square kernels (1×1, 3×3, 5×5, 7×7) to sliced kernels (1×7, 7×1), various pooling sizes (1×2, 2×2, 3×1, 3×2, 7×1, 7×2, 8×2) and different activation types (ReLU and Leaky ReLU) are used to verify the flexibility of **OPU**.

Table 2.4: RME of **OPU** for different CNNs, (B) indicates evaluated with batch size of 8.

	VGG16	VGG19	InceptionV1	InceptionV2	InceptionV3	ResNet50	ResNet101	YoloV2	TinyYolo
RME(%)	97.2(B)	97.3(B)	90.4(B)	90.5(B)	91.1(B)	84.5	86.9	95.5	89.2
Conv RME(%)	97.0	97.8	90.5	90.8	90.9	84.5	86.9	95.5	89.2
FPS(B)	12.2	9.8	112.5	89.8	30.0	54.4	27.1	7.2	68.3
FPS	11.3	9.4	104.5	84.6	27.3	54.4	27.1	7.2	68.3

2.4.2 Runtime MAC Efficiency

We define the runtime MAC efficiency (RME) to indicate how effective **OPU** is for running different CNNs. The computation of RME is shown in Equ.(2.2).

$$RME = T_{test}/T_{theo}, \tag{2.2}$$

where T_{test} and T_{theo} are the testing and theoretical throughput, respectively. The theoretical throughput is defined in Equ.(2.3), and for our current design, $T_{theo} = 1024 \times 2 \times 200MHz = 409.6GOPS$.

$$T_{theo} = \# \ MAC \times 2 \times frequency. \tag{2.3}$$

We evaluate RME on all the networks and the results are shown in Table 2.4. On average, the overall RME of **OPU** is 91.4% under all the tested networks. The high RME indicates that all computation resources are well-utilized and **OPU** is efficient for running the CNNs. Note that for fully connected layers, high RME is difficult to achieve under non-batch mode because of the bandwidth constraints. In this way, we also report RME under batch mode (batch size is 8), and it only influence the networks with fully connected layers.

Table 2.5: Comparison with FPGA accelerators, C: convolutional layer only.

	[SCD16]	[QWY16]	[XLL17a]	**OPU**	[GSQ18]	**OPU**	
Device	XC7Z045	XC7Z045	XC7Z045	XC7K325T	XC7Z020	XC7K325T	
Network		VGG16			TinyYolo	TinyYolo	YoloV2
# DSP	727(900)	780(900)	824(900)	516(840)	190(220)	516(840)	
Data format	16-bit	16-bit	8-bit	8-bit	8-bit	8-bit	
Frequency(MHz)	120	150	100	200	214	200	
T_{theo} (GOPS)	174	234	329	409.6	162	409.6	
T_{test} (GOPS)	118/137(C)	137/188(C)	230	354/397(C)	62.9	366	391
RME(%)	67/78(C)	58/79(C)	69	**86/97(C)**	39	**89**	**95**

2.4.3 Comparison with Existing FPGA Accelerators

In this section, we compare the performance of **OPU** with existing FPGA accelerators of VGG or Yolo, as shown in Table 2.5. The batch size is set to 1 for fair comparison. As shown in Table 2.5, **OPU** outperforms other automatically compiled network-specific accelerators in terms of RME. To be specific, **OPU** has 86% RME while other accelerators only have 58% to 69% RME for running VGG.

2.4.4 Case Study of Cascaded CNNs

To further evaluate the effectiveness of **OPU**, we evaluate **OPU** on a real-time task to recognize car license plate by running cascaded CNNs. We will first use a car-YoloV3 to detect cars from pictures, and a plate-TinyYolo to detect license plates from cars, and finally a plate-CR to recognize the characters in the license plates. As shown in Table 2.6, we run the cascaded CNNs on both **OPU** and Nvidia Jetson TX2 GPU for comparison. TX2 GPU is running under batch mode (batch size = 5) and the latency is recorded as the average latency for running one image, while **OPU** is tested without using batch. It can be seen from Table 2.6 that **OPU** is faster for running all the three networks compared with TX2

Table 2.6: Latency comparison with Jetson TX2 GPU on cascaded CNNs.

	Frequency(MHz)	T_{theo}(GOPS)	car-YoloV3	plate-TinyYolo	plate-CR	Speedup
Jetson TX2	845	450	607 ms	174 ms	26 ms	1×
OPU	200	409.6	64 ms	19 ms	1.2 ms	9.6×

GPU. Overall, **OPU** is 9.6× faster than TX2 GPU on average. With similar computation capability, the higher speed achieved by **OPU** comes from the higher PE utilization enabled by our domain micro-architecture.

2.5 Conclusions and Discussions

In this work, we propose **OPU**, a domain-specific FPGA-based overlay processor for general CNN acceleration. We generate an instruction set after analyzing the basic operations in general CNNs. A fully-pipelined micro-architecture is proposed to optimize computation efficiency and reduce communication latency. Comprehensive experiments are performed on KC705 evaluation board for **OPU** with 1024 MACs. It shows that **OPU** has a high RME of 91% for 9 commonly used CNNs, which indicates a high efficiency of the computation resources. For running VGG16, TinyYolo and YoloV2, **OPU** outperforms other automatically generated network-specific accelerators in terms of RME. Moreover, for cascaded CNN network to detect license plate, **OPU** is 9.6× faster than Nvidia Jetson TX2 GPU with similar amount of computation resources.

CHAPTER 3

LPFP: Low Precision Floating-point Arithmetic for High Performance FPGA-based CNN Acceleration

3.1 Introduction

Larger and deeper CNNs have been developed to improve performance for a broader range of scenarios. For example, the top-5 error for ImageNet [RDS15b] classification decreases from 17% to 2.9%. However, computation complexity and number of parameters increase dramatically as shown in Fig. 3.1, where the name and top-5 error of different CNNs are depicted in x-axis. To be specific, the computation complexity of a feed-forward process of a 224×224 RGB image increases from 2.27 GOP of AlexNet [KSH12b] in 2012 to 74 GOP of EfficientNet-B7 [TL19] in 2019. At the same time, the number of parameters stays large at 264 MB. Such great computation complexity makes it harder for general-purpose processors to meet the requirements of real-time applications. On the other hand, the great quantities of parameters lead to a big challenge for communication between off-chip and on-chip memories because of bandwidth constraints.

There are two types of research to reduce computation and parameter complexities for CNN inference. The first one is deep compression including weight pruning, weight quantization and compression storage [HPT15, HMD15a]. However, irregularity caused by deep compression degrades parallelism and hardware performance. Cambricon-S [ZDG18] alleviates irregularity in sparse neural networks through a software/hardware co-design approach to improve hardware performance. However, all the above accelerators need time-consuming

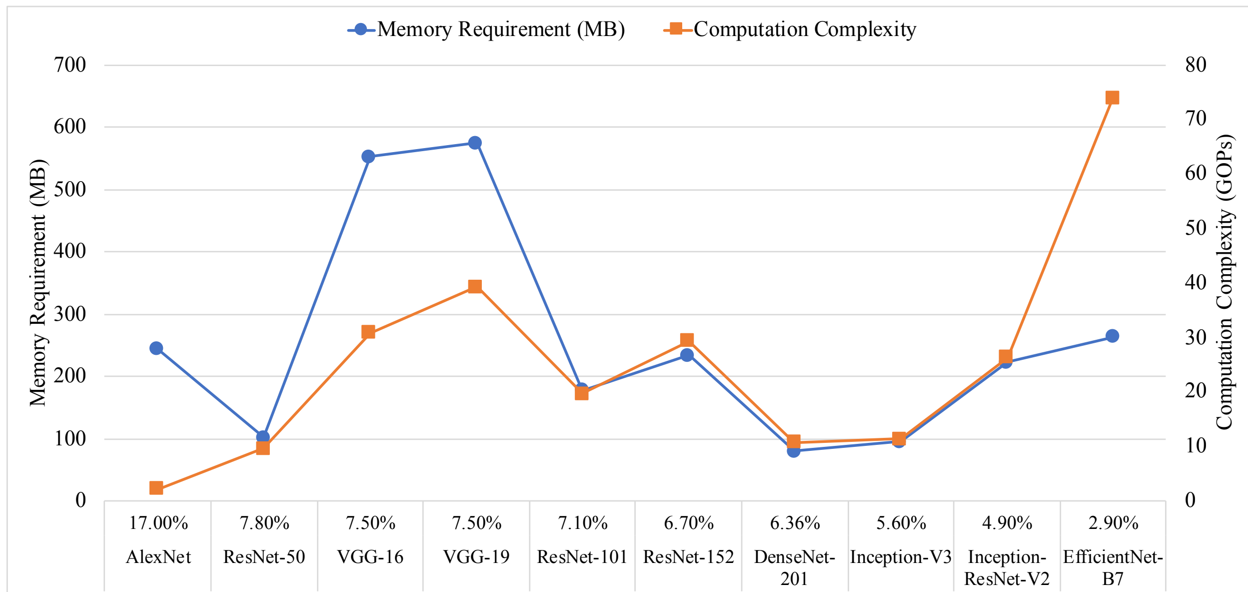


Figure 3.1: Computation complexity and memory requirement with respect to different CNNs.

retraining process to maintain accuracy.

The second type of research is more efficient data representation, also known as quantization for circuit implementation. The authors [MLN17] use 16-bit floating-point in contrast to 32-bit floating-point, which is commonly used for computing. However, one 16-bit floating-point multiplier on FPGA needs 1 DSP, 85 LUTs and 167 FFs when using Xilinx floating-point IP [Log12] as shown in Table 3.1, leading to a low hardware efficiency. Since one 16-bit or smaller fixed-point multiplier can be fit into one DSP, both 16-bit [XLL17b, MCV18] and 8-bit [ZLS15c, JYP17b, CDS14, YWZ19a] fixed-point are employed to gain more hardware efficiency than 16-bit floating-point does. Another 8-bit arithmetic, called block floating-point (BFP), is also applied [SLW18, LLS19], where a parameter has its own mantissa but shares a same exponent for one data block. ARM [LSC17] proposes a mixed data representation with floating-point for weights and fixed-point for activations (*e.g.*, outputs of a layer). Xilinx [SBD18] develops an 8-bit floating-point quantization scheme, which needs an extra training batch to compensate for the quantization error. However, [LSC17] and [SBD18]

Table 3.1: Resource utilization of multipliers on FPGA for different data representations. DSP: digital signal processing, LUT: look-up table, FF: flip-flop. $M4E3$: 1-bit sign, 4-bit mantissa and 3-bit exponent.

Data Representation	DSP	LUT	FF
one 16-bit floating multiplication	1	85	167
one 16-bit fixed multiplication	1	0	0
two 8-bit fixed multiplications	1	2	0
four 8-bit floating ($M4E3$) multiplications	1	20	27

do not present a circuit design for their approaches. While all aforementioned work has a good accuracy with retraining, more aggressive data representations such as binary [CBD15], ternary [LCM15], and mixed precision (2-bit activations and ternary weights) [CNN18] may suffer from great accuracy loss even with time-consuming retraining.

In this work, we first propose a low precision floating-point (LPFP) to quantize both weights and activations. During the quantization process, an optimal LPFP data format and the corresponding scale factor are decided for a workload of CNNs. Our proposed quantizer works for *deep* CNNs (more than 100 *convolutional/fully-connected* layers). On average, the top-1 accuracy loss is within 0.5%, while V-Quant [PYV18] that works for such *deep* CNNs has a top-1 accuracy loss about 1% with fine-tuning. Then, we design a LPFP based FPGA processor to further improve the performance for CNN inference. We are able to implement four 8-bit floating-point multiplications within one DSP (see Table 3.1). We experiment for inference of AlexNet, VGG16 [SZ14c], ResNet50/101/152 [HZR16b] and DenseNet201 [HLV17] via Xilinx KC705 and Xilinx ZCU106. We can achieve an average throughput of 1100.4 GOPS (Giga-Operations Per Second), and it is 1.43 GOPS per DSP on KC705. On ZCU106, the average throughput and per DSP throughput are 1650.6 GOPS and 2.15 GOPS per DSP, respectively. Moreover, the average throughput for these networks is 82.3% and 1.5× over Intel I7-8700T CPU and existing accelerators, respectively. Compared

with six existing accelerators for VGG16 and YOLO, on average, our processor improves throughput by 3.5× and 27.5×, while improving per DSP throughput by 4.1× and 5×, respectively. To the best of our knowledge, this is the first work that can fit four 8-bit multiplications for inference in one DSP while maintaining comparable accuracy without any retraining.

To summarize, the main contributions of this work are listed as follows:

- The **non-uniform quantization method with low-precision float-point data format** are used to quantize the input activations and weights for CNNs. The optimal data format, which achieves negligible accuracy loss, can be selected automatically without any fine-tuning, calibration or retraining.

- The previous work with 8-bit fixed-point quantization converts the original 32-bit floating point multiplication into 8-bit fixed-point multiplication, and one DSP slice can only be implemented to perform *two* 8-bit fixed-point multiplication. However, LPFP quantization converts the original 32-bit floating point multiplication into 8-bit floating-point multiplication, *four* of which can be implemented inside one DSP slice. Thus, 2× number of multipliers can be implemented under the same resource constraints for the same FPGA. Note that the computational throughout mainly comes from the number of multipliers, and our approach can achieve 2× computational throughput compared with previous work with 8-bit fixed-point quantization methods.

3.2 Background and Motivation

3.2.1 Background: Low Precision Floating-point

Similar to the definition of 32-bit floating-point from the IEEE-754 standard [ZCA08], the binary representation of LPFP number comprises *sign, mantissa* and *exponent* in order. The

decimal value of LPFP number is then calculated by:

$$V_{dec} = (-1)^S \times 1.M \times 2^{E-E_b}, \tag{3.1}$$

where V_{dec} is the value in decimal, S, M and E are all unsigned values and denote the *sign, mantissa* and *exponent*, respectively. For exponent bias E_b in Eq. (3.1), it is introduced to support both positive and negative exponents as

$$E_b = 2^{DW_E - 1} - 1, \tag{3.2}$$

where DW_E is the data width of E. Different from the IEEE Standard, data widths for M and E in this work are not fixed. In later sections, we use the term $MaEb$ to indicate different combinations, where a and b indicate the bit width of M and E, respectively. For example, $M3E4$ means the mantissa is 3 bits while the exponent is 4 bits.

There are three special definitions in IEEE-754 standard. The first is subnormal numbers when $E = 0$, then Eq. (3.1) is modified to:

$$V_{dec} = (-1)^S \times 0.M \times 2^{1-E_b}. \tag{3.3}$$

Note that Infinity (Inf) and Not a Number (NaN) are the other two special cases, but are not used in our work. This is because our saturation scheme saturates large numbers to the maximal number, as illustrated in detail in Subsection 3.3.1.

3.2.2 Motivation

CNN accelerators with lower data width have significant improvements in terms of memory size, memory bandwidth and power efficiency. Due to the lack of floating-point arithmetic units in FPGA, researchers have used low precision fixed-point instead of floating-point. A 16-bit fixed-point quantization to find the best scale factor for each layer is proposed in [XLL17b]. However, this requires time-consuming retraining to amend the weights to maintain accuracy. Furthermore, a model is developed to quantitatively analyze the convolution loops and optimize design objectives such as memory access and latency [MCV18].

However, it has an accuracy loss as large as 2%. A shared drawback for the above two approaches is the low per DSP throughput (0.279 GOPS/DSP for [XLL17b] and 0.472 GOPS/DSP for [MCV18]) because of using 16-bit multiplication.

An 8-bit fixed-point accelerator is designed in [GSQ17] for embedded FPGAs, with a low per DSP throughput of 0.444 GOPS/DSP. DNNBuilder [ZWZ18b] aims to automatically build high-performance DNN hardware accelerators for both cloud- and edge-FPGAs with 8-bit fixed-point quantization. It increases the per DSP throughput to 0.771 GOPS by better architecture exploration; however, its quantization method incurs 4.6% top-1 accuracy degradation without fine-tuning. FPGA accelerator with the aforementioned BFP arithmetic [LLS19] has a per DSP throughput of 0.741 GOPS. However, only *slim* and *medium* CNNs are validated in their approach. Approaches in the industry focus on improving the accuracy with 8-bit fixed-point data representation [JGW19, Inc21a, Mig17]. However, even if we use the quantization policies from these work, we will still suffer from the low per DSP throughput as one DSP can only be decomposed to 2 8-bit fixed-point multiplications. In short, existing approaches cannot improve the per DSP throughput while maintaining comparable accuracy for all *slim, medium* and *deep* CNNs.

3.3 Low Precision Floating-point Quantization

In this section, we present the details of our proposed low precision floating-point (LPFP) quantization method, including the quantization process, data flow in processor and quantization results.

3.3.1 Quantization Process

The quantization process is divided into two steps: 1) fusing operations; 2) finding scaling factor and quantizing data. We will discuss in detail in this section.

Fusing layers. In each CNN, we define the *convolutional/fully-connected* layers as key

layers while others as subordinate layers. During the quantization process, we try to merge as many subordinate layers into the previous key layers as possible to simplify the design. This is because most of the subordinate layers will not change the data precision during calculation. For example, batch normalization layer is linear and can be merged in advance; max pooling layer only does comparison and will not influence the precision. Moreover, for the subordinate layers which cannot be merged into the key layers, (*i.e.,* average pooling), we treat it as a separate layer and will do quantization in the next step.

Finding scaling factor and quantizing data. The proposed LPFP quantization method is applied to the output activations and weights of each fused layers. The quantization function is defined as follows:

$$V_{lfp} = quan(V_{fp32} \times 2^{sf}, MIN_{lfp}, MAX_{lfp}), \tag{3.4}$$

where V_{lfp} and V_{fp32} denote the decimal values represented by LPFP and traditional single floating-point format, respectively; MIN_{lfp} and MAX_{lfp} indicate the minimal and maximal numbers represented by LPFP, and sf is the scaling factor which is used to better fit the data into the dynamic range of LPFP. After finding the optimal scaling factor and quantizing the activations and weights of each layer, the scaling factor needs to be re-normalized for accuracy. To simplify the calculation in the processor, we do the re-normalization of the scaling factor when we re-quantize the output activations to LPFP format in the data conversion step (see details in the Section 3.3.2). The *quan* function in Eq. (3.4) rounds the data to the nearest value with saturation considered, formulated as

$$quan(x, MIN, MAX) = \begin{cases} MIN & x <= MIN \\ MAX & x >= MAX \\ round(x) & \text{otherwise} \end{cases}, \tag{3.5}$$

where MIN and MAX are the minimal and maximal values, respectively.

The mean square error (MSE) of the values before and after quantization is used as the

metric to evaluate the quantization error, illustrated as:

$$MSE = \frac{1}{N}\sum_{i=0}^{N}(V_{lfp}/2^{sf} - V_{fp32})^2, \tag{3.6}$$

where N denotes the amount of data.

As illustrated from Eq. (3.4) to (3.6), MSE is influenced by the data format of LPFP and the scaling factor (sf). Since the quantization process is performed offline only once for each CNN, we use exhaustive search to find the optimal combination of LPFP data format and scaling factors for both weights and activations. During the inference process for each CNN, the quantized weights and scaling factors of activations are fixed for different test images. Therefore, no extra computation is need during the inference process. In this work, we assume **the same data format for a CNN and a same scaling factor for each layer**. This assumption can be removed as needed. Furthermore, we choose to use a same optimized data format for all test cases in our experiments, while the problem formulation is to decide a data format for each CNN.

3.3.2 Data Flow in Processor

The data flow of our proposed approach is shown in Fig. 3.2. In order to explicitly illustrate the data flow, we list the bit width in each step with $M4E3$ data format as an example. The weights and biases of the pre-trained model are represented by 32-bit floating-point. The weights are first quantized into $M4E3$ while the biases are quantized into 16-bit fixed-point to reduce quantization error. All the quantized weights and biases are then stored into the external memory of the FPGA board. The quantization of weights and biases are performed on a server only once for each CNN. In our processor, the raw input image which indicates the input of the first layer is also quantized from 32-bit floating point into $M4E3$ and stored into the external memory. During the inference on our FPGA processor, the quantized image, weights and biases are fetched from the external memory, and the multiplications of image and weights are performed with $M4E3$ data format. The $M4E3$ multiplication is

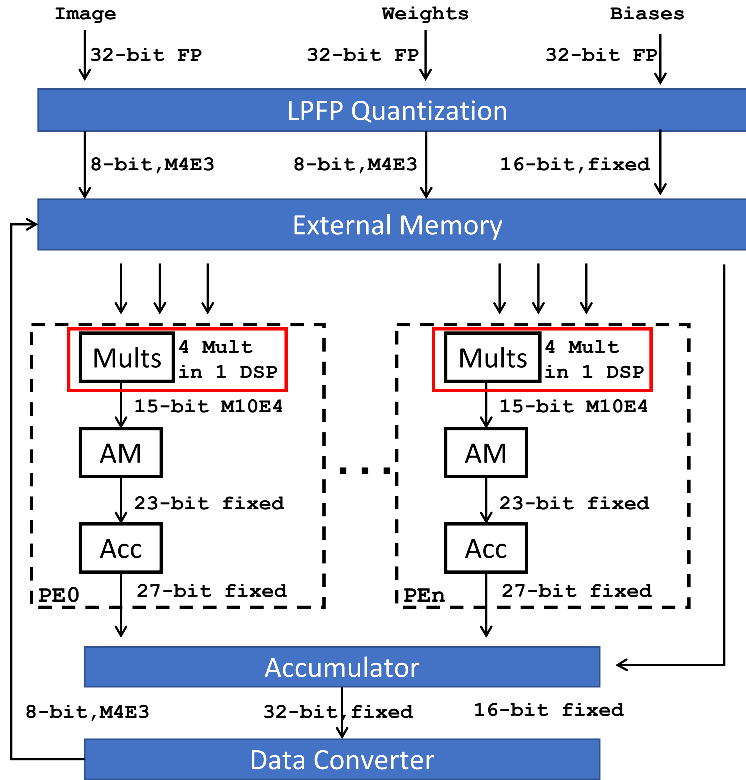


Figure 3.2: The data flow in our processor with $M4E3$ data format as an example (FP: floating-point, Mult: LPFP multiplier, AM: alignment module, Acc: accumulator, DC: data converter).

decomposed into three parts: 1) xor of the sign bit; 2) multiplication of the mantissa; and 3) addition of the exponent. To maintain full precision during computation, the results of the LPFP multiplier are kept 15-bit, with 1 sign bit, 10 mantissa bits and 4 exponent bits. The followed align module (AM) converts the 15-bit products to 23-bit fixed-point without any precision loss. Considering the exponent of the product can be varied from -4 to 8, we can use 12 more bits to cover all the 13 cases (12 bits can have 13 dot positions). Moreover, we set the 23-bit fixed-point to have 12 decimal places by considering the worst case (4 bits from the worst case of exponent to be -4 and 8 bits from the 10-bit mantissa). In this way, all the accumulation can be done in 32-bit fixed-point accumulators with saturation, which consumes fewer resources in FPGA than floating-point accumulators. Since the fixed-point

accumulator does not change the dot position, the final outputs of the accumulators still have 12 decimal places. Finally, we quantize the 32-bit fixed-point data back to $M4E3$ floating-point and store them in the external memory before being used by another CNN layer. As the scaling factors of the input activations and kernels are propagated to the output activations during convolution, we need to re-normalize the output activations for accuracy by multiplying $2^{sf_{oa}-sf_{ia}-sf_k}$, where sf_{oa}, sf_{ia} and sf_k indicate the scaling factors of output activations, input activations and weights, respectively. This is done by shifting the 32-bit fixed-point data in the data conversion step. Moreover, to simplify the data conversion, we will first quantize the shifted data to 16-bit fixed-point with 1 sign bit, 7 integer bits and 8 decimal bits. After that, the 16-bit fixed-point data is quantized to 8-bit LPFP data format. In the whole data flow, only the final data conversion step introduces bit truncation and precision loss. However, the precision loss introduced by the final step has little impact on the final accuracy and is validated in Section 3.5.1.2 with comprehensive experimental results.

3.4 Processor Architecture

In this section, we discuss in detail the architecture of the processor, which efficiently supports the inference process of quantized networks for various CNNs.

3.4.1 Overview

The overall architecture of the proposed processor is depicted in Fig. 3.3. A floating-point function unit (FPFU), which is composed of multiple processing elements (PEs), is developed to compute the outputs of a layer in parallel. The PE, which is the key component of FPFU, is designed to efficiently perform dot product with LPFP data format. The on-chip memory system (MS) consists of three buffers, *e.g.,* input feature map buffer (IFMB), weight buffer (WB) and output feature map buffer (OFMB). All these three buffers are ping-pong

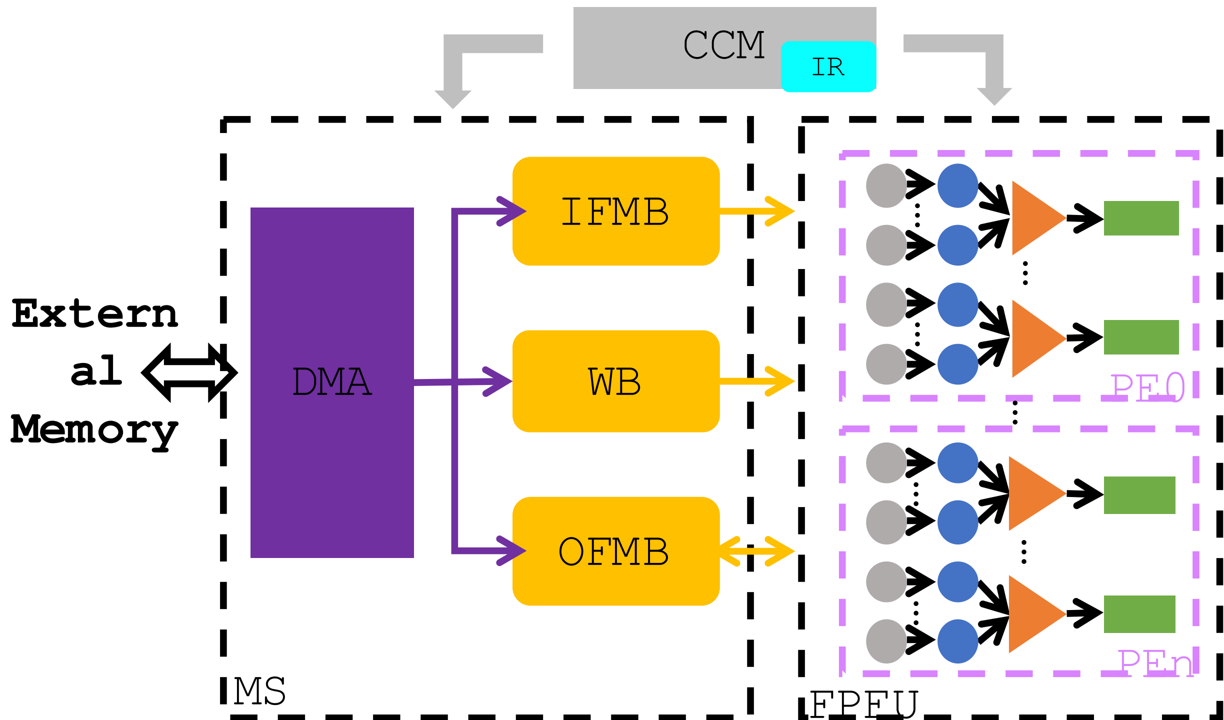


Figure 3.3: The overall architecture of proposed processor.

architecture to hide the communication time between on-chip and off-chip memories through direct memory access (DMA) module. The central control module (CCM) is designed to arbitrate between different modules. Moreover, the CCM decodes various instructions stored in the instruction RAM (IR) into detailed signals for other modules.

3.4.2 Floating-point Function Unit

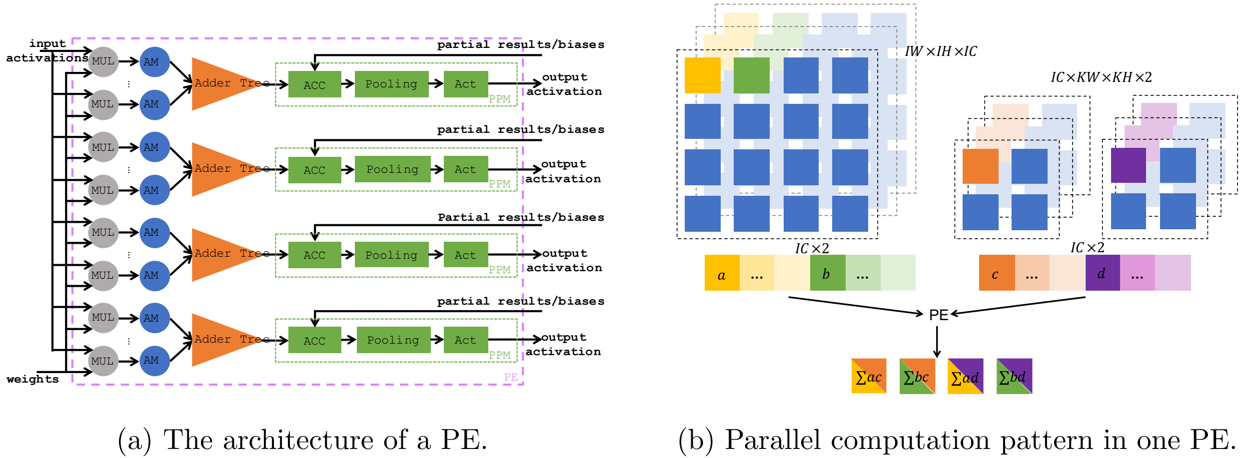
FPFU, which is constructed by multiple PEs, is designed to perform convolution in LPFP data format efficiently for performance gain and power reduction. Different parallel computation patterns, including parallel in input feature maps, parallel in output feature maps and parallel in both input and output feature maps, are developed in FPFU and are discussed in the following paragraphs. FPFU receives activations and weights from IFMB and WB, respectively, and distributes the activations and weights to different PEs to perform

convolution according to the control signals decoded by CCM.

3.4.2.1 Architecture of PE

The PE is designed as a fully pipelined data-flow-based architecture, as shown in Fig. 3.4a. Once a PE receives the activation and weight vector, which are represented with $M4E3$ data format, it distributes the data to N_m multipliers inside the PE. The products of the multipliers keep full precision and are transferred into the alignment module (AM). The full precision products are aligned and converted to fixed-point numbers without any bit truncation. The aligned products are then fed into four fixed-point adder trees to finalize four dot product processes in parallel, which indicates the feed-forward process of four pixels in two output channels (see details in Section 3.4.2.2). The accumulation of partial results (including bias), pooling and activation processes are performed in series inside the post process module (PPM).

The multipliers in each PE are developed for LPFP, which are represented with scientific notation in the sign-and-magnitude format, as illustrated in Eqs. (3.1) and (3.3). The multiplication of two LPFP numbers is then divided into three fixed-point components: (1) XOR of the signs; (2) multiplication of mantissas; (3) addition of exponents. Take the $MaEb$ format as an example. An a-bit unsigned MAC and a b-bit unsigned adder are needed. Considering the first hidden bit of mantissas – "1" for normal numbers and "0" for subnormal numbers – we design an a-bit multiplier and an $a+2$-bit adder to perform the $a+1$-bit multiplication. The a-bit multipliers and $a+2$-bit adders are implemented within one DSP slice to improve per DSP throughput (see details in Section 3.5.2.2). Meanwhile, the exponent bias E_b is not included during addition, because the E_b is the same for all the numbers in one CNN as we assume, and we can address this at the last step to simplify the adders.



(a) The architecture of a PE.   (b) Parallel computation pattern in one PE.

Figure 3.4: The architecture of a PE and the parallel computation pattern in a PE. MUL: LPFP multiplier, AM: alignment module, ACC: accumulator, Act: activation.

3.4.2.2 Parallel Computation Pattern

The *convolutional/fully-connected* layers are calculated in the PE. For the fully-connected layers, we treat the input feature maps as "weights" and weights as "input feature maps" to increase the data reuse and make the calculation pattern the same as convolutional layers. Therefore, each pixel in one output channel for *convolutional/fully-connected* layers is calculated as

$$y_i = \sum_{k=0}^{KW \times KH} \sum_{ic=0}^{IC} x_{k,ic} w_{k,ic} + b_i, \tag{3.7}$$

where IC indicates the number of input channel, KW and KH denote the width and height of the kernel, and x, y, w and b are input activation, output activation, weight and bias, respectively. In our implementation on FPGA, we implement 4 LPFP multipliers with one DSP slice, which follows the pattern: $(a + b) \times (c + d) = ac + bc + ad + bd$ (see details in Section 3.5.2.2). Therefore, each PE is designed to process convolution in two output channels in parallel, and in each output channel, it will calculate the convolutional results of two pixels at the same time, as shown in Fig. 3.4b. To be specific, in the first cycle, the

first pixel in IC input channels and the first value in the corresponding kernels are fed into the PE, marked with a and c in Fig. 3.4b, respectively. To follow the computation pattern in these four multipliers, the second pixel in IC input channels (marked with b), and the corresponding kernels to calculate the pixel in another output channel (marked with d) are also fed into the PE. In this way, a and b are reused to produce the pixels in different output channels, while c and d are reused to produce the pixels in different positions of the same output channel. After $KW \times KH$ cycles, four convolution results are produced by one PE.

As illustrated in Section 3.4.2.1, N_m multipliers are used in each PE, and IC is designed to be $N_m/4$. In this way, $N_m/4$ input channels are calculated in parallel in each PE. With the corresponding weights and biases, 2 pixels in 2 output channels are calculated in parallel. When the number of input channels is larger than $N_m/4$ and/or when the number of pixels in each output channel is larger than 2 and/or when the number of output channels is larger than 2, multiple rounds of computation are needed in series to finalize the convolution. In order to further increase the parallelism, we use N_p PEs in the FPFU. In different PEs, we can feed in different pixels in input feature maps and weights to perform different parallel computation pattern. For example, the N_p PEs can share the same input feature map and use different weights to parallelize the computation in output channels, or the N_p PEs can share the same weights and use different input feature maps to parallelize the computation in input channels. The N_m, N_p and the parallel computation pattern are decided by considering the CNNs, the throughput and the bandwidth requirement. This will be explained with experiments in Section 3.5.2.2.

3.4.3 Memory System

In order to keep the PE working without waiting for the data to be ready, the bandwidth of IFMB and WB for each PE are designed to be $N_m/2$ LPFP input activations and weights per cycle, respectively, while the OFMB is 4 output activations per cycle. Although each pixel in the output feature map is represented with LPFP data format, we keep the intermediate

results with 16-bit precision to reduce accuracy loss. In this way, the bandwidth of OFMB for each PE is set to 64 bits per cycle. As the input activations and/or weights can be shared by different PEs according to different computation patterns, we define P_{ifm} and P_{ofm} ($P_{ifm} \times P_{ofm} = N_p$) to indicate the parallelisms in input feature map and output feature map, respectively. In this definition, P_{ifm} indicates that we have P_{ifm} PE groups where the same weights are shared during calculation, while in each PE group, P_{ofm} PEs share the same input activations. In conclusion, the bandwidth for IFMB, WB and OFMB are $N_m/2 \times P_{ifm} \times BW$, $N_m/2 \times P_{ofm} \times BW$ and $64N_p$ per cycle, respectively, where BW denotes the bit width of LPFP data format.

As the amount of on-chip buffers are limited, we stored all the feature maps, weights, and biases in the off-chip memory and preload the feature maps, weights and biases needed by a computation block in the on-chip memories before computation. In this way, the parameters N_m, P_{ifm} and P_{ofm} are decided to trade off between the throughput, bandwidth requirement and resource utilization. Previous proposed work applied large enough buffers to store all the activations or weights for one layer [HLM16a] to avoid costly off-chip memory access. However, such designs incurred large area and unscalability for larger and deeper CNNs. In our processor, we trade off among the throughput, bandwidth requirement, resource utilization and scalability, and employ the smallest size which can hide the DMA communication time. In our implementation on FPGA, we use block RAM to deploy IFMB and OFMB, while we use distributed RAM to deploy WB, as distributed RAM can provide higher bandwidth than block RAM. During inference on our processor, only when all the input feature maps have been processed and reused, or all the weights have been processed and reused, or OFMB is full, will the off-chip memory be accessed for loading new input feature maps, loading new weights or storing output feature maps, respectively.

3.4.4 Central Control

The CCM is designed to arbitrate among different modules and control the whole execution process. First, CCM decodes the instructions from IR efficiently and sets the corresponding control registers. Second, different modules are activated according to the control registers and the status of each module is monitored by the control registers as well. Finally, the CCM decides when to fetch the next instruction from the feedback of the control registers. We also design a compiler to generate the block-level instructions.

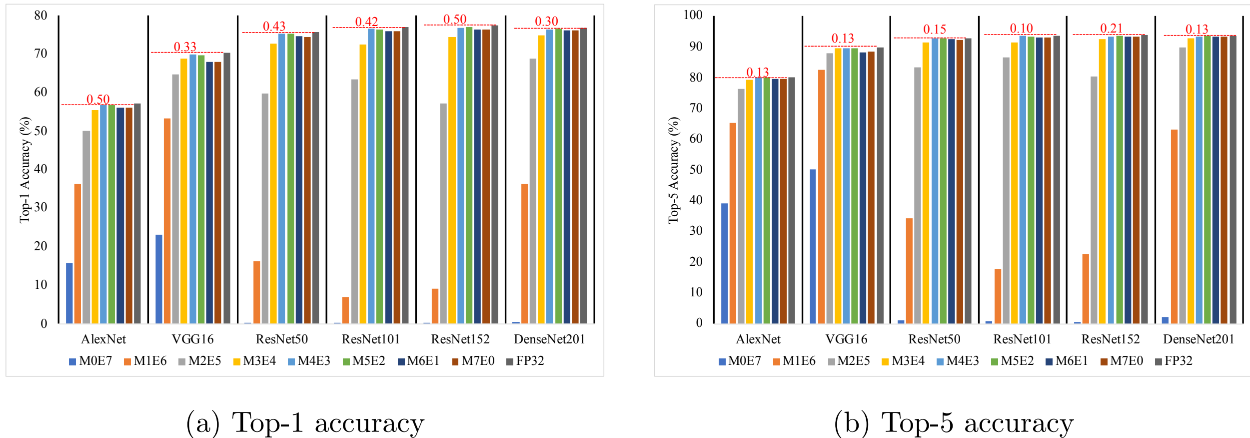
3.5 Evaluation

In this section, the evaluation of the proposed quantization method is first provided, then the implementation details and comprehensive experimental results are provided.

3.5.1 Evaluation of Quantization Method

3.5.1.1 Experiment Setup

We implement our LPFP quantization method with C language based on the Darknet framework [Red16], and the inference process of the quantized network follows the same data flow as that in our processor illustrated in Fig. 3.2. The validation accuracy with single center-crop is then evaluated via the ImageNet validation set (50,000 labelled images) [RDS15b]. Our quantization process is run on an Intel (R) Core (TM) I7-8700T CPU working under 2.40GHz, while the evaluation process is run on a Nvidia TITAN Xp GPU. During the evaluation process on GPU, all the quantized data are converted to 32-bit floating without any precision loss. In addition, the computation on GPU are based on 32-bit floating point. Six representative CNNs (AlexNet, VGG16, ResNet50/101/152 and DenseNet201) including the *slim, medium* and *deep* CNNs are evaluated.



(a) Top-1 accuracy

(b) Top-5 accuracy

Figure 3.5: Top-1/Top-5 accuracy for different (mantissa, exponent) combinations with respect to different CNNs.

3.5.1.2 8-bit Quantization

The detailed validation accuracies on the quantized networks with 8-bit floating-point data format are shown in Fig. 3.5a and 3.5b. We emulate all 8 different (mantissa, exponent) combinations to validate the top-1 and top-5 accuracy of the quantized CNNs, and the 32-bit floating-point results are included as the baseline.

In Fig. 3.5a and 3.5b, the dashed lines illustrate the 32-bit floating-point baseline, while the values above the dashed lines are the accuracy loss compared with the baseline. We can see that our LPFP quantization approach can maintain comparable top-1 and top-5 accuracy to the baseline. On average, the top-1 and top-5 accuracy loss is within 0.5% and 0.3% compared with the full precision results, respectively. Particularly, $M5E2$ always achieves the highest accuracy compared with the other cases. Data formats with more than or equal to 3-bit mantissa all have a low accuracy loss for all the six CNNs, while those with less than 3-bit mantissa can hardly find accurate results. We also compare our proposed approach with the fixed-point situation, marked as $M7E0$ in the figures ($M7E0$ means 1-bit sign, 7-bit mantissa and no exponent, exactly fixed-point). As shown in Fig. 3.5a and 3.5b, $M4E3$ and $M5E2$ outperform the fixed-point for all six benchmarks.

Table 3.2: Accuracy comparison between $M4E3$, $M5E2$, references and FP32. "-" means no reported results. "R" means the method with retraining.

	Top-1 Accuracy (%) Top-5 Accuracy (%) for each network											
	AlexNet		VGG16		ResNet50		ResNet101		ResNet152		DenseNet201	
Nvidia [Mig17]	0.03	-0.01	0.03	-	0.13	0.12	-0.01	0.06	0.08	0.05	-	-
BFP [LLS19]	-	-	0.03	0.02	0.11	0.12	-	-	-	-	-	-
Ours ($M4E3$)	**0.58**	**0.19**	**0.33**	**0.13**	**0.55**	**0.15**	**0.42**	**0.10**	**0.81**	**0.39**	**0.45**	**0.19**
Ours ($M5E2$)	**0.48**	**0.13**	**0.64**	**0.32**	**0.43**	**0.19**	**0.67**	**0.37**	**0.50**	**0.21**	**0.30**	**0.13**
Ours ($M4E3$) **R**	**0.03**	**0.01**	**0.03**	**0.00**	**0.08**	**0.06**	**0.05**	**0.02**	**0.04**	**0.01**	**0.07**	**0.03**
Ours ($M5E2$) **R**	**0.04**	**0.00**	**0.02**	**0.00**	**0.05**	**0.03**	**0.06**	**0.04**	**0.03**	**0.02**	**0.03**	**0.01**

3.5.1.3 Comparison with the Prior Quantization Strategies

$M4E3$ and $M5E2$, which achieve the two best accuracies among all the test cases, are also compared with five typical approaches. We report both the top-1 and top-5 accuracy losses for all six benchmarks in Table 3.2, where "-" indicates no reported results in the literatures. We also retrain our quantized network using $M4E3$ and $M5E2$ for 10 epoch with the original training data, and the Top-1 and Top-5 accuracy loss are also included in Table 3.2. Although the top-1 and top-5 accuracy losses achieved by our LPFP quantization method without retraining are not the best among all the literatures, our method without retraining is suitable for fast deployment and does not need any training data in the real-world. Moreover, after retraining, our LPFP quantization method can regain the validation accuracy compared with the 32-bit floating point networks. Moreover, our method can reach *deep* networks.

3.5.1.4 Lower Bit Width Quantization

We further reduce the bit width from 8-bit to 4-bit and also evaluate the top-1 and top-5 accuracy of the quantized networks. We pick the best (mantissa, exponent) combination for

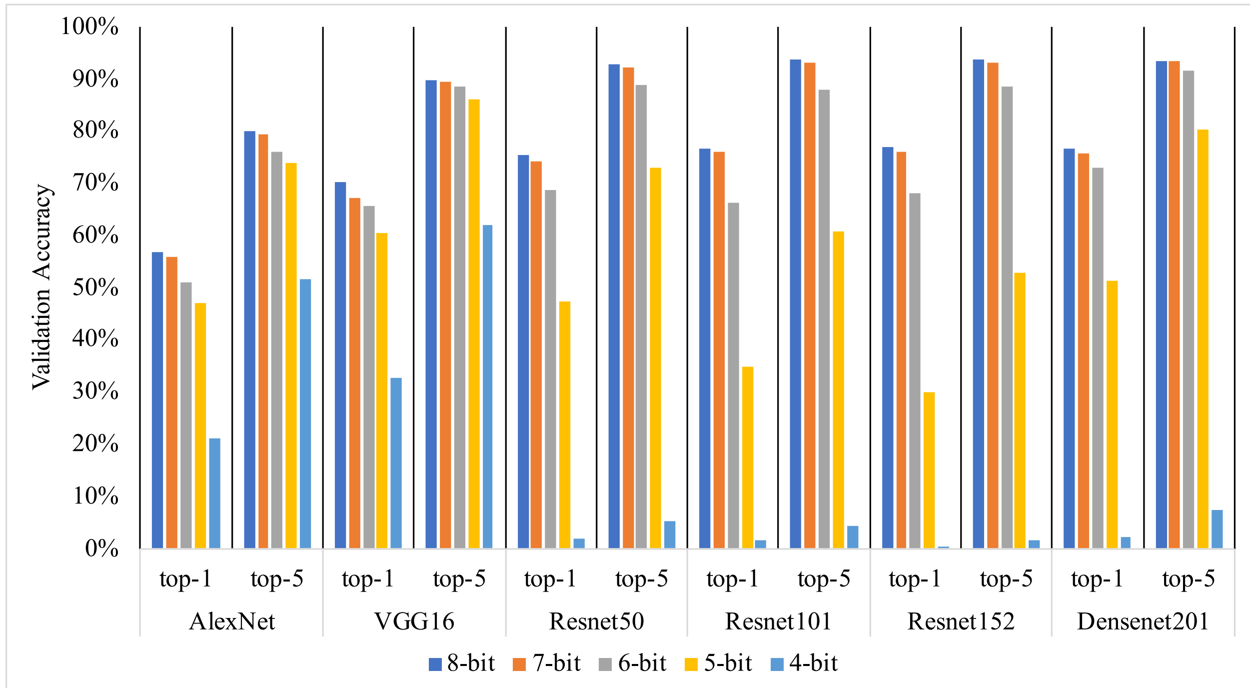


Figure 3.6: Top-1 and top-5 accuracies for different bit width with respect to different CNNs.

each data format and the results are shown in Fig. 3.6. We can see that both the top-1 and top-5 accuracy decrease when lower bit length is utilized to represent the weights and activations of CNNs. Particularly, the average top-5 accuracy degradations for 7-bit and 6-bit are 0.8% and 4.2%, respectively. However, the accuracy drops dramatically when the bit width decreases to less than 6 bits, which means our LPFP quantization approach can hardly find accurate results without any retraining process.

3.5.2 Evaluation of Hardware Implementation

3.5.2.1 Environment Setup

Our processor is implemented on the KC705 evaluation board with a Kintex-7 XC7K325T FPGA and ZCU106 evaluation board with a Zynq UltraScale+ XCZU7EV FPGA. First, we explore the parallel computation patterns to find the optimal parameters to best fit the

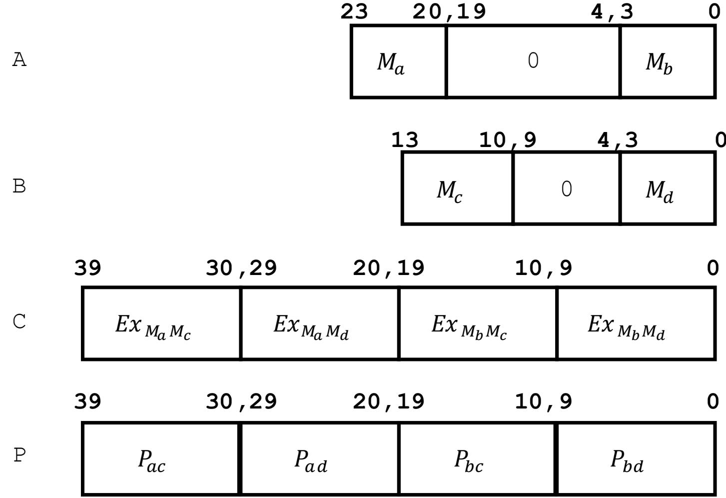


Figure 3.7: Data format of the DSP to implement four 4-bit MACs. M_a, M_b, M_c and M_d: the mantissas of LPFP data a, b, c and d, respectively; $Ex_{M_aM_c}, Ex_{M_aM_d}, Ex_{M_bM_c}$ and $Ex_{M_bM_d}$: the extra term expressed as $Ex_{M_aM_c} = 1.M_a + 0.M_c$; P_{ac}, P_{ad}, P_{bc} and P_{bd}: the mantissas of the product of two LPFP data expressed as $P_{ac} = 1.M_a \times 1.M_c$.

two FPGAs. Second, with these parameters, the processor is described in Verilog-HDL, and synthesized and implemented with the Xilinx Vivado 2018.2 Design Suite. Finally, we evaluate the throughput, inference latency and per DSP throughput of running different networks on our processor, and the results are compared with two prior accelerators [MCV18, LWC18]. The Intel (R) Core (TM) I7-8700T CPU under 2.40GHz working frequency and the Nvidia Xavier NX GPU with a 8GB LPDDR4 are also used for comparison. More comprehensive experimental results on VGG16 and YOLO are compared with latest FPGA accelerators [MLN17, XLL17b, MCV18, GSQ17, LWC18, LLS19, WMS18].

3.5.2.2 Implementation Details

We use the $M4E3$ data format for FPGA implementation in this work for two reasons. First, $M4E3$ achieves the top two best validation accuracies among all the LPFP (mantissa, exponent) combinations we tested (see Section 3.5.1). Particularly, the average top-1 and

top-5 accuracy loss of $M4E3$ compared with 32-bit floating-point are 0.53% and 0.19%, respectively. Second, $M4E3$ only needs a 4-bit fixed-point MAC and a 3-bit fixed-point adder, resulting in fewer resources on FPGA than $M5E2$. To be specific, four 4-bit fixed-point MACs can be implemented inside one DSP48E1 slice in XC7K325T FPGA.

In order to clearly explain the way to implement four MACs with one DSP48E1 slice, we take the multiplication of two normal numbers (X and Y) as an example. The mantissa of the product can be explained as:

$$\begin{aligned} Prod &= 1.M_x \times 2^{E_x - E_b} \times 1.M_y \times 2^{E_y - E_b} \\ &= (0.M_x \times 0.M_y + (1.M_x + 0.M_y)) \times 2^{E_x + E_y - 2E_b}, \end{aligned} \tag{3.8}$$

where M_x, M_y, E_x and E_y are the mantissas and exponents of X and Y, respectively. In Eq. (3.8), the term $0.M_x \times 0.M_y + (1.M_x + 0.M_y)$ is performed with a 4-bit unsigned fixed-point MAC and the term $E_x + E_y$ is performed with an extra 3-bit unsigned fixed-point adder. As the DSP48E1 slice can be implemented as a MAC followed by $P = A \times B + C$ (where the maximal bit width of A, B and C are 25, 18 and 48, respectively), we add blank bits to the three inputs to fully utilize the functionality of DSP48E1, as shown in Fig. 3.7. This proposed method also works for the next generation of Xilinx DSP slice (DSP48E2), which can also be configured as a MAC, with the maximal bit width of A, B and C to be 27, 18 and 48. During the calculation process, the dot position is kept at the right most position. That is, the terms $0.M_x$ and $0.M_y$ are converted to 4-bit integers, while the extra term $1.M_x+0.M_y$ is converted to 10-bit integers to make sure that no overlap occurs. In this way, with a few LUTs and FFs to perform additions of the exponents and the extra term $1.M_x + 0.M_y$, four multiplications with $M4E3$ data format can be carried out in on DSP slice (see Table 3.1), thus dramatically increasing the per DSP throughput.

Parallel Exploration. Since one DSP slice is divided into four 4-bit LPFP MACs in our implementation, the parameters should meet the requirement that $N_m \times N_p = 4 \times \#ofDSP$. Considering the resources of XC7K325T FPGA, we set the targeted number

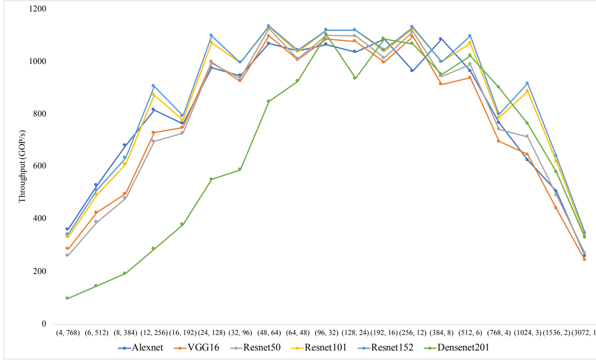
Table 3.3: Resource Utilization on XC7K325T.

Resource	LUT	LUTRAM	FF	BRAM	DSP
Used	154625	7860	180561	234.5	768
Available	203800	64000	407600	445	840
Utilization	75.87%	12.28%	44.30%	52.70%	91.43%

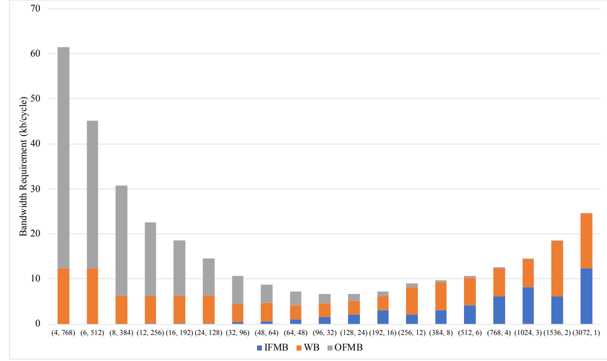
of DSP as 768, which accounts for 91.43% of the available DSPs. We then evaluate the throughput for different CNNs and the bandwidth requirement with respect to different N_m and N_p combinations as shown in Fig. 3.8a and 3.8b, respectively. We also explore different combinations of the parameters P_{ifm} and P_{ofm}, and only depict the P_{ifm} and P_{ofm} for achieving the optimal throughput and minimal bandwidth requirement in Fig. 3.8a and 3.8b.

In general, when N_m keeps increasing, the throughput first increases and then decreases when it reaches the peak. The small N_m and large N_p indicate that more output channels are calculated in parallel while large N_m and small N_p mean more input channels are calculated in parallel. When N_m is larger than the total number of input channel (denoted as IC), only IC multipliers are used while the rest are wasted, resulting in a low throughput. This is the same for large N_p, and the peak throughput comes from balanced N_m and N_p. For different CNNs, the peak throughput comes from different N_m and N_p combinations due to different network configurations. For example, DenseNet201 has lots of inception layers, which concatenate layers with small output channels (*e.g.,* 32) to form layers with large input channels (*e.g.,* 1568). In this case, larger N_m and smaller N_p incur fewer wasted computations and lead to higher throughput. From Fig. 3.8a, we can see that the combination of $N_m = 96$ and $N_p = 32$ results in an optimal throughput for all cases on average.

The bandwidth requirement is extremely high when N_p is large. This is because larger N_p indicates more parallel computations in output channels. Moreover, OFMB is designed to store 16-bit intermediate results, which also lead to higher bandwidth requirement with



(a) Throughput for different CNNs with respect to different N_m and N_p combinations.



(b) Bandwidth requirement with respect to different N_m and N_p combinations.

Figure 3.8: Parallel exploration with respect to throughput and bandwidth requirement.

Table 3.4: Resource Utilization on Ultrascale+ 7EV.

Resource	LUT	LUTRAM	FF	BRAM	URAM	DSP
Used	133517	15760	201465	84.5	48	768
Available	230400	101760	460800	312	96	1728
Utilization	57.95%	15.48%	43.72%	27.08%	50%	44.5%

larger N_p. The total bandwidth requirement decreases when N_p decreases, and then increases again since larger N_m needs more bandwidth to load input activations and weights. The smallest bandwidth requirement comes when we have a balanced combination of N_m and N_p. As concluded from Fig. 3.8b, the optimal combinations are $N_m = 96, N_p = 32$ and $N_m = 128, N_p = 24$. Take the case for optimal throughput, we set $N_m = 96$ and $N_p = 32$ in this implementation.

3.5.2.3 Experimental Results

Resource Utilization. Given the parameters that $N_m = 96$ and $N_p = 32$, the detailed post-implementation resource utilization under 200MHz working frequency on Kintex 325T is

Table 3.5: Comparison between Intel I7-8700T CPU, Nvidia Xavier NX GPU, existing accelerators and our processor with respect to different CNNs. "-" means no reported results.

	Throughput (GOPS)	Inference Latency (ms)	per DSP throughput (GOPS/DSP)						
	AlexNet			VGG16			ResNet50		
Intel i7-8700T [Inc21b]	-	-	-	-	-	-	755.6	10.19	-
Nvidia Xavier NX	1317.6	1.72	-	4253.4	7.19	-	2402.4	3.21	-
Ma, et. al. [MCV18]	-	-	-	715.9	42.74	0.47	611.4	12.59	0.40
RNA [LWC18]	687.8	3.3	-	878.1	34.85	-	804.3	9.57	-
ours on 325T	**1066.4**	**2.13**	**1.39**	**1086.8**	**28.16**	**1.42**	**1101.9**	**6.99**	**1.43**
ours on 7EV	**1599.6**	**1.42**	**2.08**	**1630.2**	**18.77**	**2.12**	**1652.9**	**4.66**	**2.15**
	ResNet101			ResNet152			DenseNet201		
Intel i7-8700T [Inc21b]	-	-	-	-	-	-	-	-	-
Nvidia Xavier NX	2584	5.88	-	2734.6	8.26	-	972	11.1	-
Ma, et. al. [MCV18]	-	-	-	707.2	31.96	0.47	-	-	-
RNA [LWC18]	-	-	-	-	-	-	-	-	-
ours on 325T	**1121.4**	**13.6**	**1.46**	**1121.3**	**20.2**	**1.46**	**1104.7**	**9.78**	**1.44**
ours on 7EV	**1682.1**	**9.04**	**2.19**	**1682.0**	**13.4**	**2.19**	**1657.1**	**6.52**	**2.16**

listed in Table 3.3 and resource utilization under 300MHz working frequency on Ultrascale+ 7EV is listed in Table 3.4.

Throughput and per DSP throughput for different CNNs. Six representative CNNs, including *slim, medium* and *deep* networks, are mapped on our processor. When calculating the CNN size, one MAC is counted as two operations. The throughput is measured in GOPS (Giga Operations Per Second), and is reported for different networks on our processor, Intel I7-8700T and Nvidia Xavier NX GPU in Table 3.5. We map our processor on two typical FPGAs (Xilinx Kintex-7 325T and Xilinx Ultrascale+ MPSoC 7EV) which target on edge applications. For fair comparison, we use the reported int8 results with OpenVINO on Intel I7-8700T [Inc21b], and run all the 8-bit fixed-point networks on Nvidia Xavier NX with TensorRT from PyTorch models [Inc21c]. As the existing studies [MCV18, LWC18] support multiple networks, we also include their results in Table 3.5.

Table 3.6: Comparison with prior accelerators on VGG16. ”-” means no reported results.

	Mei, et.al. [MLN17]	Xiao, et. al. [XLL17b]	Ma, et. al. [MCV18]	Angel-Eye [GSQ17]	RNA [LWC18]	BFP [LLS19]	OPU [YWZ19a]	**ours**
Year	2017	2017	2018	2018	2018	2019	2019	**2019**
Platform	XC7VX690T	XC7Z045	Arria 10 GX1150	XC7Z020	XC7Z045	XC7VX690T	XC7K325T	**XC7K325T**
Frequency (MHz)	200	100	200	214	-	200	200	**200**
Quantization	16-bit	16-bit	16-bit	8-bit	8/4-bit	8-bit	8-bit	**8-bit**
Strategy	floating	fixed	fixed	fixed	fixed/log	block floating	fixed	**floating**
DSP Used	1728	824	1518	780	-	1027	516	**768**
Throughput (GOPS)	202.42	229.55	715.9	84.3 (CONV)	878.11	760.83	354	**1086.8**
per DSP Throughput (GOPS/DSP)	0.117	0.279	0.472	0.444	-	0.741	0.69	**1.42**
Power (W)	10.81	9.4	-	3.5	10.56	9.18	8.23	**9.42**
Power Efficiency (GOPS/W)	18.72	24.42	-	24.1	83.15	82.88	43.0	**115.4**

Compared with the existing accelerators, our processor outperforms them in throughput, inference latency and per DSP throughput. Particularly, the average improvement of throughput is 63.5% and 1.45× for 325T and 7EV compared with [MCV18], respectively. Meanwhile, compared with RNA, we can achieve 38.6% and 1.08× better throughput for 325T and 7EV, respectively. Moreover, the average improvement of per DSP throughput is 2.2× and 3.9× compared with [MCV18] for 325T and 7EV, respectively. In the approach proposed in [LWC18], they use LUT to implement multipliers, so we do not compare per DSP throughput with them. For the comparison with CPU on ResNet50 which is only reported by OpenVINO, our FPGA processor outperforms 82.3% in terms of throughput because of the high parallelism in our processor. Although Nvidia Xavier NX can achieve higher throughput (due to more hardware resources) in most cases than our processor does, their computation efficiency is low as their peak throughput is reported to be 21TOPS [Inc20].

Comparison with Previous Accelerators on VGG16. We run the classification network VGG16 on our processor, and compare the results with seven typical studies, as shown in Table 3.6. We also list the detailed implementation information, such as platform, working frequency and quantization strategy in Table 3.6. First, our processor, which uses the

LPFP quantization scheme, has a negligible top-1 and top-5 accuracy degradation. Although the work in [MLN17] and [LWC18] can also maintain negligible accuracy loss, the approach in [MLN17] uses 16-bit floating-point data format, which results in higher bandwidth and memory requirement and lower per DSP throughput, while the approach in [LWC18] needs 144 extra hours for the fine-tuning process. Second, our processor outperforms all the six accelerators in terms of throughput and per DSP throughput. Particularly, the improvements of throughput and per DSP throughput are from 24% to 11.89× and from 92% to 11.14×, respectively. These improvements mainly come from the parallel computation pattern in FPFU and the implementation of four 4-bit MACs within one DSP slice. To the best of our knowledge, this is the first work that can simplify the multiplication to 4-bit and implement four MACs inside one DSP slice while maintaining comparable top-1/top-5 accuracy without any retraining process. Finally, we also show the power efficiency in Table 3.6, and our processor improves the power efficiency by 39% – 5.16×.

Comparison with Previous Accelerators on YOLO. We further compare the detection network YOLO [RDG16b, RF17b] with prior accelerators [MCG17, GHY17, GSQ17, WMS18] and we use the tiny version of the YOLO network. The comparison results are shown in Table 3.7, where we also list the mean average precision (mAP) loss of our quantized networks. Compared with the full precision network, the mAP loss of quantized tiny-yolo and tiny-yolo-v2 is 0.3% and 0.1%, respectively. The hardware comparison with prior accelerators shows that our processor is 20.1× and 49.7× higher in terms of throughput for tiny-yolo and tiny-yolo-v2, respectively. Moreover, due to the implementation of four 4-bit MACs within one DSP slice, the per DSP throughput improves by 5× compared with prior accelerators on average.

Table 3.7: Comparison with prior accelerators on YOLO. "-" means no reported results.

	Ma, et.al. [MCG17]	Aristotle [GHY17]	Angel-Eye [GSQ17]	Wai, et.al. [WMS18]	OPU [YWZ19a]	**ours**	
Year	2017	2017	2018	2018	2019	**2019**	
Platform	XC7V485T	XC7020	XC7Z020	Cyclone V	XC7K325T	**XC7K325T**	
Frequency (MHz)	143	214	-	117	200	**200**	
Quantization Strategy	16-bit fixed	8-bit fixed	8-bit fixed	8-bit fixed	8-bit fixed	**8-bit floating**	
Network	tiny-yolo	tiny-yolo	tiny-yolo	tiny-yolo-v2	tiny-yolo	**tiny-yolo**	**tiny-yolo-v2**
mAP loss (%)	-	-	-	-	1.3	**0.3**	**0.1**
DSP Used	112	198	-	122	516	**768**	
Throughput (GOPS)	48	36.5	62.9	21.6	366	**987.2**	**1095.4**
per DSP Throughput (GOPS/DSP)	0.429	0.184	-	0.177	0.71	**1.29**	**1.43**

3.6 Related Work

Weight and Computation Reduction. CNNs are typically over-parameterized, and extensive accelerator developers in recent years focus on using CNN approximation algorithms, including weight reduction, computation complexity reduction and quantization to accelerate CNN inference [WDZ19a]. The accelerator proposed in [LLX17, WCC18] uses Winograd algorithm to reduce the number of multiplication in convolution, thus reducing computation complexity. EIE [HLM16a], Cambricon-X [ZDZ16a] and Cambricon-S [ZDG18] are the mainstreaming accelerators that benefit from weight and computation complexity reduction techniques. Unnecessary computations (*i.e., multiplication of zeros*) in CNN inference are skipped for better inference time and energy [AYS18, SZH18]. All these methods for computation reduction take the 8-bit fixed-point as the target data representation. However, the irregularity caused by these algorithms degrades the parallelism and hardware efficiency [WLD18].

Quantization. Accelerators with quantization is another concentration. XNOR_Net [ROR16] applied weights binarization by quantizing weights into {-1, 1} with a scaling

factor for AlexNet. The lightweight YOLOv2 [NYF18] is another binarization approach which focuses on object detection CNN. Accelerator with ternary representation, which adds zero to the binary set, is introduced to help improve the accuracy [PBP17]. Although these accelerators achieve remarkable power and storage saving, they both suffer from significant accuracy loss. Moreover, they all need time-consuming retraining process to compensate for the quantization error. 16-bit quantization oriented accelerators, including floating-point and fixed-point representations, solve the problem of accuracy loss [MLN17,XLL17b,MCV18, MCG17]. However, the storage requirement is still huge, and the per DSP throughput is extremely low (less than 0.5GOPS/DSP) because of the usage of 16-bit.

8-bit quantization makes a trade-off between storage and accuracy. The accelerators [GSQ17, LWC18] optimize the computation patterns with 8-bit fixed-point quantization to improve the performance for different CNNs. DNNBuilder [ZWZ18b] is proposed to automatically build DNN accelerators to satisfy the performance and power efficiency demands on embedded and cloud FPGAs, while Cloud-DNN [CHZ19] is the framework for mapping DNN models to cloud FPGAs. Block floating-point scheme with 8-bit mantissa is used in [LLS19] to accelerate the inference of CNN while maintaining accuracy. However, all these accelerators need 8-bit MAC to perform convolution, leading to a low per DSP throughput (less than 0.8GOPS/DSP). A more aggressive method quantizes the small values of the weights into 4 bits and keeps the remaining 16 bits as full precision, by dividing the weights into the low-precision and high-precision regions according to the values of the weights [PKY18]. HAQ [WLL19a] proposes a mixed precision quantization approach with a trade-off between quantization policy and hardware performance. However, both studies need time-consuming retraining process to compensate for quantization errors.

Different from all the above methods, the proposed LPFP quantization scheme fully exploits the properties of weights and activations, thus obtaining a comparable or better accuracy for *deep* CNNs. Moreover, the LPFP quantization method gets rid of the time-consuming retraining process that needs labelled data and extra computing, because access

to labelled data can be difficult in practice as hardware and CNN algorithms are often developed by different parties. Furthermore, with the help of the LPFP quantization method, our processor only needs 4-bit MACs, thus dramatically improving the per DSP throughput. Moreover, our LPFP quantization method can also be applied to the aforementioned computation reduction methods. This is because the LPFP and fixed-point data representation share the same representation of zeros, and the aforementioned computation reduction methods all focus on eliminating the computations on zeros. Overall, the proposed processor achieves better performance on FPGA.

3.7 Conclusion

We have proposed a low precision floating-point quantization method, called LPFP, to reduce memory size and memory access with negligible accuracy degradation (less than 0.5% for top-1 and 0.3% for top-5 accuracy) for CNN interference. Furthermore, we have reduced the bit width for multiplication to 4-bit with comparable accuracy and implemented four 4-bit MACs within one DSP48E1 slice in Xilinx Kintex 7 FPGA family or DSP48E2 in Xilinx Ultrascale/Ultrascale+ FPGA family. Experiments using Xilinx KC705 and ZCU106 platforms and six typical CNN networks show that we achieve an average throughput and per DSP throughput of 1100.4 GOPS, 1.43 GOPS 1650.6 and 2.15, respectively. Moreover, the average throughput for these networks is 82.3% and 1.5× over Intel I7-8700T CPU and existing accelerators, respectively. Particularly for VGG16 and YOLO, we outperform six existing accelerators in terms of average throughput by 3.5× and 27.5×, while improving per DSP throughput by 4.1× and 5×, respectively. To the best of our knowledge, this is the first in-depth work that can simplify the multiplication to 4-bit and accommodate four MACs in one DSP slice while maintaining comparable top-1/top-5 accuracy without any retraining.

CHAPTER 4

MP-OPU: A Mixed Precision FPGA-based Overlay Processor for Convolutional Neural Networks

4.1 Introduction

Recent deep convolutional neural networks (CNNs) are widely used in real-time applications, such as autonomous driving, where model size and inference latency are the main constraints. Many researchers have introduced low-precision quantization techniques to reduce the model size, computation complexity and communication bandwidth so that the inference latency is reduced [HMD15b]. However, conventional quantization approaches take the same bit width of weights and activations for all layers, and 8-bit fixed point has been proved effective in hardware implementation while maintaining accuracy [YWZ19b, WWC20]. Lower precision networks are also possible to achieve high accuracy, but require expert design and re-training [ZHM16b]. To this end, mixed precision comes to be a significant solution that assigns different precision for different layers and different networks [WLL19b]. However, the variety of the precision of each layer makes the architecture design more challenging.

Trying to overcome the challenge, Nvidia releases Turing GPU architecture which supports mixed precision arithmetic operations (1-bit, 4-bit, 8-bit and 16-bit) [MDL18]. BitFusion [SPS18a] is another accelerator which can support multiplication on 2, 4, 8 and 16 bits. However, these architectures cannot fully leverage the advantages of the quantized model. For example, when the network is quantized to be 6-bit, the model has to be modified to 4-bit or to 8-bit when mapped on these architectures, which either leads to accuracy reduc-

tion or latency increment. On the other hand, bit-serial multipliers, which can support more flexible mixed precision multiplications, are used in Bismo [URS18]. However, Bismo uses large amount of look-up-tables (LUTs) and Block RAMs (BRAMs) to implement matrix multiplication on the FPGA, which makes it difficult to support CNNs.

To this end, we propose the Mixed Precision FPGA-based Overlay Processor (called **MP-OPU**) that effectively accelerates the inference of mixed precision models. By using the similar instructions and compilation flow as Light-OPU [YZW20a], we redesign the computation core and memory system to expand the support from 8-bit fixed point CNNs to 2-bit to 8-bit mixed precision CNNs. More specifically, the computation core can be configured to operate with different number of multipliers according to the given bit width of activations and weights, while the memory system in ping-pong architecture is capable of run-time data rearrangement to fully utilize the bandwidth of external memory.

To summarize, the main contributions of the proposed **MP-OPU** can be summarized as follows:

- **High Flexibility.** Different from the previous work that only support a subset of the the possible precision choices [MMN18], our **MP-OPU** manages to address mixed precision CNNs varying from 2-bit to 8-bit. In our design, the computation core is developed to be programmable during run-time to support mixed precision efficiently. By setting different parameter registers in the instructions, **MP-OPU** can support mixed precision without re-implementing the design. Moreover, the memory system is designed with run-time data pre-fetch and placement modules to optimize the external memory access for mixed precision.

- **High Scalability.** Our implementation is highly scalable as it can be easily scaled up or down to different FPGAs by adding or removing PEs. We only implement batch parallelism among different PEs so that the PEs are separate to each other and can be simply added or removed.

- **High Performance.** With all the DSPs configured to support 2-bit multiplication, **MP-OPU** can reach 4.92 TOPS peak throughput on Xilinx VC709 evaluation board. Furthermore, we take conventional and lightweight CNNs as benchmarks to be tested on **MP-OPU**. All these networks are accelerated and **MP-OPU** achieves 12.9× latency reduction and 2.2× better throughput/DSP for conventional CNNs, while 7.6× latency reduction and 2.9× better throughput/DSP for lightweight CNNs on average compared with existing FPGA accelerators, respectively.

4.2 Background and Related Work

4.2.1 Low Precision Quantization

Extensive explorations have been made on compressing and accelerating neural networks by using quantization. Deep compression methods [HMD15b] quantize the network weights to reduce the model size by rule-based strategies. More aggressive quantization strategies use 1-bit or 2-bit to represent the weights [CHS16]. Neural architecture search (NAS) based mixed precision quantization is also proposed to improve the performance and efficiency of quantizing a network. FBNet [WDZ19b] builds a lookup table of latency for different operations running on the hardware, and optimizes the latency during the design process. HAQ [WLL19b] computes the hardware feedback directly from two customized accelerators and optimizes the quantization policy until the resource constraints (*e.g.,* latency, energy) are met. All these studies manage to quantize the full precision network (32-bit) into low precision (less than or equal to 8-bit) with negligible accuracy loss. Therefore, we utilize the quantization results from these methods and the quantization approach is not included in the scope of this work.

4.2.2 Fixed Precision Processors/Accelerators

Customized processors/accelerators on FPGA are proposed to accelerate the inference of the quantized networks. OPU [YWZ19b] accelerate 8-bit fixed point CNNs, while the work in [WWC20] addresses 8-bit floating point CNNs. Light-OPU [YZW20a] expands the work from conventional CNNs to light-weighted CNNs with 8-bit fixed point as well. Different from these studies, our work focuses on mixed precision CNNs.

4.2.3 Mixed Precision Processors/Accelerators

Mixed precision processors or accelerators on FPGA have also been also proposed recently. Moreover, power-of-2 and fixed point data are supported by the auto-generated accelerators on FPGA [ZGG19]. The approach in [MMN18] keeps the activations to be 8-bit fixed point while the weights vary from 1 to 16-bit. However, they only support a limited part of the bit width combinations, while **MP-OPU** is optimized for all the combinations of low precision (2 to 8 bits).

4.3 Micro-Architecture of MP-OPU

The micro-architecture of **MP-OPU** discussed in this section is shown in Fig. 4.1, consisting of *Computation Core and Memory System*. In the *Computation Core*, the *PE Array and Post Process* modules perform the computation for all the conventional convolution, depth-wise convolution, fully-connected, pooling, activation, residual and concatenation layers. In the *Memory System*, we have 4 on-chip buffers, which are designed in ping-pong manner, to perform data communication with the external memory. Moreover, in each module, we have a separate control module to update the parameter registers defined in instructions, as well as to control the data flow in these modules.

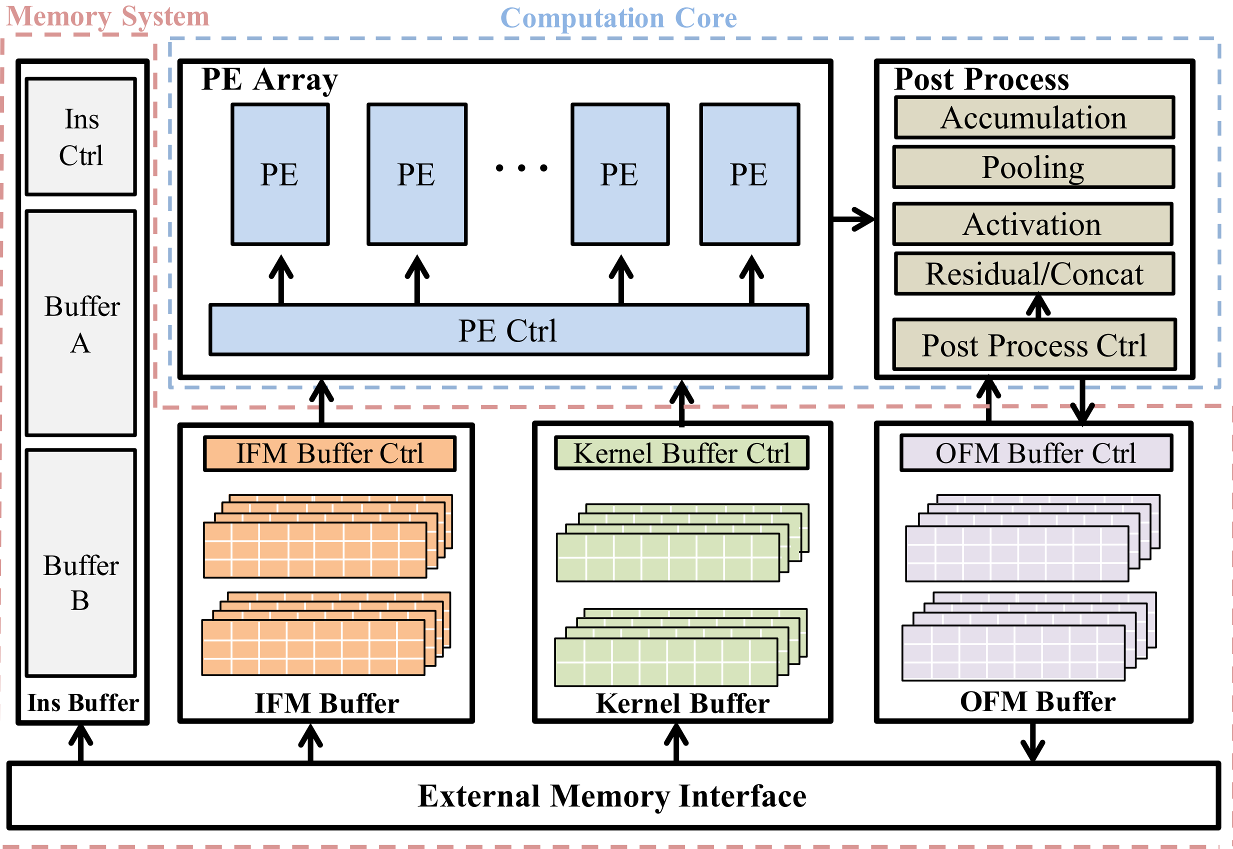


Figure 4.1: The micro-architecture of **MP-OPU**. The computation core has multiple PEs for convolutional layers followed by a post process module for other types of layers. The memory system is designed with on-chip buffers in ping-pong manner to save communication time with the external memory.

4.3.1 Computation Core

In the *Computation Core*, the *PE Array* finalizes the convolution of one layer block, while the *Post Process* handles the accumulation of the intermediate results of different layer blocks and other types of layers, such as pooling, residual and concatenation layers. To fully leverage the advantages of mixed precision CNNs, both the *PE Array* and the *Post Process* modules are designed to support mixed precision operations and can be programmed by the parameter registers defined in the instructions.

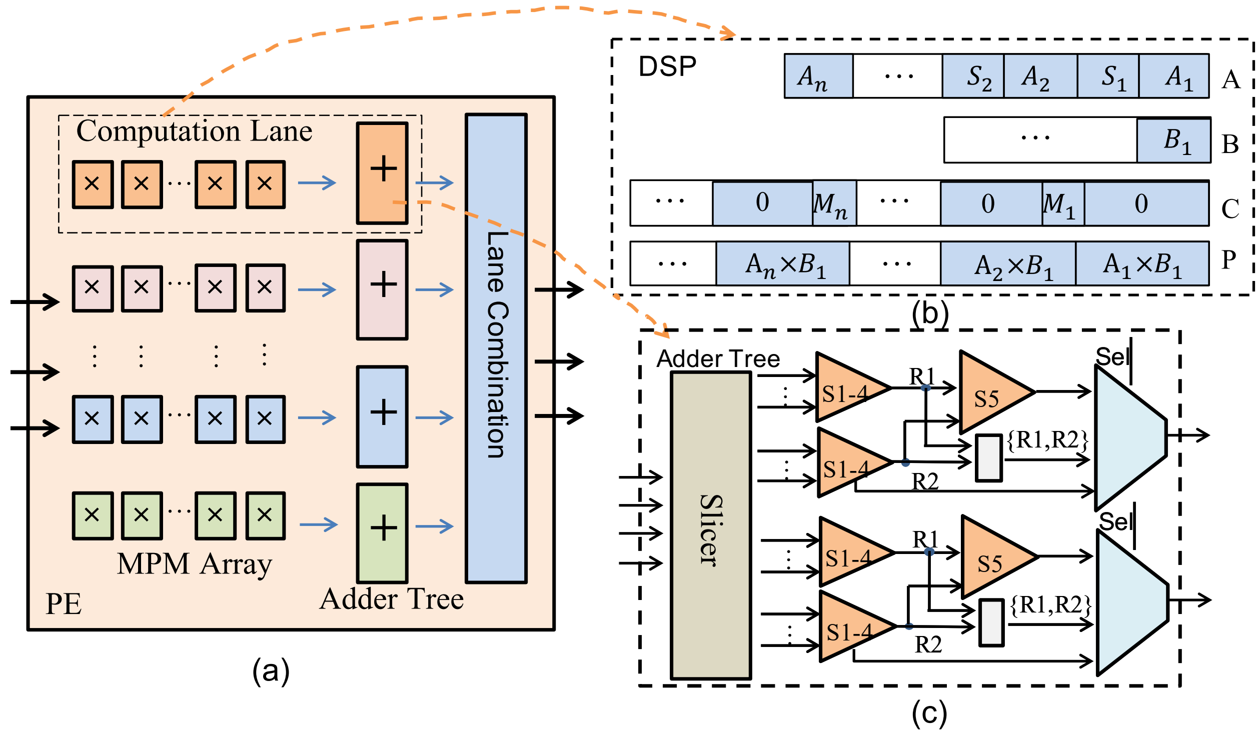


Figure 4.2: The architecture of one PE.

4.3.1.1 Architecture of PE

As shown in Fig. 4.2(a), each PE has multiple computation lanes to perform computation in parallel, and one lane combination module is followed to combine or select the results according to different parallelism modes. Each computation lane is composed of a mixed precision multiplier (MPM) array and an adder tree. In order to fully utilize the resources on FPGA, we use the DSP slices to implement the mixed precision multipliers, and we decompose one DSP slice into several low precision multipliers to increase the resource usage efficiency. As the weights and activations are always quantized by different bit width in a CNN, we propose the decomposition rule to support different bit width of multiplier and multiplicand, as shown in Fig. 4.2(b). We share the same multiplier B_1 and use n multiplicands (A_1, A_2 to A_n) to compact n multipliers into one DSP. In our implementation, we configure each DSP to perform as a multiply-adder ($P = A \times B + C$), and fit different operands (A, B

and C) to the multiply-adder according to different precision combinations. As the signed multiplication of lower significant bits may impact the higher significant bits, we introduce a modification bit for each low precision multiplier. We explore all the combinations of A_i and B_1 in an exhaustive way and conclude that the ith modification bit follows:

$$M_i = \begin{cases} Sign(B_1) \oplus Sign(A_i) & B_1 \neq 0 \text{ and } A_i \neq 0 \\ 0 & \text{otherwise} \end{cases}, \tag{4.1}$$

where $Sign(x)$ indicates the sign of the data x and $\oplus$ means xor. In this way, we just need to slice P into n parts to have the product of each low precision multiplier.

For each computation lane in **MP-OPU**, the number of DSPs is set to be fixed. Since each DSP can be decomposed to multiple low precision multipliers, the number of outputs for the MPM array varies according to the number of multipliers. Therefore, the following adder tree is also designed to be programmable by the parameter registers, as shown in Fig. 4.2(c). In order to use unified bit width for the adders in the adder tree, we first do slicing and sign extension for each input in the slicer. The aligned data are then fed into the 4-stage adder tree. Afterwards, the sum of two 4-stage adder trees are either summed up or concatenated according to the computation mode. Moreover, some stages of the adder tree can also be bypassed in order to support depth-wise convolution.

4.3.1.2 Parallelism Exploration

In order to speed up the calculation of the CNN, we use four levels of parallelism in the **MP-OPU**, as shown in Fig. 4.3. For the output channel parallelism, we use one activation to multiply different kernels to generate different activations of different output channels, as shown in Fig. 4.3(a). In the input feature map parallelism (shown in Fig. 4.3(b)), we will compute the convolution of different activations in the same input feature map with the corresponding kernels to produce one activation of one output feature map. For the

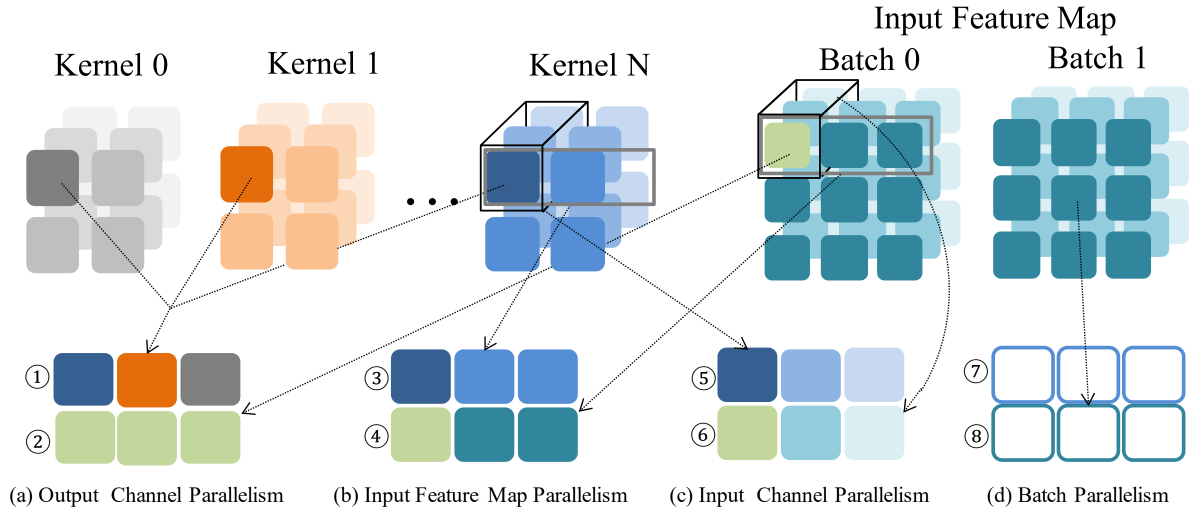


Figure 4.3: Four levels of parallelism in **MP-OPU**.

input channel parallelism (shown in Fig. 4.3(c)), multiplication of activations in different input channels and corresponding kernels are computed in parallel and then summed up to form a result (or intermediate result) of one output channel. The batch parallelism will calculate activations from different input images in parallel. These four levels of parallelism are selected according to different types and parameters of the layers. For example, for the conventional convolutional layer, we will use input channel parallelism in each computation lane, and use output channel parallelism among different computation lanes in each PE. Since each output feature map channel is generated by convolving one kernel and one input feature map channel in the depth-wise convolutional layer, we use the input feature map parallelism in each computation lane and output channel parallelism among different computation lanes in each PE. To simplify the design, we set all the PEs to be independent and only do batch parallelism among different PEs.

4.3.2 Data Pre-fetch and Placement for on-chip Memory

Since the maximal bandwidth of the external memory is fixed, we use ping-pong architecture and data pre-fetch module to increase the run-time bandwidth of the external memory. With

Table 4.1: Comparison with customized FPGA accelerators/processors on conventional CNNs.

	OPU1024 [YWZ19b]	[ZGG19]	[APR20]	LPFP [WWC20]	**MP-OPU**	
Year	2019	2019	2020	2019	**2021**	
Device	XC7K325T	2 XC7VX690T	XC7VX485T	XC7K325T	**XC7VX690T**	
Network	VGG16	VGG16	Tiny-Yolo-V3	VGG16	**VGG16**	**Tiny-Yolo-V3**
Bit width	8	mixed [1]	18	FP8	**mixed**	
Frequency (MHz)	200	156	200	200	**200**	
DSP Used	516	-	2304	768	**3072** [2]	
Inference latency (ms)	88.7	200.9	-	29.6	**11.2**	**2.91**
Throughput/DSP (GOPS)	0.68	-	0.2	1.42	**0.90**	**0.62**

[1] Weights are mixed precision while activations are 8-bit.

[2] 2048 DSPs are used for low precision multipliers while 1024 DSPs are used for adders.

ping-pong architecture, the communication time can be hid under the computation time. The data pre-fetch module manages to load as many data with different precision as possible. As different parallelism levels require activations and weights to be placed in different orders, we arrange and store the data accordingly in the memory system in advance. On the other hand, the feature maps are arranged during run-time to fit different parallelism levels. As the output feature map of one layer will be the input of another layer, we add an extra data rearrangement logic to change the data arrangement between row major and channel major according to the parallelism levels. Only when the parallelism level changes between two adjacent layers, will the data rearrangement logic be enabled.

4.4 Experiments

4.4.1 Experimental Setup

The proposed **MP-OPU** is implemented on the Xilinx VC709 evaluation board with an XC7VX690T FPGA. The design is described in Verilog-HDL, synthesized and implemented with Vivado 2020.1. For the network benchmarks, we use both conventional and lightweight

Table 4.2: Resource Utilization of **MP-OPU** on XC7VX690T.

Resource	LUT	LUTRAM	FF	BRAM	DSP
Used	278548	42853	324033	912	3072
Available	433200	174200	866400	1470	3600
Utilization	64.3%	24.6%	37.4%	62.0%	85.3%

CNNs for a comprehensive comparison to show the effectiveness of **MP-OPU**. These CNNs cover different kernel sizes (1×1 and 3×3), strides (1×1 and 2×2), and convolutional layer types (conventional convolution and depth-wise convolution). In addition, irregular layer operations such as residual and concatenation are also included.

4.4.2 Hardware Implementation

In this work, 8 PEs are implemented. To balance the usage of DSPs and LUTs, we use 256 DSPs to implement mixed precision multipliers and 128 DSPs to implement mixed precision adders in each PE. All the 256 DSPs for multipliers are configured by the same parameter registers to support multipliers with bit width varying from 2 to 8 bits. The processor is designed to meet 200MHz timing constraints and the detailed resource utilization is listed in Table 4.2.

4.4.3 Comparison with Customized FPGA Accelerators

We further compare **MP-OPU** with 6 FPGA accelerators/processors on both conventional and lightweight CNNs, and the results are shown in Table 4.1 and Table 4.3. As the quantization method is not included in this paper, we use the bit width generated by HAQ [WLL19b]. We use the **inference latency** to evaluate the performance of running the network on each FPGA processor/accelerator. **Throughput/DSP**, which is defined as the throughput conducted by each DSP during run-time, is utilized to indicate the efficiency of each design.

Table 4.3: Comparison with customized FPGA accelerators/processors on MobileNetV1/V2.

	[ZNL18]	[BZH18]	[ZGG19]	Light-OPU		**MP-OPU**	
Year	2018	2018	2019	2020		**2021**	
Device	Stratix V 5SGSD8	Arria 10 SoC	2 Stratix 10	XC7K325T		**XC7VX690T**	
Network	V1	V2	V1	V1	V2	**V1**	**V2**
Bit width	16	8	mixed [1]	8		**mixed**	
Frequency (MHz)	133	150	156	200		**200**	
DSP Used	1641	1278	-	704		**3072** [2]	
Inference latency (ms)	4.33	3.76	0.32	3.78	3.07	**0.47**	**0.34**
Throughput/DSP (GOPS)	0.13	0.06	-	0.21	0.14	**0.38**	**0.29**

[1] Weights are mixed precision while activations are 8-bit.

[2] 2048 DSPs are used for low precision multipliers while 1024 DSPs are used for adders.

The image size is $416 \times 416 \times 3$ for Tiny-Yolo-V3 and $224 \times 224 \times 3$ for the other networks.

As shown in Table 4.1, **MP-OPU** achieves 7.9× and 17.9× inference latency reduction on VGG16 compared with OPU1024 and the accelerator in [ZGG19], respectively. In addition, **MP-OPU** outperforms OPU1024 on throughput/DSP by 1.3×. As for Tiny-Yolo-V3, we have 3.1× better throughput/DSP than the prior accelerator. The approach in [ZGG19] does not report the DSP utilization, but they use two XC7VX690T FPGAs while we only use one. For MobileNet-V1/V2, **MP-OPU** performs 6.0× and 10.1× reduction on latency on average compared with existing works, respectively. Although we use more DSPs than others, we still have 2.4× and 3.5× better throughput/DSP on average, respectively. Better throughput/DSP indicates higher computation efficiency during run-time, and this is one main reason why we have lower inference latency.

4.4.4 Discussion

To further demonstrate the effectiveness of **MP-OPU**, we evaluate the inference latency with respect to different bit width combinations. We take Tiny-Yolo-V3 as an example and

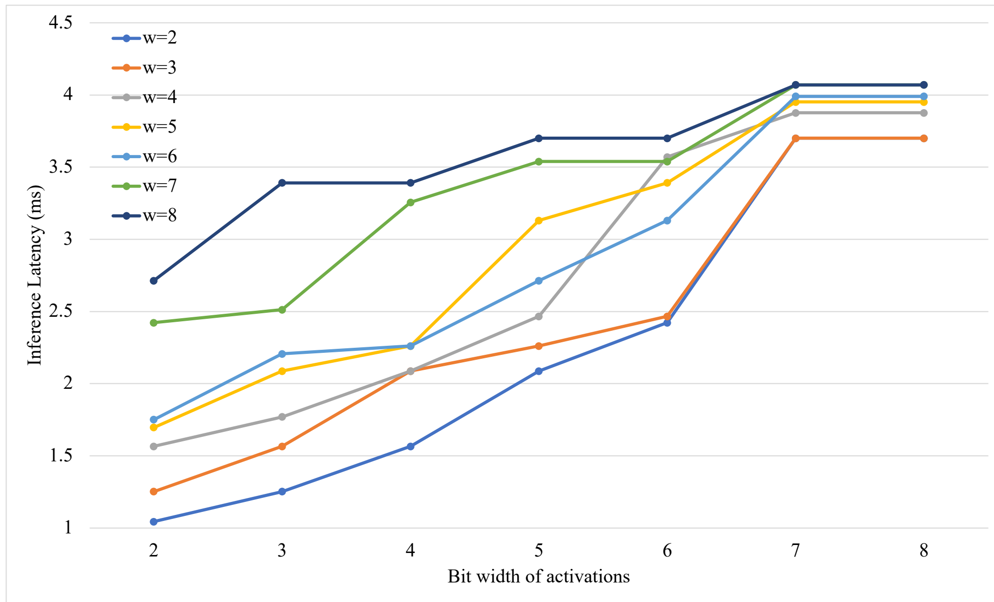


Figure 4.4: Inference latency of Tiny-Yolo-V3 with respect to different combinations of bit width. In the legends, "w=2" means the bit width of weights is 2.

the results are shown in Fig. 4.4. In general, the latency increases as the bit width increases. The observation follows the rule that one DSP can be decomposed to more multipliers when the target bit width is small. However, the latency remains unchanged when the bit width changes from 7-bit to 8-bit because one DSP can only be decomposed into two multipliers for both 7-bit and 8-bit. Furthermore, compared with the 8-bit model, the inference latency of the 2-bit model reduces to about 3.9×, although one DSP can only be decomposed to 3× as many 2×2 multipliers as 8×8 multipliers. The extra benefits come from the data pre-fetch module, where we manage to fetch 4× as much 2-bit data as 8-bit data under the same external memory bandwidth. For the 8-bit case, the bandwidth constraints is still severe as we cannot hide all the external communication time under the computation time, especially for the layers with 1×1 kernel size. Therefore, the percentage of the DSPs being effective during run-time is less than 100%. The problem of bandwidth constraints can be alleviated when the activations and weights decreased to 2-bit.

4.5 Conclusions

In this work, we propose a Mixed Precision Processor on FPGA (**MP-OPU**) to leverage the advantages of mixed precision CNNs. We reuse part of the instructions and compilation flow in Light-OPU and redesign the hardware processor. To support mixed precision CNNs, the computation core is designed to be run-time re-configurable to have different number of multipliers according to the given precision. Meanwhile, the ping-pong architecture, pre-fetch and data rearrangement logic in the memory system make fully utilization of the bandwidth of the external memory. By mapping on Xilinx VC709, **MP-OPU** can reach 4.92 TOPS peak throughput when configuring to only support 2-bit. Our experimental results show that **MP-OPU** manages to reduce inference latency by 12.9× and increase throughput/DSP by 2.2× for conventional CNNs on average, respectively. Also, the average latency reduction is 7.6× and the throughput/DSP increment is 2.9× for lightweight CNNs.

CHAPTER 5

LW-GCN: A Lightweight FPGA-based Graph Convolutional Network Accelerator

5.1 Introduction

Over recent years, deep learning paradigms such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs) have shown great success in various families of tasks such as image and text processing [ON15, SMH11]. However, these paradigms rely heavily on structural properties of euclidean data such as dense tensors, and have trouble processing non-euclidean data such as graphs. To tackle this problem, graph neural networks (GNNs) have been introduced and have demonstrated the ability to accurately process complex graph data [SGT08]. Among numerous GNNs, graph convolutional networks (GCNs) [KW16a], which borrows ideas from CNNs to aggregate neighbor data, have quickly attracted industrial attention as a popular solution to real-world problems [SZG20a, BA19, YHC18a, CLS19a]. Since then, there have been many other graph processing algorithms (*i.e.* GIN, GraphSAGE, GAT, etc.) introduced to optimize performance on existing problems and extend to new challenges [XHL18, HYL17c, VCC17, TWO18, SKB18, FLW18].

Similar to CNN, GCNs also contain multiple layers, where the main operations of each layer are *combination* and *aggregation*. *Combination* is similar to a dense layer of a multi-layer perceptron (MLP), where a feature matrix is multiplied by a weight matrix. *Aggregation* is similar to a convolution operation of a standard CNN, where the feature vector of each vertex is computed through a weighted aggregation of all feature vectors of neighboring

vertices, which can be represented as a matrix multiplication between the graph adjacency matrix and the feature matrix.

Despite the fact that the majority of GCN operations can be represented as matrix multiplication, it is unlikely for existing matrix multiplication oriented accelerators [YWZ19c, FR04, YZW20b, LDT16b] to yield high throughput on GCN. These accelerators typically exploit the structured nature of dense tensors and apply data reuse techniques to achieve performance boosts. However, such techniques are ineffective in GCNs because adjacency matrices in GCNs are often sparse, random, and irregular due to the fact that node degree distribution of random graphs follow the power law distribution. Although existing works such as EIE and Cambricon-X [HLM16b, ZDZ16b] tackles irregularity in computation and memory access in deep compressed CNNs, the sparsity of deep compressed CNNs is much lower (around 90%) than that of GCNs (over 99.9%). Due to the extreme sparseness of graph data, sparse CNN accelerators also fail to maximize computational efficiency, thus a more effective approach is required.

There are existing GCN accelerators to overcome the sparseness challenges [GLS19, LWL20a, YDH20a, LLK21]. The work EnGN [LWL20a] proposes a unified architecture for feature extraction, aggregation and operation update. The GCNAX also proposes a unified architecture while focusing specifically on loop rearrangement to improve the efficiency of loading data from off-chip. The Cambricon-G [SZF22] proposes the cuboid engine with multiple vertex processing units and hybrid on-chip memory to process the sparse data and dynamically update the graph topology. Meanwhile, the Rubik [CWX20a] develops a unified architecture to cooperate with graph reordering to support both node-level and graph-level computing. On the contrast, the work in [YDH20a, GLS19] assumes combination and aggregation are structurally different. In HyGCN, they design an aggregation engine for irregular accesses and computations, and an combination engine for regular accesses and computations. The AWB-GCN uses TDQ-1 and TDQ-2 to perform general sparse (sparsity $< 75\%$) matrix multiplication and ultra-sparse matrix multiplication, respectively. Although the

above work achieves performance boosts, they either cache large amounts of data on-chip or rapidly load data from off-chip memory. This requires either large amounts of on-chip memory or huge off-chip memory bandwidth (*i.e.*, high-bandwidth memory (HBM)). Moreover, HyGCN and AWB-GCN deploys independent hardware modules for different operations, such as combination and aggregation. Despite the efforts to balance computation in each module, the inherent workload differences across datasets make it difficult to keep both modules fully utilized. Moreover, as GCN grows in popularity and supports numerous real-world applications, it is natural for its inference workload to see heavy demand on edge devices in the near future. For example, GCN is used for autonomous exploration under uncertainty in the robotic domain [CWS19]. The authors in [AKT21] propose a GNN based algorithm to optimize pose prediction in 2D SLAM, which can be widely used in autonomous driving. It is unlikely for these resource limited devices to provide powerful hardware resources, therefore a more lightweight approach is required.

To this end, we propose a lightweight software-hardware co-optimized accelerator, named **LW-GCN**, to efficiently perform GCN inference. We first introduce the "packet" conception in compressing the sparse matrix into a packet-level column-only coordinate-list (PCOO) format in software. The PCOO format is also easy to decompress in the hardware. We then propose a unified micro-architecture to efficiently execute both *combination* and *aggregation*, where the main operations are MM and SpMM. An optimized computation pipeline is utilized in each processing element (PE) to cope with the irregularity in computation and memory access caused by SpMM. Due to the limited hardware resources, we apply tiling to process a portion of MM/SpMM at a time, which enables us to only keep a fraction of the matrices on-chip. Finally, our preprocess procedure injects "empty elements" in PCOO to indicate idle cycles and prevent data collisions caused by irregularity of the sparse matrix in software side. The preprocess algorithm has linear time and space complexity with respect to the number of elements in the sparse matrix.

We implement **LW-GCN** onto the Xilinx Kintex-7 K325T FPGA, which simulates the

Table 5.1: Dimensions and densities of widely-used datasets.

Datasets	Nodes	Edges	Input Features	Classes	Feature Density	Edge Density	Weight Density
Cora	2708	10556	1433	7	1.27%	0.144%	100%
CiteSeer	3327	9104	3703	6	0.85%	0.0822%	100%
PubMed	19717	88648	500	3	10.0%	0.0228%	100%

limited resource availability of edge devices. We evaluate **LW-GCN** for GCN and GraphSAGE on three popular datasets Cora [CGS13], CiteSeer [CWC14] and PubMed [DL17]. Compared to state-of-the-art software framework Pytorch Geometric (PyG) running on Intel Xeon Gold 5218 CPU, NVIDIA Jetson Xavier NX edge GPU, NVIDIA RTX3090 GPU, and a prior FPGA-based GCN accelerator [GLS19], **LW-GCN** achieves up to 60×, 32×, 12×, and 1.7× smaller latency, as well as 912×, 84×, 511×, and 3.87× higher energy efficiency, respectively.

To summarize, the main contributions of this work as listed as:

- **Software-Hardware Co-optimization.** We propose a linear time and space preprocess algorithm to compress the sparse matrix into PCOO format and optimize the GCN workload. In addition, the micro-architecture is designed to efficiently process the PCOO format, so that the GCN workload is also optimized in hardware side.
- **High Computation Efficiency.** We design unified micro-architecture for MM and SpMM, which efficiently performs both *combination* and *aggregation* operations in GCN. Moreover, the PCOO format skips computation and storage of zeros in the sparse matrix, and the optimized architecture in each PE addresses the irregularity issue caused by sparse matrix, which further increase the computation efficiency of **LW-GCN**.
- **Low Resource Requirement.** The compression method in the preprocess algorithm reduces both the storage and bandwidth requirement. Moreover, **LW-GCN** utilizes

tiling techniques to process a portion of MM/SpMM at a time, thus further alleviating on-chip memory burdens. Different from prior works that rely heavily on large on-chip memory availability, **LW-GCN** works effectively on resource limited edge devices.

- **High Performance.** We evaluate **LW-GCN** on a Kintex-7 FPGA on three popular datasets. Our work reduces latency by up to 60×, 32×, 12×, and 1.7× and increases energy efficiency by up to 912×, 84×, 511×, and 3.87×, compared to Intel CPU, NVIDIA edge GPU, NVIDIA server GPU and prior FPGA-based GCN accelerator.

5.2 Challenges and Motivations

In this section, we will briefly introduce the GCN algorithm, the challenges to map it on hardware, and the motivation of our accelerator design.

5.2.1 GCN Background

The forward propagation of the lth layer of a multi-layer GCN [KW16a] is illustrated in Equ. (5.1),

$$X_l = Relu(AX_{l-1}W_l), \tag{5.1}$$

where A, X_l and W_l indicate the adjacency matrix of the input graph, the feature matrix of the lth layer, and the weight matrix of the lth layer, respectively. $Relu$ is the activation function and the input feature matrix of the graph is represented as X_0.

Based on our analysis on widely-used datasets, the adjacency matrices and the input feature matrix are often sparse, while the weight matrices are dense, as shown in Table 5.1. Therefore, the computation order influences dramatically on computation complexity when skipping the zeros. Following the analysis in [GLS19], we profile the required number of scalar operations and intermediate storage under different computation orders, as shown in

Table 5.2: Required computation and storage under different computation orders.

Datasets	$(A \times X_{l-1}) \times W_l$	$A \times (X_{l-1} \times W_l)$
Cora	18.7M / 56.2Mb	1.33M / 0.661Mb
CiteSeer	38.9M / 188Mb	2.23M / 0.812Mb
PubMed	118M / 150Mb	18.6M / 4.81Mb

Table 5.2. This way, we perform $A \times (X_{l-1} \times W_l)$ as it is much more efficient.

This is also true for GraphSAGE. The computation of a layer in GraphSAGE can be expressed as follows:

$$X_l = ReLU(X_{l-1}W_{l,1} + \hat{A}X_{l-1}W_{l,2}), \tag{5.2}$$

where $\hat{A}$ is preprocessed from adjacency matrix A by dividing each element by the number of non-zero elements in the row. In this way, we can perform the same optimized compute order of $\hat{A}(X_{l-1}W_{l,2})$ as that in GCN. Similarly, GAT and the first layer of GIN both contain similar sparse-dense-dense matrix multiplication workloads and can potentially utilize this optimization.

For simplicity, we refer to step $X_{l-1} \times W_l$ as *combination* and $A \times (...)$ as *aggregation* of each GCN layer, following the conventions of [YDH20a]. Moreover, for the *combination* of the first layer and *aggregation*, we perform SpMM and for the *combination* of other layers, we perform MM. This is because X_l is produced by the previous layer and it is always dense except the first layer. From here on, for SpMM we will refer to the left sparse input matrix as X, the right dense input matrix as W, and output matrix as Y for simplicity.

5.2.2 Challenges

As illustrated in previous sections, the main operations in GCN can be extracted as SpMM and MM. Moreover, as more and more edge applications, such as autonomous exploration in robots [CWS19] and pose prediction in 2D SLAM [AKT21], use GCNs for better perfor-

Algorithm 1 SpMM

1: **Input:** $X \in \mathbb{R}^{m \times n}$, $W \in \mathbb{R}^{n \times p}$, $Y \leftarrow 0^{m \times p}$
2: **for** $X_{i,j}$ in X **do**
3:     **if** $X_{i,j} \neq 0$ **then**
4:         **for** $W_{j,k}$ in W **do**
5:             $Y_{i,k} \leftarrow Y_{i,k} + X_{i,j} \times W_{j,k}$
6:         **end for**
7:     **end if**
8: **end for**
9: **return** Y

mance. Therefore, the challenge becomes to accelerate SpMM and MM on resource limited devices.

5.2.2.1 Challenges on SpMM

The computation of SpMM on one PE can be effective to skip all zero elements of the sparse input X, as shown in Algorithm 1. However, parallel computing with multiple PEs introduce new problems in **Computation Imbalance** and **Memory Irregularity**.

Computation Imbalance: To accelerate the computation of SpMM on multiple PEs, we will first divide the workload and distribute portions to multiple PEs. In each PE, we only process the non-zero elements from X. Due to irregularity in X, it is difficult to allocate identical workloads to every PE, which leads to computation imbalance. This is challenging for the SpMM in GCN, as the matrices in *combination* and *aggregation* are extremely sparse ($> 99\%$). Moreover, real-world graphs follows the power law distribution [XYL14], which implies that the minority of rows (columns) in the adjacent matrix have the majority of non-zeros while the majority of rows have only a few (not empty) non-zeros. Such irregularity further increases the difficulty to balance workload.

Memory Irregularity: Since the optimization of SpMM only stores the non-zero elements to save memory requirement, the data irregularity incurs several issues during computation. Firstly, it is difficult to predict the position of the next non-zero element $X_{i,j}$ to be processed in the left matrix. Since matrix multiplication matches $X_{i,j}$ against W_j and j is unknown, the next non-zero $X_{i,j}$ could require any row of W. This uncertainty requires us to cache the entire W matrix on-chip, which leads to very expensive caching. Secondly, parallel computing of SpMM will process multiple non-zero elements of X simultaneously, thus requiring all corresponding data in W to be readily available, this introduces the problem of bank conflict. For example, to process non-zero elements X_{i_a,j_a} and X_{i_b,j_b} simultaneously, the PEs must be provided with W_{j_a} and W_{j_b}. However, memory resources on FPGA usually come with high depth and very limited (1 or 2) ports, where each port can only access a single depth of the memory bank at a time. In the scenario where W_{j_a} and W_{j_b} are stored on the same bank, which can only supply one of them at a time, we face a data conflict. Thirdly, since the SpMM algorithm computes each row of the SpMM result as a sum of many scalar-vector multiplications, it introduces a read-after-write (RAW) conflict. This is due to the fact that arithmetic operations tend to take multiple cycles on hardware. If we process non-zero elements X_{i,j_a} followed by X_{i,j_b} in the immediate next cycle, the multiplication and addition would not have finished in the first cycle. When the PE reads in Y_i in the next cycle to process addition for X_{i,j_b} it would inevitably read in an incorrect result. Finally, although the RAW conflict can be effectively resolved by utilizing multiply-accumulators (MACs) instead of individual multipliers and adders, doing so restraints the design to use the same PE to process each row X_i, which leads back to the issue of **Computation Imbalance**. As the individual node degree in a random graph follows the power law distribution, it is common for there to be a large difference (over 100×) between densities of individual rows of an adjacency matrix. Naively partitioning the sparse input X into row-blocks and assigning row-blocks to a PE group would result in a difference between non-zero workload assigned to each PE within the group. The latency of the group would be controlled solely by the

input row with the highest density, vastly reducing efficiency.

5.2.2.2 Challenges on resource limited devices

Accelerating the inference of GCN should include the acceleration of both MM and SpMM. Although MM does not have the issues of **Computation Imbalance** and **Memory Irregularity** as SpMM, MM requires storage of all the numbers in the matrices, which incurs the issue of **Bandwidth Constraints**. Therefore, designing a module with both MM and SpMM in consideration is challenging. Existing solutions such as [YDH20a, GLS19] view MM and SpMM as inherently different workloads, therefore introduced dedicated modules to perform each independently. Although this allows each module to efficiently tailor toward its workload, the resource allocation for each module raises a non-negligible concern. Since different problem settings come with different data dimensions and densities (examples shown in Table 5.1), the ratio between arithmetic operations required in *combination* and *aggregation* varies significantly across datasets. In order to fully utilize the dedicated modules for each, these accelerators often need to dynamically allocate computation resources to each module for each problem setting, which consumes many hours or even days for the synthesis and implementation process. Moreover, the data dependency between *combination* and *aggregation* leaves one of the MM and SpMM modules idle, which leads to a waste of resources. Such problem makes the accelerating GCN on resource limited devices more challenging.

5.2.3 Motivation

Motivated by the above challenges, we propose a software-hardware co-optimization process to address each of them, while keeping an available resource budget of an edge device. We first define a PCOO format to compress the input sparse matrix, effectively eliminating zero elements to preserve both storage space and computation time. We then design a dedicated

computation engine processing multiple non-zero elements in parallel efficiently. Some key highlights of our design include:

- **Software Preprocessing:** We first compress the sparse data into PCOO format, and leverage the binary "edge-or-no-edge" feature of graph adjacency matrices to remove value data. Then, we search the space of the sparse matrix to balance the workload on different PEs in order to resolve the issue of computation imbalance. Finally, idle data insertion is applied to solve the problem of bank conflict with a small burden.

- **Dedicated Architecture Design:** We design a dedicated architecture to decompress the PCOO format in order to further increase the computation efficiency. Moreover, a multi-port memory is applied in our design to resolve the issue of data conflict from the hardware side.

- **Unified Micro-architecture:** We observe that MM is essentially a special case of SpMM where density is 1. Therefore, we design the algorithm to process SpMM by individual non-zero elements on the sparse matrix, and apply that algorithm on MM as well. Moreover, we design a unified architecture of PE to process both MM and SpMM efficiently, which allows all computation resource to be fully utilized. This allows the full GCN workload to be deployed onto a unified module, resolving the resource allocation problem.

- **Flexible Design:** Our design is not dedicated toward any specific GCN configurations, instead it is able to support any number of layers with any size of GCN layers. Additionally, since MM and SpMM are widely used across GNNs, our design supports most operations needed for many other networks. In Section 5.5 we also evaluate our design on GraphSAGE in addition to GCN as we support it out of the box.

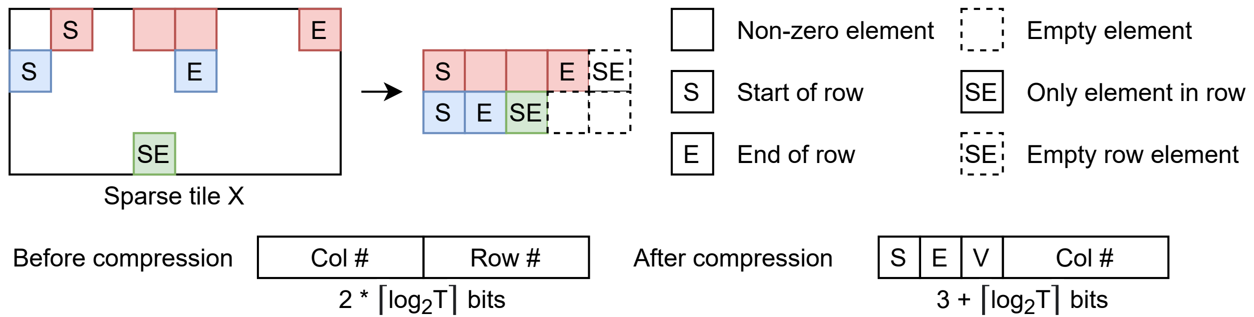


Figure 5.1: Packet-level column-only coordinate list format

5.3 Software Preprocessing

The software preprocessing algorithm will first compress the input data, then allocate and schedule GCN workloads onto different PEs. We will explain these algorithms in detail in this section.

5.3.1 Data compression

5.3.1.1 PCOO format

As shown in Table 5.1, the adjacency matrix and the input matrix of the first layer in GCNs are often extremely sparse. Therefore, we compress these matrices to process only valuable information (non-zero elements) to save storage and reduce computation complexity. We introduce the "packet" concept to propose a packet-level column-only coordinate-list (PCOO) format to compress the sparse matrix (Fig. 5.1). In detail, we treat all the elements in one row as one packet, and each non-zero element $X_{i,j}$ in one row is formatted into a bit-wise format. Firstly, the leading two bits conclude the row information of each non-zero element, which indicate the start-of-row (SOR) for first non-zero element and end-of-row (EOR) for last non-zero element. Secondly, the following one bit indicates valid (VLD) to differentiate from injected empty elements (the injected empty elements are explained in detail in Section 5.3.2). These three bits act as the header of a packet and the rest bits play

Table 5.3: Comparison of Storage Requirement between CSR, CSC, COO and PCOO.

Dataset	row	col	non-zeros	CSR	CSC	COO	PCOO
Cora Features	2708	1433	49216	781Kb	810Kb	1.33Mb	886Kb
Cora Edges	2708	2708	10556	207Kb	207Kb	296Kb	201Kb
CiteSeer Features	3327	3703	105165	1.74Mb	1.74Mb	2.94Mb	2.00Mb
CiteSeer Edges	3327	3327	9104	192Kb	192Kb	255Kb	173Kb
PubMed Features	19717	500	105165	13.2Mb	18.8Mb	27.7Mb	15.8Mb
PubMed Edges	19717	19717	9104	202Kb	202Kb	301Kb	195Kb

a role of payload, which has the column information and the value of each non-zero element. We use $log_2(T)$ bits, where T is the tile size, to represent the column position within tile (j mod T) of $X_{i,j}$. Finally, we use the remaining H bits to represent the value of the non-zero element. In the corner case where there are no non-zero elements in a given row, we set the header SOR = EOR = 1 and VLD = 0 with empty payload, in order to instruct the hardware to increment the row number without performing calculation. In this way, we totally need $3 + log_2(T) + H$ bits to represent each non-zero element in the sparse matrix. The algorithm of compressing sparse matrix with PCOO is concluded in Algorithm 2.

We also evaluate the storage consumption of the commonly used compression formats (CSR/CSC/COO) vs PCOO, and the results as shown in Table. 5.3. For all the datasets, PCOO format is comparable in terms of storage efficiency compared with CSR and CSC, and is more efficient than COO.

Since we treat MM the same operation as SpMM, we format the left dense matrix in MM to fit the unified PE (as expressed in Section 5.4). The dense matrix is first stored as normal, and then we design all rows to share the same column information in PCOO format. In this way, we only need an extra of $(3 + log_2(T)) \times Column_Size$ bits to store the dense matrix in intermediate steps.

Algorithm 2 Sparse matrix preprocessing

```
1: inputs: X ∈ ℝ^{m×n}, T, K
2: tiles, sor, eor, vld = [], T × 4, T × 2, T
3: for t ← 0 to n − 1 by T do
4:     rows ← [[] for 0:K]
5:     for i ← 0:m do
6:         row ← []
7:         for j ← t:(t + T − 1) do
8:             if X_{i,j} ≠ 0 then
9:                 row.append(j % T + vld)
10:            end if
11:        end for
12:        row ← [0] if row is empty else row
13:        row[0] ← row[0] + sor
14:        row[-1] ← row[-1] + eor
15:        rows[i % K].extend(row)
16:    end for
17:    fill zeros until rows is rectangular
18:    tiles.append(rows.transpose())
19: end for
20: return tiles
```

5.3.1.2 Quantization

In order to further reduce the memory consumption, we apply quantization onto the values of all the matrices in GCNs. There are existing quantization strategies for GNNs. Degree-Quant [TFL20a] can quantize to 8-bit signed fixed point with negligible accuracy loss. However, their quantization strategy is applied during the GCN training process.

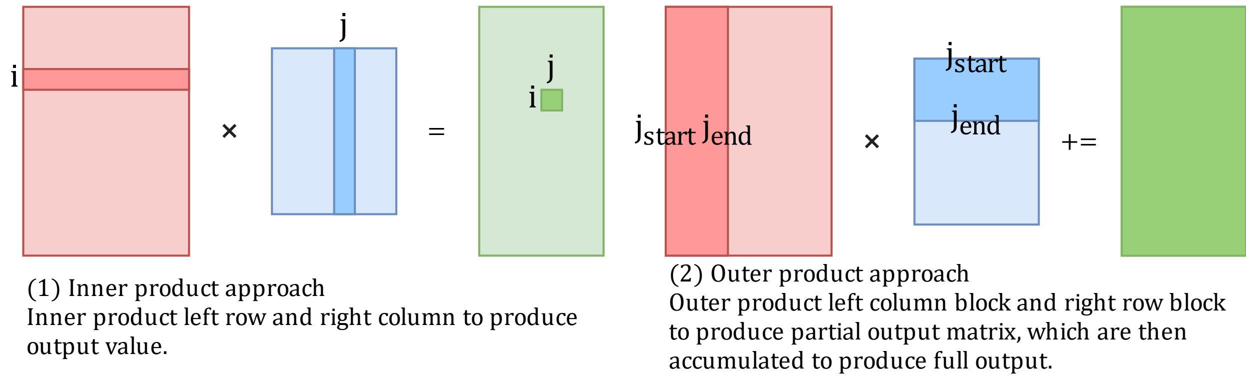


Figure 5.2: Outer product matrix multiplication

SGQuant [FWL20a] proposes a GNN-tailored quantization algorithm to reduce GNN memory consumption. However, they require a fine-tuning scheme to compensate for the accuracy loss caused by precision reduction. Our work only targets the inference phase and we target on reducing the running time for preprocess. In this way, we take post-training quantization strategy to save time for preprocessing. To maintain accuracy, we select 16-bit signed fixed point (SINT16) to quantize the features and weights. Moreover, we explore the data properties of the sparse matrices and use 4-bit signed fixed point (SINT4) to quantize the non-zero elements. In fact, for all matrices as well as two out of three feature matrices on the three popular used datasets, the value of each $X_{i,j}$ would be binary between 0 and 1, and there would be no accuracy loss at all. During the computation, we store all the intermediate results as 32-bit signed fixed point (SINT32) to maintain accuracy. The evaluation of our quantization strategy on both GCN and GraphSAGE on all three datasets shows that our proposed approach incurs negligible accuracy loss (within 0.2%).

5.3.2 Assignment and Scheduling

In order to reduce memory consumption, we employ an outer-product tiling approach, as shown in Fig. 5.2. We partition the inputs into T-column tiles for X and T-row tiles for W. The hardware processes a pair of tiles at a time, and produces the final result by

accumulating all tile results. For each pair of tiles, we perform the following preprocessing steps to balance workload and reduce data volume.

5.3.2.1 Workload assignment and scheduling

Multiplication of non-zero elements in one row of the sparse matrix X is assigned to the same PE, while multiplication of different rows are assigned to different PEs in a round-robin fashion. This way, non-zero elements from each row are processed sequentially on the same PE and do not require the same accumulator simultaneously. However, different rows of a graph adjacent matrix could have extremely different densities (with relative difference $> 100\times$). If we naively tile the workload further into row blocks, it would be inefficient for the majority of PEs to finish execution and remain idle to wait for a single PE to finish processing a particularly dense row, shown as the assignment step in Fig. 5.3. To increase PE efficiency, we design the PEs to work independently, each PE starts to compute a new row immediately when it finishes the previous one. This way, multiple rows are effectively concatenated before assigned to one PE, this way we eliminate idle time (shown as concatenation step in Fig. 5.3). Since it is unlikely for the density of a row to correlate with its row number, by Law of Large Numbers we expect the sum of densities of rows assigned to each PE to be similar. In section 5.5, we will analyze examples in details and compare the computation cost and idle time before and after the concatenation step. Finally, to ensure all PEs balance to process the same amount of elements, including zeros and non-zeros, we inject empty elements at the end of each concatenated row when necessary.

5.3.2.2 Data collision resolution

Due to constraints of on-chip memory, multiple rows of W are stored in the same memory slice, out of which only a single row may be accessed at any time. However, the sparsity of X may cause two PEs to simultaneously access different depths on the same memory slice,

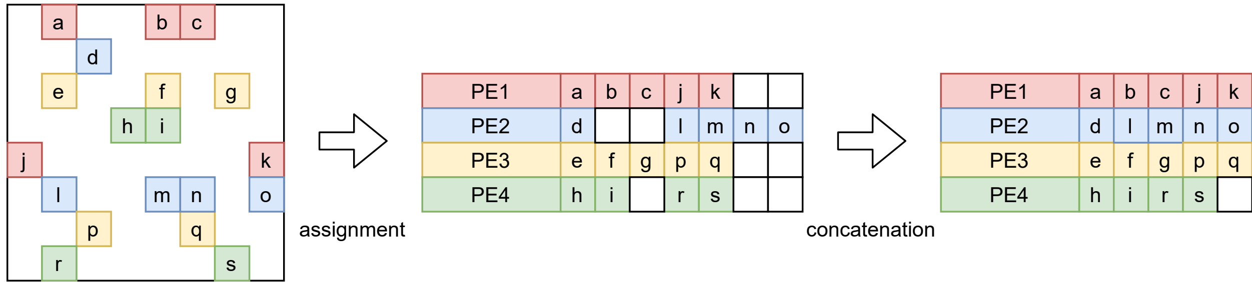


Figure 5.3: Round-robin assignment of non-zero elements to four PEs

which incurs data collision. To resolve this problem, we first develop a multi-bank memory system with data replication to reduce the occurrence of such data collisions (see details in Section 5.4.2). We then inject an empty element with VLD = 0 to prevent any data collision not resolved by the multi-port memory system. For example, there will be a bank conflict if $N+1$ elements are requiring to access the same $N-$port memory. For this case, we will insert an empty element in the place of the $N+1$th element and make the empty element to access at one of the other N addresses so that the bank conflict can be avoided. The inserted element will incur extra inference latency while more ports on a single memory will incur large usage of on-chip memories. The trade-off is then made between the usage of on-chip memory and extra latency incurred by empty elements, detailed analysis will be discussed in Section 5.5.3.

Preprocessing is summarized in two steps in Algorithms 2 and 3. Overall, this preprocessing algorithm is bounded by linear time and space complexity to the total number of non-zero elements in every unique sparse tile. On the other hand, the dense tile is quantized to SINT4 and passed to hardware without structural change. Finally, the preprocessor generates instructions to serialize the execution across layers and steps.

Algorithm 3 Collision stalling

```
1: inputs: tile ∈ ℝ^{N×K}, depth d=16
2: used, row ← [0 for 0:K], [-1 for 0:K]
3: result, share, block, j ← [], {}, {}, 0
4: while sum(used) < N × T do
5:     i ← used_j
6:     if tile_{i,j} ∈ share or tile_{i,j} % d ∉ block then
7:         row_j ← tile_{i,j}
8:         share.append(tile_{i,j})
9:         block.append(tile_{i,j} % d)
10:        used_j ← used_j + 1
11:    else
12:        row_j ← 0
13:    end if
14:    if min(row) ≠ -1 then
15:        result.append(row)
16:        row, share, block ← [-1 for 0:K], {}, {}
17:    end if
18:    j ← (j + 1) % K
19: end while
20: fill zeros until result is rectangular
21: return result
```

5.4 Micro-architecture of LW-GCN

As shown in Fig. 5.4, the micro-architecture of **LW-GCN** is composed of *Peripheral Interface, External Memory Interface, Top Control, PE Array for Sparse-Dense Matrix Multiplication* and on-chip buffers. The *Top Control* module fetches and decodes instructions,

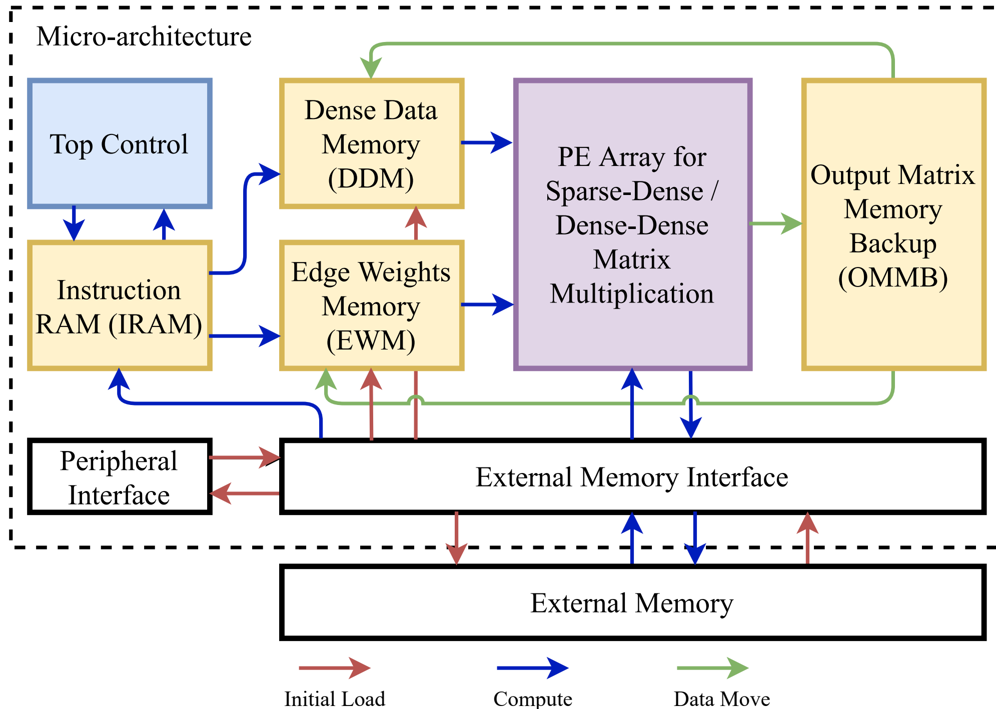


Figure 5.4: The overall micro-architecture and workflow of **LW-GCN**

before passing them to individual modules. As mentioned above in Section 5.3, the micro-architecture processes a single tile at a time.

5.4.1 Overall Workflow

The overall workflow of computing one tile of **LW-GCN** is shown in Fig. 5.4. The dense input data is transferred from external memory to *dense data memory (DDM)* during the *initial load* step. During the *Compute* step, sparse input data is streamed onto the *edge weights memory (EWM)*, and the *PE array* fetches X data from *EWM* and W data from *DDM* and performs multiply-accumulate (MAC) operations in parallel. Upon finishing all pairs of tiles from each *aggregation* or *combination* step, we move output to the *output matrix memory backup (OMMB)*, and move a copy to *DDM* after *combination* and *EWM* after *aggregation* during the *data move* step.

5.4.2 Multi-Bank Dense Data Memory (DDM)

As mentioned in Section 5.3, in order to compute different rows on different PEs in parallel, multiple non-zero elements from the sparse input are streamed on-chip during SpMM. Due to the sparseness and irregularity of X, it is difficult to predict the column positions of the non-zero elements ahead of time. Particularly, it is possible that several PEs require different addresses from the same *DDM*. Limited by read capability of on-chip memory (dual-port RAM only supports reading from two ports at most but the PE number is likely larger than two), such access restriction leads to data collision. In the micro-architecture of **LW-GCN**, we build a multi-port memory to store weights of one tile through *data replication* and *row grouping* to reduce such data collision. In addition, we further reduce the occurrence of such data collision during preprocessing, as mentioned in Section 5.3.

5.4.2.1 Data replication

We replicate the dense data into r replicas for different memory slices. Ideally, when setting r equals to the number of PEs, the aforementioned data collision can be avoided because we would have a dedicated replica of dense data for each PE. However, this incurs a large on-chip memory requirement and is unfeasible in reality. Therefore, we set a relative small r to solve part of the data collision with acceptable resource utilization (the chosen of r is explained in detail in Section 5.5.2), and we introduce *row grouping* to further reduce the occurrence of data collision.

5.4.2.2 Row grouping

We partition each dense data replica into g row groups, each of which is stored independently. Specifically, we store row W_j on group ($j \bmod g$), so that data collision can only occur between elements X_{i_a,j_a} and X_{i_b,j_b} if $(j_a \bmod g) = (j_b \bmod g)$ and $j_a \neq j_b$, which is significantly less likely compared with the undivided memory. Despite the fact that on-chip

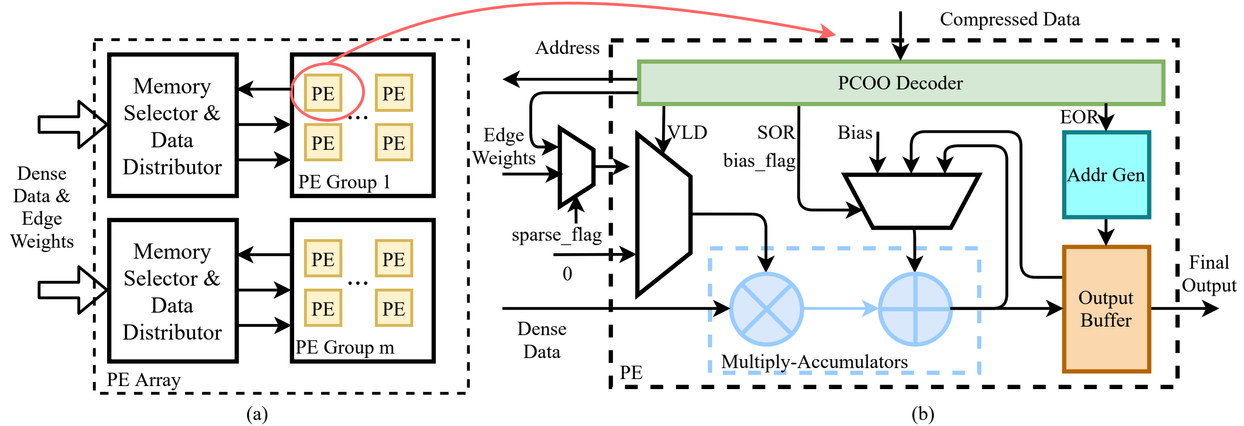


Figure 5.5: (a) The architecture of PE Array; (b) Detailed architecture of a PE.

memory requires a minimum depth to be fully utilized, we are able to use high numbers of row groups to statistically reduce the probability of data collision. However, *row grouping* with large g leads to high complexity for data distribution to PEs, which results in complex placement and routing and increases resource consumption.

Both *data replication* and *row grouping* can efficiently reduce data collision. The remaining collision is avoided by injecting empty elements and processing them as idle cycles, as mentioned in Section 5.3. We experiment with different r and g in section 5.5.3, where we analyze the number of inserted idle cycles versus hardware resource consumption to determine the optimal number of memory replicas and row groups.

5.4.3 Unified PE architecture for MM and SpMM

As shown in Fig. 5.5(a), the number of PE groups and memory banks are kept the same, so that each PE group can access the corresponding memory bank for dense data to avoid data collision. Based on addresses generated by an individual PE, *Memory Selector* and *Data Distributor* dispatch appropriate dense data. We use priority decoder when distributing addresses to memory banks, which allows different PEs to fetch from the same address of the same memory bank.

Note that data replication only applies to dense input but not sparse input. The compressed sparse data is streamed directly to each PE. As shown in Fig. 5.5(b), data first passes through *PCOO Decoder*, where the $log_2(T)$-bit column index is interpreted as memory address to fetch dense data. If a valid bit is observed (VLD = 1), the PE routes the corresponding value to its multiplier, otherwise it assumes the current value is an injected empty element (i.e. data collision, waiting for other PEs to finish, etc.), and routes 0 to the multiplier instead. Since multiple rows are concatenated to feed into each PE, we use SOR and EOR to indicate the start and end of a row, respectively. For each computation step, SOR controls the input of the accumulator to be either its previous result (SOR = 0) or the intermediate result of the previous tile saved in OMMB (SOR = 1). Meanwhile, EOR controls the address generation for storing current results into the output buffer, and also increments the internally tracked row number (EOR = 1).

The MM is also performed in the PE with the same working flow. Since the left matrix is dense, all the rows share the same row and column information, which also goes through the PCOO decoder. The sparse_flag signal then indicate which data to select. When we are processing MM (sparse_flag is 0), we will select the edge weights stored in EWM, otherwise, we will select the value decoded from *PCOO Decoder*. In this way, we can perform both MM and SpMM in the unified PE, which increases the working efficiency of PE for computing *combination* and *aggregation* of GCNs.

5.5 Evaluation

In this section, we evaluate **LW-GCN** on different configurations to identify the impact of each hardware resource. We then compare a final implementation against existing computing platforms on three popular datasets: Cora, CiteSeer, and PubMed. The dimensions and densities of each dataset are shown in Table 5.1.

5.5.1 Experiment Design

We evaluate **LW-GCN** on a two-layer GCN which uses a hidden size of 16 and trained dense weights and biases via the state-of-art framework Pytorch Geometric (PyG). Note that this setup is identical to the GCN used in [GLS19] which we will be evaluating against. In addition, to demonstrate the flexibility of our approach, we extend our evaluation to GraphSAGE under the same datasets.

LW-GCN is implemented in Verilog HDL and deployed onto a Xilinx Kintex-7 K325T FPGA, where we measure the execution time and energy consumption. The DDM is implemented with LUT RAM while other on-chip memories are implemented with Block RAM (BRAM). This is because memory banks in DDM require small depth and high bandwidth, and LUT RAM is more suitable than BRAM. In this section, we first explore the impact of tile size and dense input replication on execution latency. Then, we present a breakdown of latency in individual step of loading, computation, and data movement. Finally, we present an overall performance comparison against existing platforms in terms of latency and energy efficiency.

The preprocessing time for all the datasets evaluated are shown in Table 5.4. For reference we also provide the time it takes to read corresponding data from csv files. We can see that the preprocessing time is comparable with the data loading time. Moreover, we only run preprocess once for each dataset, and the preprocessing time is acceptable.

5.5.2 Hyper Parameter Impact

During each SpMM step, the dense input of one tile is stored in on-chip LUT RAM, where multiple rows would be stored on the same slice of memory in order to fully utilize it. The limitation where only a single row can be read from each LUT RAM slice at a time induces data collision when multiple reads are needed for a same RAM slice and at a same time. As explained in Section 5.4.2, both *data replication* and *row grouping* can effectively reduce data

Table 5.4: Preprocessing time for different datasets.

Dataset	CSV Load Time	Preprocess Time
Cora Features	95.9 ms	19.8 ms
Cora Edges	20.3 ms	8.67 ms
CiteSeer Features	177 ms	46.6 ms
CiteSeer Edges	25.4 ms	11.0 ms
PubMed Features	1.53 s	279 ms
PubMed Edges	167 ms	228 ms

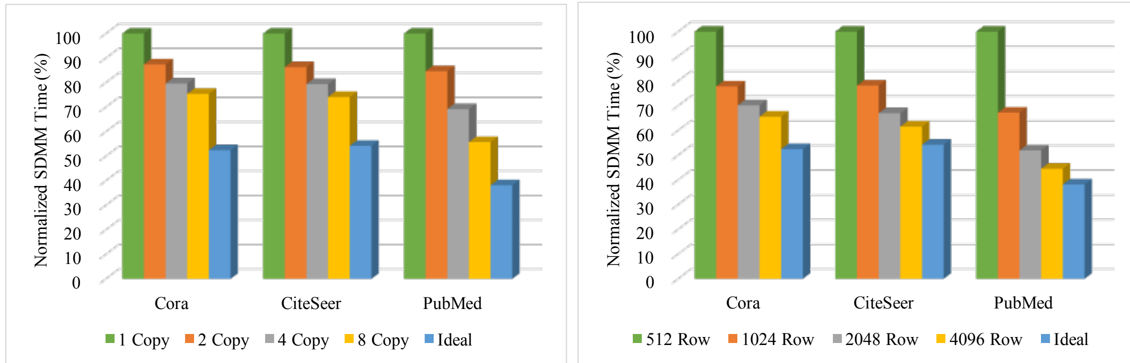


Figure 5.6: Impact of (a) dense data replication with 512-row tiles and (b) tile size with 1 replica

collision. The less data collision will in return results in smaller latency of computing SpMM. On the other hand, due to the irregular nature of graph adjacency matrices, individual rows have very different sparsity which results in PE imbalance, we statistically minimize this effect by utilizing larger tiles. As GCN has the hidden size of 16, we set each PE to have 16 multiply-accumulators and have the fixed relationship between tile size T and row grouping g that $T = 16g$. Therefore, we evaluate the impact of latency from dense data replication r and tile size T, as shown in Fig. 5.6. We can see that the latency of computing is decreased by more dense data replications as well as larger tile sizes. At 8 replicas, **LW-GCN**'s SpMM

latency is reduced by up to 44.23% (on PubMed) compared to 1 replica under the same 512-row tile setup. At 4096-row tiles, SpMM latency is reduced by up to 61.83% (on PubMed) with the same replication setup. The ideal cases in Fig. 5.6 is estimated by summing up the total amount of workload, and assuming every PE is fully utilized.

Due to resource limitations, it is unfeasible to continuously expand tile sizes and replication numbers. Considering the tile size T and data replication r, the number of LUT RAM needed can be expressed in Equ. (5.3).

$$\#LUTRAM = \frac{T \times 16 \times r}{16} = T \times r, \tag{5.3}$$

where the 16 in numerator indicates the data width. Equ. (5.3) is divided by 16 because each LUT RAM can store 16-bit of data [Xil15]. Since we insert two registers in each LUT RAM in order to achieve higher working frequency, the number of flip-flop (FF) can be expressed in Equ. (5.4).

$$\#FF = 2 \times \#LUTRAM. \tag{5.4}$$

Since the dense data are distributed to each PE in one PE group, we need multi-bit multiplexer to select the data of the appropriate memory. Moreover, in Xilinx FPGA, each 8-bit multiplexer is implemented with 1 F7 MUX and 2 LUTs [Cha14]. In this way, the number of F7 MUX and LUT required is listed as follows, respectively.

$$\#F7\ MUX = \frac{T}{16} \times g \times \frac{256}{8} = 2 \times T \times g. \tag{5.5}$$

$$\#LUT = 2 \times \#F7\ MUX. \tag{5.6}$$

In Equ. (5.5), $\frac{T}{16} \times g$ indicates the number of multi-bit multiplexers. It is divided by 16 because 16 multiply-accumulators in each PE can share a same multi-bit multiplexer. Since we using 256-bit multiplexer (each PE has 16 multiply-accumulator and each data is

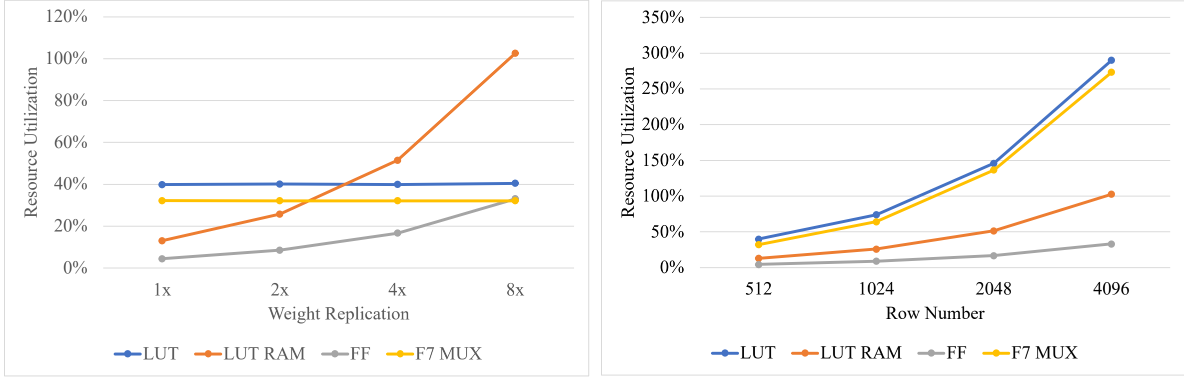


Figure 5.7: Resource consumption of (a) replication and (b) tile size

Table 5.5: Resource utilization on Kintex-7 325T FPGA.

Resource	LUT	LUT RAM	FF	F7 MUX	BRAM	DSP
Used	161529	33804	94369	32768	291.5	512
Available	203800	64000	407600	101900	445	840
Utilization (%)	79.26	52.82	23.15	32.16	65.51	60.95

in SINT16 data format), the number of multi-bit multiplexers is multiplied by 256/8 to get the number of F7 MUX.

We also evaluate the resource utilization under the resource limited device with respect to dense data replication r and tile size T, as shown in Fig. 5.7. According to the results in Fig. 5.7, the resource utilization for storing and fetching dense input data follows our analysis in Equ. (5.3) - (5.6). Moreover, $r = 4$ and $T = 512 (g = 32)$ achieve the best balance between resource and performance under the specific FPGA, and will be used for the remaining of experiments. When the FPGA platform varies, it is also easy to change the hyper parameters following Equ. (5.3) - (5.6) with the resource constraints. Given these hyper parameters, the overall resource utilization on Kintex-7 K325T FPGA is shown in Table 5.5.

5.5.3 Latency Breakdown

During preprocessing, we inject empty elements (see section 5.3.2) to handle the corner case where a row X_i contains no non-zero elements, or when a PE completes its execution. This enables each PE to internally track current row i, which allows us to remove row number i from off-chip memory and reduce memory bandwidth consumption. We also inject empty element symbols when two elements are to read from different depths of the same memory, in order to prevent data collision. Fig. 5.8 shows the latency breakdown for overall runtime (including MM/SpMM, memory load and on-chip data movement) as well as for SpMM (including computation, PE imbalance, and data collision). The latency of MM is dominant by computation as MM does not have the issues of PE imbalance and data collision. Therefore, we do not list the latency breakdown for MM. In both cases, the time spending on computation (*i.e.* MM/SpMM for overall, and computation for SpMM) is dominant. The dataset PubMed has a relatively larger PE imbalance. This is because the higher sparsity and irregularity of PubMed (the edge density in PubMed is about 1/5 of that in Cora and CiteSeer.) When using the round-robin workload assignment and scheduling scheme, there exists the number of non-zeros elements in one row is higher than the sum of non-zero elements in other rows, thus causing an imbalance. The higher PE imbalance in PubMed also indicates potential room for improvement by workload assignment and scheduling, which will be explored in the future. For example, we can assign non-zero elements of the same row to multiple PEs to make PE balance. At the same time, we need extra buffers and adders on hardware to make correct computation.

We further evaluate the specific utilization rates per PE with respect to *combination* and *aggregation* operations. For simplicity we only show the first layer on Cora dataset in Fig. 5.9. It shows that the idle time of each PE varies from 6% to 12% for *combination* and from 1% to 20% for *aggregation*, respectively. In overall, the lowest utilized PE is idle for less than 20% of the SpMM time.

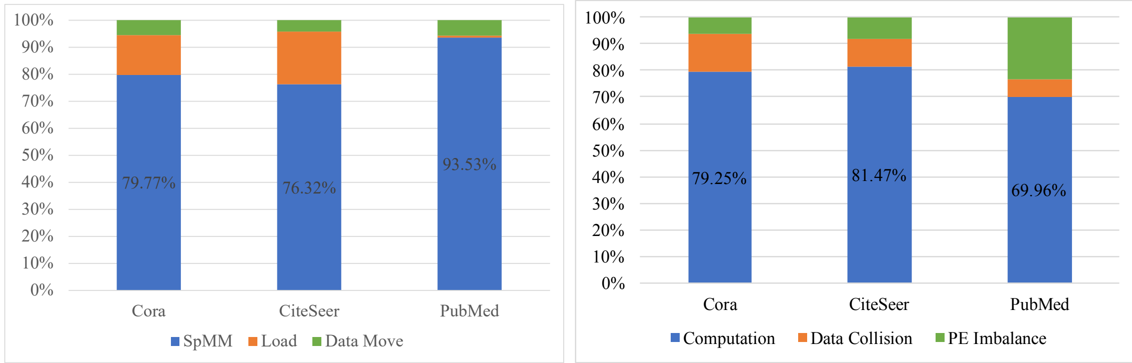


Figure 5.8: Latency breakdown for (a) full execution and (b) SpMM

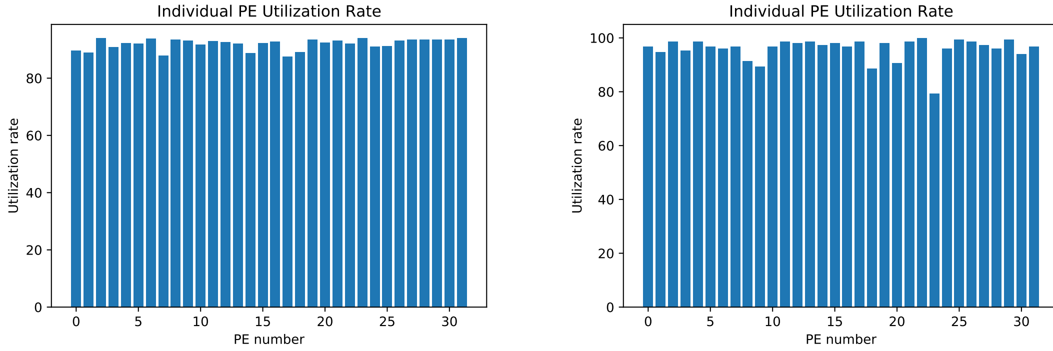


Figure 5.9: PE utilization during SpMM for Cora: (a) first combination tile and (b) first aggregation tile

5.5.4 Overall Comparison

We evaluate the overall latency and energy efficiency of **LW-GCN** against the Intel Xeon Gold 5218 CPU, NVIDIA Xavier NX edge GPU with Volta architecture, NVIDIA RTX3090 GPU with Ampere architecture and state-of-the-art FPGA-based GCN accelerator AWB-GCN [GLS19], as shown in the top half (GCN) of Table 5.6. The overall latency on CPU and GPU are evaluated under the state-of-the-art software platform Pytorch-Geometric. Note that AWB-GCN is implemented on Intel Stratix 10 D5005 with frequency of 330 MHz and uses 8192 DSP slices. We normalize their reported latency and energy efficiency with this resource utilization to our FPGA (200 MHz and 512 DSP slices) for a fair comparison.

Table 5.6: Comparison with CPU, edge GPU, general GPU and existing FPGA accelerator on GCN and GraphSAGE

Platform (Clock rate: GHz)	Latency (ms) [speedup]			Energy efficiency (graph/kJ)		
	Cora	CiteSeer	PubMed	Cora	CiteSeer	PubMed
	GCN					
Intel Xeon Gold 5218 (2.1)	1.89 [1×]	3.88 [1×]	12.5 [1×]	4.23E3	2.06E3	640
NVIDIA Xavier NX (1.1)	1.87 [1×]	1.88 [2.1×]	2.01 [6.2×]	3.57E4	3.55E4	3.32E4
NVIDIA RTX3090 (1.7)	0.492 [3.9×]	0.481 [8.1×]	0.491 [26×]	5.83E3	5.95E3	5.83E3
AWB-GCN (0.2)	0.0613 [31×]	0.115 [35×]	0.791 [16×]	7.70E5	4.82E5	6.21E5
LW-GCN (0.2)	**0.0412 [46×]**	**0.0652 [60×]**	**0.571 [22×]**	**2.98E6**	**1.88E6**	**2.14E5**
	GraphSAGE					
Intel Xeon Gold 5218 (2.1)	172 [1×]	385 [1×]	340 [1×]	46.5	20.8	23.5
NVIDIA Xavier NX (1.1)	10.6 [16.3×]	9.63 [40.0×]	10.8 [31.5×]	6.28E3	6.92E3	6.17E3
NVIDIA RTX3090 (1.7)	1.94 [89.0×]	1.88 [204.8×]	1.96 [173.6×]	1.47E3	1.52E3	1.46E3
AWB-GCN (0.2)	NA	NA	NA	NA	NA	NA
LW-GCN (0.2)	**0.086 [2.01E3×]**	**0.14 [2.75E3×]**	**1.07 [318×]**	**1.42E6**	**8.77E5**	**1.72E4**

For energy efficiency, Intel Stratix 10 D5005 uses 14nm transistors while Xilinx Kintex-7 325T uses 28nm transistors, following the analysis in [LLH05], we normalize their power consumption by $(\frac{28}{14})^2 = 4\times$. For GCN as illustrated in Table 5.6, **LW-GCN** outperforms all the other platforms in terms of latency and energy efficiency. Specifically, **LW-GCN** achieves up to 60×, 32×, 12× and 1.7× speedup, as well as 2478×, 84×, 511×, and 3.88× energy efficiency, compared with CPU, edge GPU, GPU, and AWB-GCN, respectively. **LW-GCN** is able to achieve such performance benchmarks while keeping a small resource budget, due to the techniques used in software preprocessing and micro-architecture to reduce data collision and PE imbalance for SpMM, as well as performing MM and SpMM with unified architecture.

5.5.5 Extending LW-GCN to Other Algorithms

Although **LW-GCN** is designed as a GCN accelerator, the underlying MM/SpMM acceleration is not limited to GCN, and can be applied to any MM/SpMM related GNN workloads. In fact, due to the sparse nature of graph adjacent matrices and dense nature of weight matrices, most GNNs workload involves MM/SpMM. As a proof of concept we directly applied **LW-GCN** to GraphSAGE [HYL17c] on the same datasets, and achieved an acceleration of up to 2750×, 123×, and 22.6× and energy savings of up to 42200×, 226×, and 966× over CPU, edge GPU, and GPU respectively, as shown in the bottom half of Table 5.6. Note that AWB-GCN results for GraphSAGE is not available in the literature. Additionally, note that the PyG implementation for GraphSAGE involves a sparse-sparse matrix multiplication due to computing *aggregation* before *concatenation*, when applied on the three datasets we used, therefore the latency is much higher than it could be.

We also evaluate the inference latency of larger datasets (*i.e.*, Reddit [HYL17a]) and deeper GCN architectures (*i.e.*, GraphSAINT [ZZS19a]) running on **LW-GCN** to validate the flexibility. First, we run GCN with the larger dataset Reddit, which has 232965 nodes and 602 features. We partition Reddit into small tiles that fits for **LW-GCN** and run all the tiles iteratively to get the inference results. The total inference time for running Reddit on **LW-GCN** is 1249.6ms. For reference, the original reported inference time on AWB-GCN [GLS19] is 31.81 ms, which would scale to 839 ms given our frequency and dsp usage. The performance drop is because **LW-GCN** is only capable of storing data of one tile, and we have to do data transfer between different tiles. In fact, 54.5% of inference time is spent on data loads and stores, where a typical value is 6% - 23%, as shown in Fig. 5.8 (a). Second, we run Cora dataset with GraphSAINT architecture, which has 6 graph convolutional layers. The inference latency is 0.225ms, of which 21.5% is used for data communication while others for computation. This is quite similar to that of running GCN because all the layers in GraphSAINT share the similar computation operations as GCN. The above two examples show that the proposed **LW-GCN** can also work on larger

datasets and deeper GNNs. For larger datasets, **LW-GCN** does not get promising results because of the resource limitation and off-chip memory bandwidth.

5.6 Conclusions and Future Work

GCN involves heavy computation of multiplications of sparse and dense matrices, but most neural network accelerators are targeted at CNN with dense matrix multiplication and therefore are not efficient for GCN. Recently, FPGA-based **AWB-GCN** improves performance, but still requires a large amount of on-chip memory. Therefore, it is inapplicable to resource limited hardware platforms such as edge devices.

In this paper, we have proposed **LW-GCN**, a software-hardware co-designed accelerator for GCN inference. **LW-GCN** consists of a software preprocessing algorithm and an FPGA-based hardware accelerator. The core to **LW-GCN** is our SpMM design, which reduces memory needs through tiling, data quantization, sparse matrix compression, and workload assignment with data collision resolution. Experiments show that for GCN, **LW-GCN** reduces latency by up to 60×, 12×, and 1.7× compared to CPU, GPU, and AWB-GCN and increases power efficiency by up to 912×, 511×, and 3.87×. Additionally, the underlying SpMM design used by **LW-GCN** is applicable to other graph neural network algorithms such as GraphSAGE, not limited to GCN.

CHAPTER 6

SkeletonGCN: A Simple Yet Effective Accelerator for GCN Training

6.1 Introduction

Over recent years, graph convolutional networks (GCNs) [KW16b] and similar representation learning models have grown to be an efficient family of models to process non-euclidean graph data. The graph convolution layer is able to improve learning through information aggregation from each node's neighbors, and have shown strong results among popular graph datasets. There have been numerous real-world applications of GCN at the time of writing across various popular research areas, from computational drug development to web-scale recommenders. The capability to make use of inter-sample data is likely to pick up more popularity in the near future [SZG20b, YHC18b].

However, despite its great success on the prediction accuracy front, training of GCN comes with an enormous memory and computation burden. GCN is often used to process large graph datasets, where the number of nodes can reach hundreds of thousands [HYL17b]. Therefore, the datasets can be as large as gigabytes and have trouble fitting onto hardware accelerators such as GPUs and FPGAs. Fortunately, there have been sampling-based approaches [CLS19b, ZZS19b] to decompose large graphs into many smaller subgraphs, each of which can then fit onto the hardware with ease. Specifically, the experiments in [CLS19b, ZZS19b] demonstrate that it is possible to decompose many popular datasets into subgraphs with negligible reduction in training accuracy. On the other hand, both mem-

ory consumption and computation efficiency can be further reduced through quantization. Previously, Degree-Quant [TFL20b] is able to quantize the data to INT8 with little accuracy loss during the inference phase. However, there is yet to be an accelerator to fully leverage the advantages of the approaches above.

Previous dedicated hardware accelerators of CNNs over such data volume, such as Cambricon, TPU and OPUs [LDT16c, JYP17c, YWZ20, YZW20c], explore parallel computing of large number of compute cores to improve efficiency. Such efficiencies depend heavily on data regularity and re-usage. However, it becomes challenging to apply such techniques to accelerate the training of GCNs [KW16b, CLS19b, ZZS19b] due to the sparsity in graph adjacency matrices. Firstly, the sparsity in adjacency matrices incurs a sparse matrix-matrix multiplication (SpMM), which results in a high volume of irregular memory access. Secondly, the sparsity also incurs imbalance in load and computation when designing multiple processing elements (PEs) to perform parallel computing for acceleration. Moreover, the commonly used graph convolution layers in GCNs also incur a dense matrix-matrix multiplication (MM), which requires heavy arithmetic resources for fast computation.

There exist GCN accelerators to overcome the above challenges during the inference phrase. For example, AWB-GCN [GLS20] processes each SpMM and MM during inference in pipelined modules, and uses a dynamic resource allocation scheme to keep each module actively utilized. Meanwhile, EnGN [LWL20b] proposes a uniformed architecture to accelerate SpMM and MM. However, it is difficult to expand these approaches to training acceleration as their designs either cache large amounts of data on-chip or depend on high bandwidth of external memory. In the training phase, most intermediate results need to be stored for back-propagation, and therefore require much larger memory capacities. GraphACT [ZP20] proposes a CPU-FPGA heterogeneous platform to accelerate GCN training. They first focus on redundancy reduction in the algorithm level and then design an accelerator to parallel the computation of feature propagation and weight transformation. However, their redundancy reduction scheme relies heavily on the properties of the datasets and they treat the feature

propagation and weight transformation as separate modules. The stray away from a uniform architecture may leads to low hardware efficiency as data dimensions change.

To this end, we propose **SkeletonGCN**, a simple yet effective FPGA-based accelerator for GCN training with both algorithm and hardware optimizations. We first apply quantization to a sample-based GCN training algorithm to reduce storage requirement. In addition, we simplify the non-linear operations (*i.e.,* L2 normalization, Adam optimizer) to fit better for FPGA acceleration and eliminate redundant computations by identifying reusable intermediate results. Thereafter, in order to further reduce storage and bandwidth consumption, we employ a compact packet-level column-only coordinate-list (CPCOO) format proposed in LW-GCN [TWL21] to compress the sparse data. A unified PE architecture is then developed to efficiently handle SpMM, MM and MM with transpose. The architecture is fully pipelined and equipped with ping-pong buffers to maximize DSP efficiency. We evaluate our design on a Xilinx Alveo U200 board. The experimental results show that our simplification steps incur negligible accuracy loss. Moreover, compared with the state-of-the-art FPGA accelerator [ZP20] on the same experiment settings, we can achieve up to 11.3× speedup on the total training convergence time. Compared with state-of-the-art CPU and GPU, **SkeletonGCN** can achieved up to 178× and 13.1× speedup, respectively.

To summarize, our main contributions are as follows:

- **Simple yet effective training:** We apply low precision quantization on existing training approaches for the commonly used large datasets. In addition, we simplify the computation of non-linear operations and eliminate redundant computation during the training process. Experimental results show that **SkeletonGCN** offers comparable training accuracy despite these proposed simplification techniques.

- **Unified high efficiency PE architecture:** It can support SpMM, MM and MM with transpose with high DSP efficiency. Comprehensive experiments show that we can achieve up to 95% DSP efficiency, which in turn contributes to overall low training

latency.

- **Low training latency:** Compared with a prior FPGA-based accelerator [ZP20] under the same experiment settings, **SkeletonGCN** achieves up to 11.3× speedup for total training convergence time. The speedup is up to 178× and 13.1× respectively compared with state-of-the-art CPU and GPU.

6.2 Background and Related Work

6.2.1 Workload Breakdown

Following the state-of-the-art architectures, GCN [KW16b] and GraphSAINT [ZZS19b] are composed of multiple graph convolution layers or multi-layer perceptron (MLP) layers. The training process of a network can be divided into forward propagation, backward propagation, and the weight update phase. To avoid ambiguity, the mathematical definitions we used in GCN and GraphSAINT throughout this work are listed as follows. Forward propagation in GCN for layer l is defined in Equ. (6.1).

$$X_l = \mathrm{ReLU}(AX_{l-1}W_l), \tag{6.1}$$

where X_{l-1} and X_l denote the feature matrix of layer $l-1$ and l, respectively. W_l is the weight matrix of layer l while A is the adjacency matrix of the input graph. Each graph convolution layer applies a ReLU activation function. Since the adjacency matrix is always sparse for different graph datasets, while the feature and weight matrices are typically dense, the basic operations for the forward phase are sparse-dense matrix-matrix multiplication (SpMM) and dense matrix-matrix multiplication (MM). The forward phase in GraphSAINT shares the same basic operations as that in GCN, but arranges them in different combinations. Instead of graph convolution layers, GraphSAINT introduces a concept of **order**. Each layer has an order between 0 and 1, and an order 0 layer is simply an MLP layer, while an order 1 layer

is shown as

$$X_l = \text{ReLU}(\begin{bmatrix} X_{l-1}W_{l,a} & AX_{l-1}W_{l,b} \end{bmatrix}), \tag{6.2}$$

where the [.] indicates column-wise concatenation of the two intermediate results, and $W_{l,a}, W_{l,b}$ are the self-weight and neighbor-weight matrices of layer l, respectively. The MLP layer performs a simple MM as $X_L = \sigma(X_{L-1}W_L)$. While orders above 1 were defined in the original work, it has not been used in practice. GraphSAINT also inserts an L2 normalization before the final MLP layer, as shown in Equ. (6.3).

$$X_l = \frac{X_{l-1}}{||X_{l-1}||_2}. \tag{6.3}$$

For the training process of GCN and GraphSAINT, gradients are computed during backward propagation before weights are updated via the Adam optimizer [KB14]. The gradients of layer $l-1$ for GCN are shown in Equ. (6.4) and (6.5).

$$\frac{\partial \mathcal{L}}{\partial X_{l-1}} = \mathbb{1}_{X_{l-1}>0}[W_{l-1}^T A^T \frac{\partial \mathcal{L}}{\partial X_l}], \tag{6.4}$$

$$\frac{\partial \mathcal{L}}{\partial W_{l-1}} = A^T \frac{\partial \mathcal{L}}{\partial X_l} X_{l-1}^T, \tag{6.5}$$

where the superscript T denotes matrix transpose. The basic operations during the backward phase are also SpMM and MM. However, the input to an MM may be a transposed copy of a previous feature map or weight, which requires a different data access pattern, we refer to this case as "MM with transpose". The gradients of layer $l-1$ for GraphSAINT are computed as follows:

$$\begin{bmatrix} \frac{\partial \mathcal{L}}{\partial X_{l,a}} & \frac{\partial \mathcal{L}}{\partial X_{l,b}} \end{bmatrix} = \frac{\partial \mathcal{L}}{\partial X_l}, \tag{6.6}$$

$$\frac{\partial \mathcal{L}}{\partial X_{l-1}} = \mathbb{1}_{X_{l-1}>0}[W_{l-1,a}^T \frac{\partial \mathcal{L}}{\partial X_{l,a}} + W_{l-1,b}^T A^T \frac{\partial \mathcal{L}}{\partial X_{l,b}}] \tag{6.7}$$

$$\frac{\partial \mathcal{L}}{\partial W_{l-1,a}} = \frac{\partial \mathcal{L}}{\partial X_{l,a}} X_{l-1,a}^T \tag{6.8}$$

$$\frac{\partial \mathcal{L}}{\partial W_{l-1,b}} = A^T \frac{\partial \mathcal{L}}{\partial X_{l,b}} X_{l-1,b}^T \tag{6.9}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial X_{l-1}} &= \frac{\frac{\partial \mathcal{L}}{\partial X_{l-1}}}{||X_{l-1}||_2} - X_{l-1} \frac{\sum (X_{l-1} \times \frac{\partial \mathcal{L}}{\partial X_{l-1}})}{||X_{l-1}||_2^3} \\ &= \frac{\frac{\partial \mathcal{L}}{\partial X_{l-1}}}{||X_{l-1}||_2} - \frac{X_{l-1}}{||X_{l-1}||_2} \sum (\frac{X_{l-1}}{||X_{l-1}||_2} \times \frac{\frac{\partial \mathcal{L}}{\partial X_{l-1}}}{||X_{l-1}||_2}) \end{aligned} \tag{6.10}$$

In Equ. (6.4) - (6.10), $\mathcal{L}$ denotes the training loss, specifically categorical cross entropy loss. The [.] on the left side of Equ. (6.6) indicates a column-wise partitioning into two blocks with equal number of columns, opposite of the concatenation step in Equ. (6.2). We show Equ. (6.10) in this format to show that the intermediate result $||X_{l-1}||_2$ can be reused to simplify the computation (see details in Section 6.4.2.3). After computing the gradients, we use the Adam Optimizer [KB14] to update weights for learning.

For all input (sub)graphs in GraphSAINT related computation, we directly use the random-walk sampler as proposed in the original text [ZZS19b]. The number of roots per subgraph is tuned per dataset to generate subgraphs with approximately 2048 nodes in order to fit our hardware design.

6.2.2 Low Precision Training

Various low precision training algorithms have been proposed to train CNNs. The work in [SCC19] uses 8-bit floating point to train CNNs by exploring different bit-width of exponent and mantissa. Research recently uses neural architecture search (NAS) to search the optimal precision combinations for each layer in CNNs [WDZ19c, WLL19c]. They also consider hardware feedback, such as latency and power, to achieve speedup and power reduction on hardware during inference.

However, these studies cannot directly be applied to GCNs training due to the inherent differences between CNNs and GCNs, such as input sparsity. The work in [TFL20b]

proposes an architecture-agnostic method, which is applied to existing quantization-aware training algorithms, to quantize GNN while maintaining accuracy during inference. On the other hand, SGQuant [FWL20b] proposes a GNN-tailored quantization algorithm to reduce GNN memory consumption. Moreover, they use a fine-tuning scheme to compensate for the accuracy loss caused by precision reduction. Although the above approaches can reduce the memory requirements for GCNs during inference, the studies on quantization of large graphs and hardware-aware quantization is not sufficient.

6.2.3 FPGA-based Accelerators

FPGA-based accelerators have been explored extensively on CNN inference and training. The work in [YZW20c, WWC21, WZW21] uses the FPGA overlay technique to accelerate CNN inference with different data representations, including 8-bit fixed point, 8-bit floating point and mixed precision. CNN training accelerators are developed in these studies [KMY19, VSY20]. They take a fully pipelined architecture to accelerate mini-batch training on CNNs and use HBM to further resolve the issue of external bandwidth constraints.

Although GCNs share a similar layer-based network architecture as CNNs, accelerating GCNs is quite different as the workload consists of two different major operations, SpMM and MM. AWB-GCN [GLS20] proposes a configurable architecture to fit for SpMM and MM with different sparsity during GCN inference. EnGN [LWL20b] processes SpMM and MM in a unified architecture to speed up the inference phase of GCNs. Although existing accelerators achieve speed boosts on GCN inference, their designs rely heavily on large on-chip buffers or high external memory bandwidth. Caching the same amount of data on chip for GCN training would be impractical, as training requires larger amounts of intermediate results to be stored for back-propagation. The Rubik [CWX20b] develops an ASIC accelerator to cooperate with graph reordering to support both node-level and graph-level computing. GraphACT [ZP20] accelerates GCN training through redundancy reduction in software and parallel computing in hardware. However, the redundancy reduction technol-

ogy requires a uniform weight value for all the edges, which is not applicable to support other GCN architectures and datasets. HP-GNN [LZP21] proposes a framework to generate GNN accelerators on a CPU-FPGA platform automatically. They reduce the memory traffic and random memory access to accelerate the GNN training. However, both GraphACT and HP-GNN treat feature aggregation and weight transformation as separate modules that will reduce DSP runtime utilization.

6.3 Optimized Training

6.3.1 Training Simplifications

In this work, we perform the majority of GCN training with 16-bit signed integers (SINT16). We chose signed integer representation over floating point as arithmetic operations with integers consume less hardware resources and power. The precision of 16 bits was selected as Xilinx provides native IP support for 16-bit integer operations in DSP configuration. While SINT8 is also natively supported, our experiments show that direct quantization to SINT8 would result in a significant accuracy degradation.

We apply quantization to the initial input feature maps and adjacency matrices, and initialize trainable parameters directly as SINT16. Experiments show that for most of computation, quantization yields negligible loss to final training accuracy, and would not affect convergence time in terms of epochs. However, for some non-linear operations, applying quantization would devastate results, and they must be kept in FP32. Specifically, L2 normalization, softmax, and square root cannot be quantized without incurring a major loss of accuracy.

Due to the representation precision of SINT16, multiplication with a number within $[1 \pm 2^{-16}]$ results in no change at all. This allows two simplifications to the Adam weight update procedure. First, we eliminate the need to compute $\hat{m}_t$ and $\hat{v}_t$, as the quantized results are identical to the original m_t and v_t starting from the second to third epoch. Second, the

Table 6.1: Dimensions, Densities and Workloads Across Datasets.

Datasets	Nodes	Edges	Edge Density
PPI	14755	225270	0.1035%
Reddit	232965	11606919	0.0214%
Yelp	716847	6977410	0.0014%

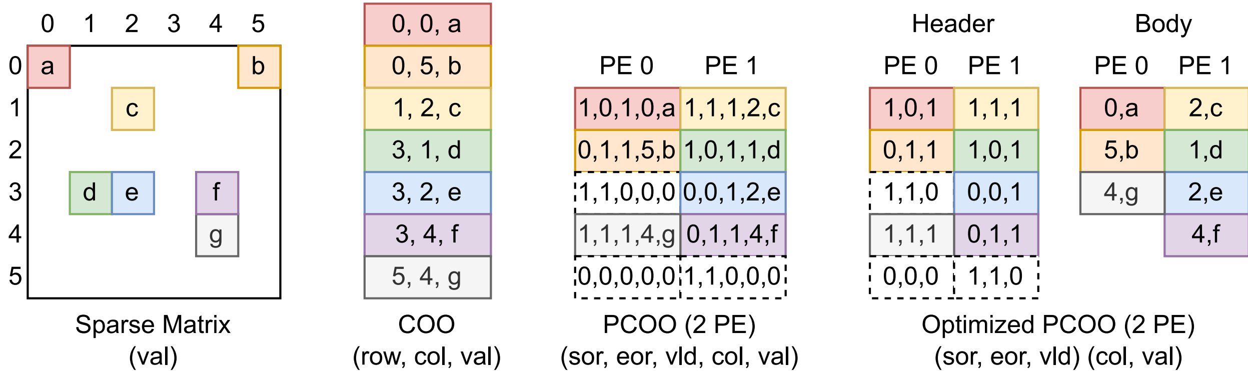


Figure 6.1: Packet-level column-only coordinate list format

learning rate η is rounded to the nearest power of 2 to replace the multiplication by a simple bit shift.

Furthermore, we detect intermediate results used in multiple places and cache them to prevent redundant computation. Specifically, the step $A^T \frac{\partial \mathcal{L}}{\partial X_l}$ was used twice in back propagation, and $||X_{L-1}||_2$ was used multiple times in L2 normalization and its gradient.

6.3.2 Hardware-aware Compression

After exploring the GCN architectures and commonly used datasets, an immediate observation is that the adjacency matrix is often extremely large and sparse, as shown in Table 6.1. Without compression, the adjacency matrices would consume an impractical amount of storage, while less 1% are non-zero elements. Therefore, it is necessary to compress sparse matrices to only store and compute the non-zero elements.

Table 6.2: Compressed matrix sizes across datasets and algorithms (random-walk sampled subgraphs [ZZS19b])

Dataset	Avg. Nodes	Avg. Edges	Avg. COO Size	Avg. PCOO Size	Avg. CPCOO Size
PPI	1992	41939	2.01Mb	4.89Mb	1.76Mb
Reddit	1977	10780	517Kb	1.65Mb	486Kb
Yelp	1959	7561	363Kb	1.98Mb	412Kb

Following the approach proposed in [TWL21], we use their packet-level column-only coordinate-list (PCOO) format to compress sparse matrices. PCOO provides an easy decompression mechanism on hardware and allows the dense matrices to be processed in the same way with negligible extra cost, which enables an uniform PE design. The PCOO format treats all the elements in one row as a packet. As shown in Fig. 6.1: PCOO, three control bits are used to indicate each start-of-row (SOR), end-of-row (EOR) and valid (VLD) non-zero element in each row, respectively. While the column information of each non-zero element is included, the row information can be tracked on corresponding PEs, and therefore can be hidden from the data. Since PCOO injects empty elements for empty rows, it will be less effective for larger and sparser matrices, especially when we are tiling large matrices into small matrices. For large datasets shown in Table 6.1, there are few elements per row. When we tile these matrices by columns, we end up with large numbers of empty rows, which incurs large amounts of storage with empty elements.

To this end, we optimize the PCOO format by dividing it into **header** and **body** parts, as shown in Fig. 6.1: CPCOO. The header field includes the SOR, EOR and VLD signals while the body field includes the column information. The representation of the rows with non-zero elements is the same as PCOO. For empty rows, the header information would contain 0 in the VLD bit, and we do not insert any empty element into the body field. On the hardware during decompression, we will fetch the body according to the VLD signal provided in the header field (see details in Section 6.4.2). Since the data width for the body

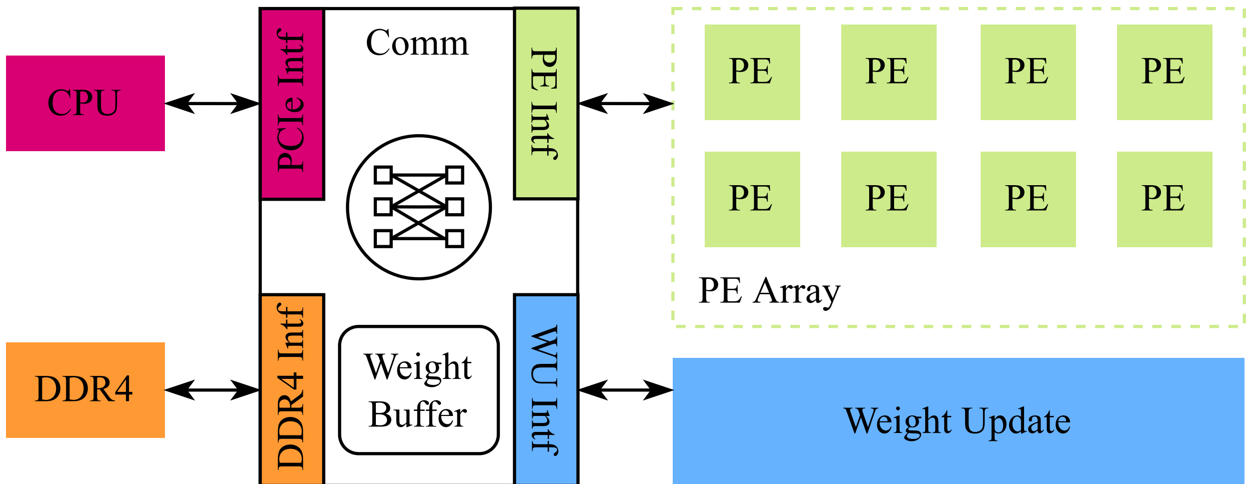


Figure 6.2: Overall architecture of **SkeletonGCN**.

field is much larger than the header field, we can significantly reduce storage consumption for empty rows of large datasets than PCOO. Table 6.2 shows the storage consumption of original and updated PCOO per average subgraph across three datasets. CPCOO reduces memory by 2.87× to 4.81× compared to PCOO.

6.4 Hardware Architecture

In this section, we discuss in detail the hardware architecture of the proposed **SkeletonGCN**, which efficiently supports the training process of quantized GCN.

6.4.1 Overall Architecture

We first analyze the GCN training process. The forward and backward phases are computationally expensive, thus are assigned to the FPGA. The weight update process, which includes large amounts of element-wise operations, is also assigned to FPGA to perform immediately when the gradients are computed during the backward phase. On the other hand, since softmax and categorical cross-entropy loss require exponential and logarithmic func-

tions that are hardware expensive to compute accurately, we assign them to the CPU. Other software processes, such as graph sampling, data pre-processing, are also assigned to the CPU. The detailed scheduling between CPU and FPGA will be discussed in Section 6.4.5.

Regarding workload assignment, the overall architecture of **SkeletonGCN** is shown in Fig. 6.2. The *PE Array Module* performs all the operations in the forward and backward phases, while the *Weight Update Module* is followed to update the weights after back-propagation. The *Communication Module* is responsible for the data transfer between CPU and FPGA, between off-chip memory (DDR4) and FPGA, and also among different PEs.

6.4.2 Unified PE Architecture

In order to improve the performance of the training accelerator, we need to 1) reduce the overhead of off-chip memory access, and 2) increase the efficiency of computation resources. Since the whole graph is sampled and only one subgraph is trained each time, we can set the size of the subgraph appropriate so that all the data of a subgraph can be handled by the on-chip memories of the FPGA. In this way, the FPGA only need to access the off-chip memory three times during the training of one subgraph: 1) to get the initial data for the forward pass, 2) to send back the results of the last layer to compute loss and gradients on CPU, and 3) to get the gradients of the last layer for back-propagation.

To increase computation efficiency, we propose a unified PE architecture to perform each step in each layer during the forward and backward phases, as shown in Fig. 6.3(a). As discussed in Section 6.3, the main operations during forward and backward are SpMM, MM and MM with transpose, where the basic operation is multiply-accumulate. In this way, we first design an $M \times N$ *Multiply-Accumulator (MACC) Array* to support each SpMM, MM and MM with transpose in parallel. Generally, all the MACC units in each row of the *MACC Array* share a same input. The challenge then becomes how to feed data into the MACCs to make it perform efficiently under different workloads.

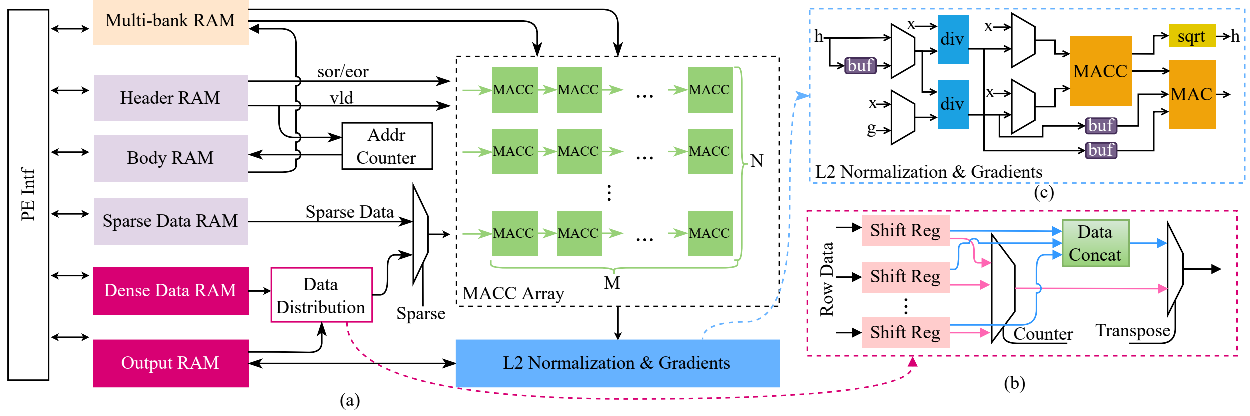


Figure 6.3: (a) The architecture of the unified PE. (b) The architecture of the data distribution module. (c) The architecture of the L2 normalization and gradients module.

6.4.2.1 SpMM

As discussed in Section 6.3, we use the CPCOO format to compress sparse matrices and we store the compressed data (header, body and non-zero elements) into 3 separate RAMs, as shown in Fig 6.3 (a). During computation, the Header (SOR/EOR/VLD) is flushed out from the *Header RAM* and we use "VLD" signal to enable the *Address Counter*, which is used to generate the address of the *Body RAM*. The output of the *Body RAM* is the column position of the non-zero element in the sparse matrix, thus indicating the address of the corresponding dense data in the *Multi-bank RAM*. In this way, the decompression logic of data under the CPCOO format can be as simple as several wires and a counter, as shown in Fig 6.3(a). The non-zero sparse data and corresponding dense data are then fed into the *MACC Array*, and the "SOR" and "EOR" signals control when to start computation for a new row and when to save the results. With the fully pipelined architecture, the MACC units keep active during most cycles of computing SpMM, thus leading to a high DSP efficiency (see details in Section 6.5.4).

6.4.2.2 MM and MM with transpose

Although MM and MM with transpose share the same computation operations, they require different memory access for matrices and transposed matrices. Moreover, the output of one layer can be either used in standard arrangement or the transposed arrangement for subsequent computations. To save memory access burden and improve DSP efficiency, we design a same memory load and store logic for both MM and MM with transpose. In contrast, a *Data Distribution Module* is added to control the data needed by the *MACC Array* for computing MM and MM with transpose, as shown in Fig. 6.3(b). The row data of the left matrix in MM or MM with transpose is first fetched from the on-chip memory and then fed into the shift registers. Each shift register stores one row and can be configured to output one element or all the elements in one row according to computation mode. When computing MM, each shift register is first configured to output one element. Then all the elements from different shift registers are concatenated and fed into the N rows of the *MACC Array*, as shown with the blue arrow in Fig. 6.3(b). On the other hand, when computing MM with transpose, each shift register is first configured to output all the elements in one row. Then the elements are selected one by one to feed into the N rows of the *MACC Array*, as shown with the pink arrow in Fig. 6.3(b). By setting the data width of the on-chip memory $N \times N \times 16$ (we use 16-bit signed integers), we can keep outputting active data every cycle for both MM and MM with transpose to make the MACCs active, thus maintain high DSP efficiency. Since we set $N = 16$ in our accelerator, the data width of $N \times N \times 16$ is easy to achieve by using block RAMs (BRAMs) or ultra RAMs (URAMs) in Xilinx FPGA.

6.4.2.3 L2 normalization and its gradients

The L2 normalization and its gradients both follow a MM in the forward and backward phase. Therefore, we develop an extra module which takes the results of the *MACC Array* as inputs to pipeline the computation. We use the CORDIC IP and division IP in Xilinx

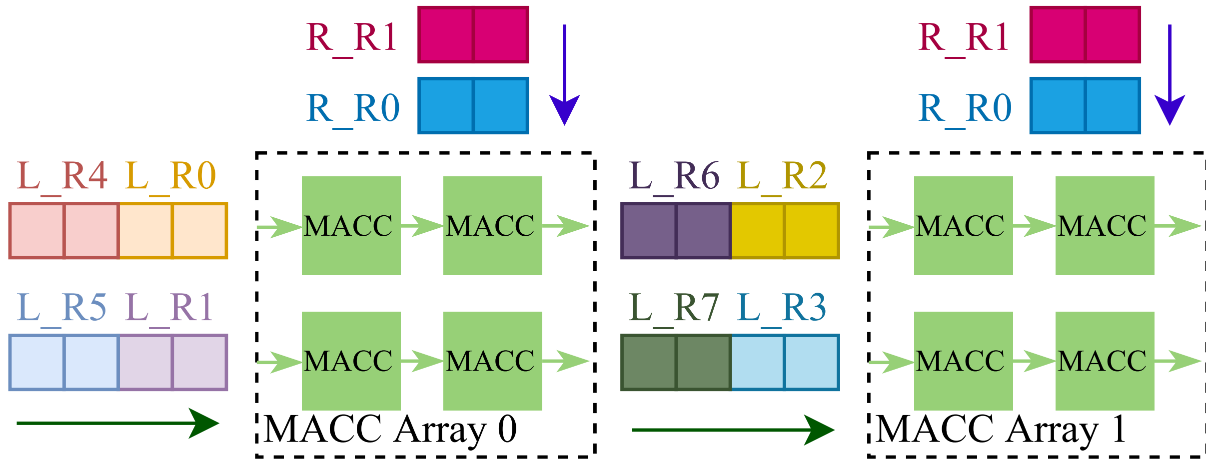


Figure 6.4: An example of allocating SpMM, MM or MM with transpose onto 2 *MACC Arrays*. The notion "L_R#" indicates the row index of the left matrix while the notion "R_R#" indicates the row index of the right matrix.

FPGA to compute square root and division needed by L2 normalization, respectively. As the L2 normalization and its gradients are computed in serial, we reuse most of the computation units to save resource utilization, as shown in Fig. 6.3(c). All the multiplexers in Fig. 6.3(c) are selected by the signal indicating the computation of L2 normalization or gradients. As expressed in Equ. (6.3) and Equ. (6.10), the $||X_{l-1}||_2$, which is produced by the square root module, is used in both computing L2 normalization and its gradients. Therefore, we buffer the results of the square root module h to eliminate redundant computation. Moreover, $\frac{\frac{\partial \mathcal{L}}{\partial X_{l-1}}}{||X_{l-1}||_2}$ and $\frac{X_{l-1}}{||X_{l-1}||_2}$ are also used several times in different computation steps for computing the gradients. Therefore, we buffer them after the first computation and reuse them to save computation resources and reduce latency.

6.4.3 Weight Update

Since the weights are updated after computing the gradients by MM with transpose, we design a separate module that takes the outputs of the PE array to fully pipeline the com-

putation, as shown in Fig. 6.2. The square root and division operations in Adam are also computed by using CORDIC and Division IPs in Xilinx FPGA. Moreover, we define the hyper parameters in Adam to the nearest power of 2 (*i.e.,* learning rate η) to simplify the multiplications to shift.

6.4.4 Data Communication

The *Communication Module* handles data transfer between external memory and FPGA (both with CPU and DDR4), and among different PEs, as shown in Fig. 6.2. The interface between CPU and FPGA is PCIe Gen3 X16, and only the initial data, final results of the forward phase and the first gradients for the backward phase are transferred via PCIe. DDR4 is also used as the external memory to save initial data for different training epochs and on-chip buffers are designed to perform in ping-pong manner, so that the communication time between DDR4 and FPGA can be hidden under the computation time. Since the computation of one layer is allocated to perform in parallel on different PEs (see allocation details in Section 6.4.5), we also transfer data among different PEs. Moreover, we also collect all the data of one layer and input into the *Weight Update Module* for updating weights after back-propagation. Considering the properties of FPGA (*i.e.,* constraints of the number of long connections between different super logic regions), we use FIFOs in the *PE Interface Modules* to control the bandwidth between different PEs.

6.4.5 Allocation and Scheduling

In this subsection, we will discuss in detail 1) how to allocate SpMM, MM and MM with transpose onto different PEs, and 2) how to schedule computation between CPU and FPGA as well as between different FPGA modules.

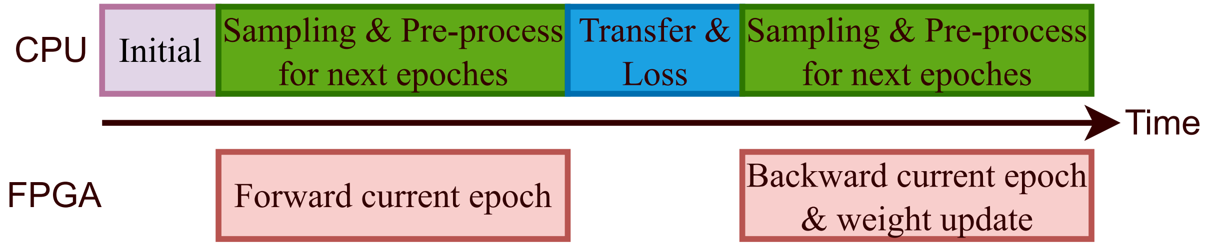


Figure 6.5: Scheduling between CPU and FPGA.

6.4.5.1 Allocation

For SpMM, MM and MM with transpose, we use a round robin method to assign different rows of the left matrix onto different rows of different PEs, as shown with a simple example in Fig. 6.4. In this way, we can hide the row information of the non-zero elements in the sparse matrix under the row index of the *MACC Array* in each PE, thus reducing the complexity and memory requirements of the CPCOO format. Moreover, the element of each row in the left matrix is fed into the MACCs one by one, and the element is accumulated to get the MM results. Therefore, the elements of the right matrix is fed into the PEs row by row, as shown by the blue arrow in Fig. 6.4. Since we are not able to perform the computation of the whole matrix, we do a row-wise partition in the left matrix and a column-wise partition in the right matrix to fit one tile into the *MACC Array* of each PE. In this way, each PE can only achieve a portion rows of the result matrix (in the example shown in Fig. 6.4, *MACC Array 0* achieves ROW 0/1/4/5 of the result matrix while *MACC Array 1* achieves ROW 2/3/6/7). Since each *MACC Array* only uses part of the rows in the left matrix, we will propagate the results inside each PE if the result matrix is used as the left matrix in next steps. Otherwise, we will communicate among different PEs to collect the whole result matrix.

6.4.5.2 Scheduling

During the training of GCN, most of the computation expensive operations are assigned to the FPGA while others are assigned to CPU. In order to improve the overall training performance, we parallel most of the operations between CPU and FPGA, as shown in Fig. 6.5. After the first data initialization, the CPU keeps on sampling and preprocessing data for the next epochs while the FPGA is forwarding the current epoch. The CPU is interrupted to transfer forward results and compute the softmax, cross-entropy loss and the corresponding gradients once the forward phase of one epoch is finished on the FPGA. The gradients are then transferred back to the FPGA for backward and the CPU is back to preparing data for next epochs. Once the data of one epoch is prepared on the CPU, the data will be transferred to the FPGA via PCIe and saved to the DDR4 for later use. In addition, the multi-core CPU can be set to work in parallel because the subgraphs are independent to each other.

6.5 Experimental Results

In this section, our proposed approach is evaluated with comprehensive experiments. We first evaluate the proposed 16-bit signed integer training approach on different datasets and networks to show its effectiveness. The FPGA accelerator is then evaluated on GraphSAINT with different configurations. Finally, we compare our work with a state-of-the-art FPGA accelerator with the same FPGA configurations.

6.5.1 Experimental Setup

SkeletonGCN is implemented with Verilog HDL and deployed on a Xilinx Alveo U200 board. According to the resources of the U200 board, we implement 8 PEs, each of which is equipped with a 32×16 *MACC Array* for SpMM, MM and MM with transpose. The

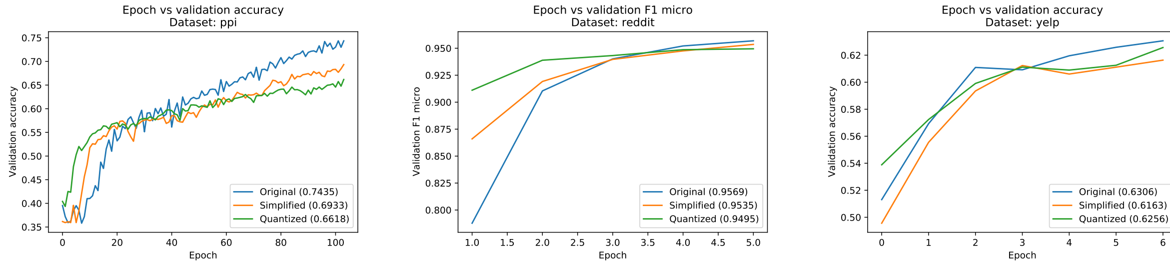


Figure 6.6: Training Accuracy Comparison

Table 6.3: Resource Utilization on Alveo U200 Board.

Resource	LUT	LUTRAM	BRAM	URAM	DSP
Used	1021386	183191	1338	598	4460
Available	1182240	591840	2160	960	6840
Utilization(%)	86.39	30.95	61.94	62.29	65.20

BRAMs and URAMs of the U200 board are used to store all the data used. After synthesis and implementation with Vivado 2020.1, the overall resource utilization is shown in Table 6.3. Our design relies heavily on on-chip memories (LUTRAM, BRAM and URAM) to buffer the graph data to resolve the issue of bandwidth constraints.

We consider the commonly used large datasets in our experiments, as shown in Table 6.1. We also list the densities of the features, edges and weights. As we cannot buffer all the graph data on board for large datasets, we take the the same sampling algorithms as GraphSAINT and GraphACT for fair comparison. All the results on CPU and GPU are generated by using PyTorch Geometric [FL19] and the open-sourced codes provided by GraphSAINT [ZZS19b] and GraphACT [ZP20].

6.5.2 Training Accuracy and Latency

Fig. 6.6 shows a training accuracy cross-comparison between three GraphSAINT [ZZS19b] implementations across three popular datasets. The original GraphSAINT configuration uses

Table 6.4: Training Latency Comparison.

		Intel Xeon	Nvidia Tesla P100	Ours
Data type		Float32	Float32	INT16
Frequency(GHz)		2.2	1.2	0.25
DSP/CPU/Cuda Cores		40	3584	4460
Total	PPI	352.5	8.3	7.1
convergence	Reddit	72.5	2.9	0.96
time (s)	Yelp	965.1	27.7	27.1

large subgraphs (*i.e.,* 8000 nodes) as proposed by the original authors. The simplified version uses 2000 node subgraphs to better fit our hardware, and removes a portion of functionality including dropout and batch normalization. The quantized version is the one we run on our hardware, mostly in 16-bit signed integer as mentioned in previous sections. As we can see, there were insignificant drops in F1 score of approximately 0.5-0.7% for Reddit and Yelp datasets. The drop for PPI was significant at 8% because the PPI dataset is less robust to smaller subgraph sizes during sampling [ZZS19b].

The training latency under the same configurations is shown in Table 6.4. On average, we can achieve 53.5× and 1.7× speedup compared with Intel Xeon CPU and Nvidia Tesla P100 GPU, respectively.

6.5.3 Comparison with State-of-the-art

We also compare our work with GraphACT to further show the effectiveness of our approach. We follow the same experimental settings, including network architecture, testing datasets and FPGA board, as those in GraphACT for fair comparison. The GCN evaluated has two graph convolution layers and one MLP layer in the classifier, and the hidden size is set to 256 for all graph convolution layers. As shown in Table 6.5, we have speedup up to 8.9×

Table 6.5: Comparison with GraphACT.

		GraphACT	Xeon Gold 5128	Nvidia Tesla P100	Ours
Data type		Float32	Float32	Float32	SINT16
Frequency(GHz)		0.2	2.2	1.2	0.25
DSP slices / CPU cores / CUDA cores		5632	40	3584	4460
Total convergence time (s)	PPI	9.6	151.4	10.6	0.85
	Reddit	7.6	95.5	11.4	0.87
	Yelp	23.4	359.4	30.4	3.76

compared with GraphACT across all datasets. On average, we still achieve 6.5× speedup on PPI, Reddit and Yelp.

The advantages come from both our quantization-aware training algorithm and our unified PE architecture. First, we reduce precision from 32-bit floating point to 16-bit signed integers with negligible accuracy loss, which in turn greatly reduces the usage of DSPs (in Xilinx FPGA, each Float32 multiplier consumers 3 DSPs while each INT16 multiplier only consumers 1 DSP). Therefore, we can have more multipliers than GraphACT for computation with same number of DSPs. Second, we use the unified PE architecture, which dramatically increase the DSP efficiency (see details in Section 6.5.4). In GraphACT, they design separate modules for feature aggregation and weight transformation. Although their scheduling algorithm tries to overlap the operations of both modules, there still has quantitative idle cycles for either of the two modules.

6.5.4 Discussion

We will discuss the DSP efficiency, which dramatically influences the overall training latency, in this subsection. The DSP efficiency is defined as follows:

$$DSP_EFF = \frac{Lat_{theo}}{Lat_{test}}, \tag{6.11}$$

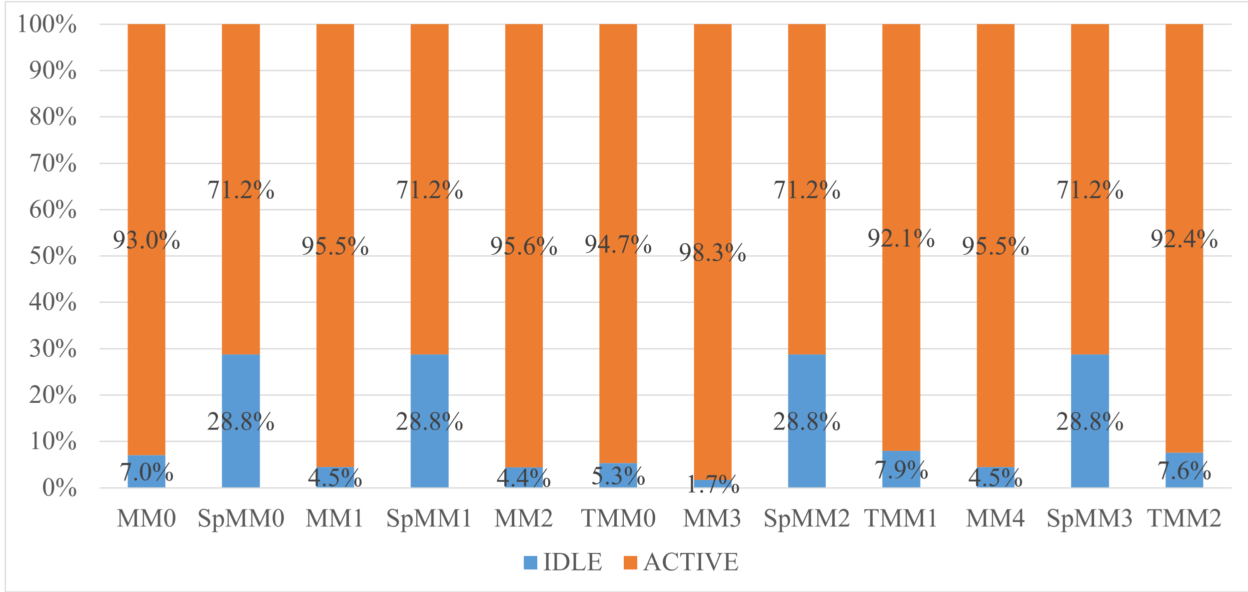


Figure 6.7: The DSP efficiency of computing SpMM, MM, and MM with transpose in training GCN on reddit. The MM with transpose is marked as "TMM" in the figure.

where the Lat_{theo} and Lat_{test} indicate the theoretical latency and tested latency, respectively. We skip all the zeros in SpMM and the theoretical latency of SpMM and MM are then calculated by using Equ. (6.12) and (6.13).

$$Lat_{theo}^{SpMM} = \frac{\#\ of\ non-zero\ MAC\ ops}{\#\ of\ MAC\ units}. \tag{6.12}$$

$$Lat_{theo}^{MM} = \frac{\#\ of\ MAC\ ops}{\#\ of\ MAC\ units}. \tag{6.13}$$

Following the above definitions, we analyze the average DSP efficiency of training GCN on reddit, as shown in Fig. 6.7. We can see that DSP efficiency for computing MM and MM with transpose can be up to 98.3% for some cases. The DSP efficiency of SpMM is 71.2% because we inject empty elements to avoid bank conflicts as that of the PCOO format. However, it has little influence on the total training latency because SpMM only accounts for around 1% of the total computation workloads.

6.6 Conclusion and Future Work

In this work, we propose a software-hardware co-optimized GCN accelerator on FPGA, named **SkeletonGCN**, to improve GCN training efficiency. The data representation in **SkeletonGCN** is first quantized from 32-bit floating point to 16-bit signed integer to reduce computation and storage requirements. In addition, we simplify the non-linear operations and eliminate redundant computations to better fit the computation on FPGA. Moreover, we employ a linear time sparse matrix compression algorithm to further reduce memory bandwidth while allowing efficient decompression on hardware. A unified hardware architecture is then proposed to compute SpMM, MM and MM with transpose to improve DSP efficiency. Evaluation first shows that our simplified training approach can train the network with negligible accuracy loss. Moreover, **SkeletonGCN** can achieve up to 11.3× speedup over existing FPGA-based accelerator while executing the same network structure and maintaining the same training accuracy. In addition, **SkeletonGCN** achieves up to 178× and 13.1× speedup over state-of-art CPU and GPU implementation, respectively.

CHAPTER 7

Summary

Deep learning algorithms have shown promising performance among various real-world applications. At the same, domain specific accelerators/processors that target on deep learning algorithms help improve the performance in real-time scenarios. In this thesis, we propose a series of software and hardware co-optimized acceleration for deep learning algorithms, including CNNs and GCNs.

Starting from 2016, we worked on the OPU project and proposed the base OPU architecture. With the base OPU, any CNN can be executed by instructions without the change of FPGA implementation. Moreover, we used 8-bit fixed point data representation in the base OPU to reduce computation and memory cost of CNNs. We evaluated the OPU on a real-world application with a cascade of three CNNs, where YoloV3 was used for car detection, TinyYolo was used for license plate detection and a self-trained CNN was used for license plate recognition. Compared with the resource comparable GPU, the OPU is 9.6× faster for running the three network in sequence.

After that, we explored more on the data representation in order to improve the quantization accuracy and further compress CNN. We first utilized the 8-bit floating-point data representation, and quantized the CNNs with 8-bit floating-point. The quantization strategy reduced the quantization error from around 1% to 0.5% compared with the 8-bit fixed-point data representation. Moreover, we propose the LPFP processor, which was designed to accelerate CNN inference quantized with 8-bit floating-point. By decomposing one DSP into four LPFP multipliers, the LPFP processor greatly improved the per DSP throughput. To

be specific, compared with seven FPGA accelerators on VGG16 and Yolo, the LPFP processor improves average throughput by 3.5× and 27.5× and average throughput per DSP by 4.1× and 5×, respectively.

We further compressed the CNNs with mixed precision data representation while maintaining quantization accuracy. The FPGA processor MP-OPU was also improved to support operations with mixed precision data during runtime. The experimental results show that MP-OPU achieves 12.9× latency reduction and 2.2× better throughput per DSP for conventional CNNs, while 7.6× latency reduction and 2.9× better throughput per DSP for lightweight CNNs, all on average compared with existing FPGA accelerators/processors, respectively.

Starting from 2020, we expand our work from CNNs to GCNs. The GCN inference phase was first optimized with the proposed LW-GCN. We first developed the PCOO format to compress the sparse matrix so that computation and storage was reduced. Based on the PCOO format, we proposed the unified PE architecture to accelerate both SpMM and MM. Compared with existing CPU, GPU and state-of-the-art FPGA-based accelerator, LW-GCN reduces latency by up to 60×, 12× and 1.7× and increases power efficiency by up to 912×, 511× and 3.87×, respectively. Moreover, compared with Nvidia's latest edge GPU Jetson Xavier NX, LW-GCN achieves speedup and energy savings of 32× and 84×, respectively.

Finally, we expand the support of GCN inference to GCN training to overcome the challenges in GCN training. We improved the PCOO format to reduce the bit width needed of empty rows, thus reducing the total storage requirements of the sparse matrices. Moreover, we developed the SkeletonGCN based on the improved PCOO format, and the unified PE architecture is improved to support SpMM, MM and MM with transpose efficiently. Evaluated on Alveo U200 FPGA board, SkeletonGCN can achieve up to 11.3× speedup while maintaining the same training accuracy with 16-bit fixed-point data representation compared with existing FPGA-based accelerator on the same network architecture. In addition, SkeletonGCN is 178× and 13.1× faster than state-of-the-art CPU and GPU implementation

on popular datasets, respectively.

REFERENCES

[AAA16] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. "Deep speech 2: End-to-end speech recognition in english and mandarin." In *International conference on machine learning*, pp. 173–182, 2016.

[ABC16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. "Tensorflow: A system for large-scale machine learning." In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.

[ADJ17] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. "Bit-pragmatic deep neural network computing." In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 382–394, 2017.

[AHB18] Mohamed S Abdelfattah, David Han, Andrew Bitar, Roberto DiCecco, Shane O'Connell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, Andrew C Ling, et al. "DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration." In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 411–4117. IEEE, 2018.

[AKT21] Rana Azzam, Felix H. Kong, Tarek Taha, and Yahya Zweiri. "Pose-Graph Neural Network Classifier for Global Optimality Prediction in 2D SLAM." *IEEE Access*, **9**:80466–80477, 2021.

[APR20] Afzal Ahmad, Muhammad Adeel Pasha, and Ghulam Jilani Raza. "Accelerating Tiny YOLOv3 using FPGA-Based Hardware/Software Co-Design." In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5. IEEE, 2020.

[Aut11] N. N. Author. "Suppressed for Anonymity.", 2011.

[AYS18] Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K Gupta, and Hadi Esmaeilzadeh. "Snapea: Predictive early activation for reducing computation in deep convolutional neural networks." In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 662–673. IEEE, 2018.

[BA19] Federico Baldassarre and Hossein Azizpour. "Explainability techniques for graph convolutional networks." *arXiv preprint arXiv:1905.13686*, 2019.

[BC14] Jimmy Ba and Rich Caruana. "Do Deep Nets Really Need to be Deep?" In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pp. 2654–2662, 2014.

[BCN06] Cristian Bucila, Rich Caruana, and Alexandru Niculescu-Mizil. "Model compression." In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, pp. 535–541, 2006.

[BMW11] Moez Baccouche, Franck Mamalet, Christian Wolf, Christophe Garcia, and Atilla Baskurt. "Sequential deep learning for human action recognition." In *Human Behavior Understanding*, pp. 29–39. Springer, 2011.

[BYB18a] A. Boutros, S. Yazdanshenas, and V. Betz. "Embracing Diversity: Enhanced DSP Blocks for Low-Precision Deep Learning on FPGAs." In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 35–357, 2018.

[BYB18b] Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. "Embracing diversity: Enhanced DSP blocks for low-precision deep learning on FPGAs." In *2018 28th International Conference on Field Programmable Logic and Applications*, pp. 35–357, 2018.

[BZH18] Lin Bai, Yiming Zhao, and Xinming Huang. "A CNN accelerator on FPGA using depthwise separable convolution." *IEEE Transactions on Circuits and Systems II: Express Briefs*, **65**(10):1415–1419, 2018.

[CBD14a] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. "Low precision arithmetic for deep learning." *CoRR*, **abs/1412.7024**, 2014.

[CBD14b] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. "Training deep neural networks with low precision multiplications." *arXiv preprint arXiv:1412.7024*, 2014.

[CBD15] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. "Binaryconnect: Training deep neural networks with binary weights during propagations." In *Advances in neural information processing systems*, pp. 3123–3131, 2015.

[CDS14] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning." In *ACM Sigplan Notices*, volume 49, pp. 269–284, 2014.

[CES16] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks." In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture*, pp. 367–379, 2016.

[Used to compare the array area and power, need to check.]

[CFO17] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengil, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. "Accelerating persistent neural networks at datacenter scale." In *Hot Chips*, volume 29, 2017.

[CGS13] Cecile Cabanes, Antoine Grouazel, Karina von Schuckmann, Michel Hamon, Victor Turpin, Christine Coatanoan, François Paris, Stephanie Guinehut, Cathy Boone, Nicolas Ferry, et al. "The CORA dataset: validation and diagnostics of in-situ ocean temperature and salinity measurements." *Ocean Science*, **9**(1):1–18, 2013.

[Cha14] Ken Chapman. "Multiplexer Design Techniques for Datapath Performance with Minimized Routing Resources." *Xilinx Application Note (XAPP522)*, October 31, 2014.

[CHS16] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1." *arXiv preprint arXiv:1602.02830*, 2016.

[CHS17] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. "Deep learning with low precision by half-wave gaussian quantization." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5918–5926, 2017.

[CHZ19] Yao Chen, Jiong He, Xiaofan Zhang, Cong Hao, and Deming Chen. "Cloud-DNN: An Open Framework for Mapping DNN Models to Cloud FPGAs." In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 73–82. ACM, 2019.

[CKE17] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks." *IEEE Journal of Solid-State Circuits*, **52**(1):127–138, 2017.

[CLC18] Gordon R Chiu, Andrew C Ling, Davor Capalija, Andrew Bitar, and Mohamed S Abdelfattah. "Flexibility: FPGAs and CAD in Deep Learning Acceleration." In *Proceedings of the 2018 International Symposium on Physical Design*, pp. 34–41, 2018.

[CLL14] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. "Dadiannao: A machine-learning supercomputer." In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622, 2014.

[CLS19a] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. "Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks." In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 257–266, 2019.

[CLS19b] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. "Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks." In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 257–266, 2019.

[CMB10] Srihari Cadambi, Abhinandan Majumdar, Michela Becchi, Srimat Chakradhar, and Hans Peter Graf. "A programmable parallel accelerator for learning and classification." In *PACT*, pp. 273–284. ACM, 2010.

[CMS12] Dan Ciresan, Ueli Meier, and Jürgen Schmidhuber. "Multi-column deep neural networks for image classification." In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pp. 3642–3649. IEEE, 2012.

[CNN18] Philip Colangelo, Nasibeh Nasiri, Eriko Nurvitadhi, Asit Mishra, Martin Margala, and Kevin Nealis. "Exploration of low numeric precision deep learning inference using intel® FPGAs." In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 73–80. IEEE, 2018.

[CS14] Christopher Clark and Amos Storkey. "Teaching deep convolutional neural networks to play go." *arXiv preprint arXiv:1412.3409*, 2014.

[CSJ10] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. "A dynamically configurable coprocessor for convolutional neural networks." In *ACM SIGARCH Computer Arch. News*, volume 38, pp. 247–257, 2010.

[CSW17] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. "Realtime multi-person 2d pose estimation using part affinity fields." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 7291–7299, 2017.

[CWC14] Cornelia Caragea, Jian Wu, Alina Ciobanu, Kyle Williams, Juan Fernández-Ramírez, Hung-Hsuan Chen, Zhaohui Wu, and Lee Giles. "Citeseer x: A scholarly big dataset." In *European Conference on Information Retrieval*, pp. 311–322. Springer, 2014.

[CWS19] Fanfei Chen, Jinkun Wang, Tixiao Shan, and Brendan Englot. "Autonomous exploration under uncertainty via graph convolutional networks." In *Proceedings of the International Symposium on Robotics Research*, 2019.

[CWT15] Wenlin Chen, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, and Yixin Chen. "Compressing Neural Networks with the Hashing Trick." *CoRR*, **abs/1504.04788**, 2015.

[CWV14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. "cuDNN: Efficient Primitives for Deep Learning." *CoRR*, **abs/1410.0759**, 2014.

[CWX20a] Xiaobing Chen, Yuke Wang, Xinfeng Xie, Xing Hu, Abanti Basak, Ling Liang, Mingyu Yan, Lei Deng, Yufei Ding, Zidong Du, et al. "Rubik: A hierarchical architecture for efficient graph learning." *arXiv preprint arXiv:2009.12495*, 2020.

[CWX20b] Xiaobing Chen, Yuke Wang, Xinfeng Xie, Xing Hu, Abanti Basak, Ling Liang, Mingyu Yan, Lei Deng, Yufei Ding, Zidong Du, et al. "Rubik: A hierarchical architecture for efficient graph learning." *arXiv preprint arXiv:2009.12495*, 2020.

[DCM12a] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. "Large scale distributed deep networks." In *Advance in Neural Info. Processing Systems*, pp. 1223–1231, 2012.

[DCM12b] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. "Large scale distributed deep networks." In *Advances in neural information processing systems*, pp. 1223–1231, 2012.

[DDS09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "Imagenet: A large-scale hierarchical image database." In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

[DFC15a] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. "ShiDianNao: Shifting vision processing closer to the sensor." In *ACM SIGARCH Computer Architecture News*, volume 43, pp. 92–104. ACM, 2015.

[DFC15b] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. "ShiDianNao: Shifting vision processing closer to the sensor." In *ACM SIGARCH Computer Architecture News*, volume 43, pp. 92–104, 2015.

[DHS00] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. John Wiley and Sons, 2nd edition, 2000.

[DL17] Franck Dernoncourt and Ji Young Lee. "Pubmed 200k rct: a dataset for sequential sentence classification in medical abstracts." *arXiv preprint arXiv:1710.06071*, 2017.

[DSD13] Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas. "Predicting Parameters in Deep Learning." In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States.*, pp. 2148–2156, 2013.

[Fei10] Li Fei-Fei. "ImageNet: crowdsourcing, benchmarking & other cool things." In *CMU VASC Seminar*, 2010.

[FL19] Matthias Fey and Jan E. Lenssen. "Fast Graph Representation Learning with PyTorch Geometric." In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[FLW18] Matthias Fey, Jan Eric Lenssen, Frank Weichert, and Heinrich Müller. "Splinecnn: Fast geometric deep learning with continuous b-spline kernels." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 869–877, 2018.

[FPH09] Clément Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun. "Cnp: An fpga-based processor for convolutional networks." In *FPL*, pp. 32–37. IEEE, 2009.

[FR04] Achim Frick and Arif Rochman. "Characterization of TPU-elastomers by thermal analysis (DSC)." *Polymer testing*, **23**(4):413–417, 2004.

[FWL20a] Boyuan Feng, Yuke Wang, Xu Li, Shu Yang, Xueqiao Peng, and Yufei Ding. "Sgquant: Squeezing the last bit on graph neural networks with specialized quantization." In *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 1044–1052. IEEE, 2020.

[FWL20b] Boyuan Feng, Yuke Wang, Xu Li, Shu Yang, Xueqiao Peng, and Yufei Ding. "Sgquant: Squeezing the last bit on graph neural networks with specialized quantization." In *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 1044–1052. IEEE, 2020.

[GHY17] Kaiyuan Guo, Song Han, Song Yao, Yu Wang, Yuan Xie, and Huazhong Yang. "Software-hardware codesign for efficient neural network acceleration." *IEEE Micro*, **37**(2):18–25, 2017.

[GJW19] Chengyue Gong, Zixuan Jiang, Dilin Wang, Yibo Lin, Qiang Liu, and David Z Pan. "Mixed Precision Neural Architecture Search for Energy Efficient Deep Learning." In *ICCAD*, pp. 1–7, 2019.

[GLS19] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, and Martin Herbordt. "AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing.", 2019.

[GLS20] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. "AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing." In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 922–936. IEEE, 2020.

[GLU12] Andreas Geiger, Philip Lenz, and Raquel Urtasun. "Are we ready for autonomous driving? the kitti vision benchmark suite." In *Computer Vision and Pattern Recognition (CVPR)*, pp. 3354–3361. IEEE, 2012.

[GSQ17] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. "Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **37**(1):35–47, 2017.

[GSQ18] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. "Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **37**(1):35–47, 2018.

[GZY19] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. "[DL] A survey of FPGA-based neural network inference accelerators." *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, **12**(1):1–26, 2019.

[Hau18] Elmar Haußmann. "Benchmarking Google's new TPUv2 ." `https://blog.riseml.com/benchmarking-googles-new-tpuv2-121c03b71384`, 2018.

[HLM16a] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. "EIE: efficient inference engine on compressed deep neural network." In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture*, pp. 243–254, 2016.

[HLM16b] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. "EIE: Efficient inference engine on compressed deep neural network." *ACM SIGARCH Computer Architecture News*, **44**(3):243–254, 2016.

[HLV17] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. "Densely connected convolutional networks." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

[HMD15a] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding." *arXiv preprint arXiv:1510.00149*, 2015.

[HMD15b] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding." *arXiv preprint arXiv:1510.00149*, 2015.

[HPT15] Song Han, Jeff Pool, John Tran, and William Dally. "Learning both weights and connections for efficient neural network." In *Advances in neural information processing systems*, pp. 1135–1143, 2015.

[HVD15] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. "Distilling the Knowledge in a Neural Network." *CoRR*, **abs/1503.02531**, 2015.

[HYL17a] Will Hamilton, Zhitao Ying, and Jure Leskovec. "Inductive representation learning on large graphs." *Advances in neural information processing systems*, **30**, 2017.

[HYL17b] Will Hamilton, Zhitao Ying, and Jure Leskovec. "Inductive representation learning on large graphs." *Advances in neural information processing systems*, **30**, 2017.

[HYL17c] William L Hamilton, Rex Ying, and Jure Leskovec. "Inductive representation learning on large graphs." In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pp. 1025–1035, 2017.

[HYL17d] William L Hamilton, Rex Ying, and Jure Leskovec. "Inductive representation learning on large graphs." *arXiv preprint arXiv:1706.02216*, 2017.

[HZC17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. "Mobilenets: Efficient convolutional neural networks for mobile vision applications." *arXiv preprint arXiv:1704.04861*, 2017.

[HZR16a] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[HZR16b] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[HZR16c] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[HZR16d] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Identity mappings in deep residual networks." In *European conference on computer vision*, pp. 630–645. Springer, 2016.

[Inc20] Nvidia Inc. "Jetson Xavier NX.", 2020.

[Inc21a] Intel Inc. "INT8 vs FP32 Comparison on Select Networks and Platforms.", 2021.

[Inc21b] Intel Inc. "OpenVINO™ Model Server Benchmark Results.", 3 2021.

[Inc21c] Nvidia Inc. "torch2trt.", 2021.

[IS15] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." *arXiv preprint arXiv:1502.03167*, 2015.

[JAH16a] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. "Stripes: Bit-serial deep neural network computing." In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1–12, 2016.

[JAH16b] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. "Stripes: Bit-serial deep neural network computing." In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12. IEEE, 2016.

[JGW19] Sambhav R Jain, Albert Gural, Michael Wu, and Chris H Dick. "Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks." *arXiv preprint arXiv:1903.08066*, 2019.

[JKC18] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. "Quantization and training of neural networks for efficient integer-arithmetic-only inference." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, 2018.

[JSD14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. "Caffe: Convolutional Architecture for Fast Feature Embedding." *arXiv preprint arXiv:1408.5093*, 2014.

[JXY13] Shuiwang Ji, Wei Xu, Ming Yang, and Kai Yu. "3D convolutional neural networks for human action recognition." *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, **35**(1):221–231, 2013.

[JYP17a] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. "In-datacenter performance analysis of a tensor processing unit." In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pp. 1–12. IEEE, 2017.

[JYP17b] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. "In-datacenter performance analysis of a tensor processing unit." In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

[JYP17c] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. "In-datacenter performance analysis of a tensor processing unit." In *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

[KB14] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980*, 2014.

[Kea89] M. J. Kearns. *Computational Complexity of Machine Learning*. PhD thesis, Department of Computer Science, Harvard University, 1989.

[KH09] Alex Krizhevsky and Geoffrey Hinton. "Learning multiple layers of features from tiny images.", 2009.

[KMY19] Shreyas Kolala Venkataramanaiah, Yufei Ma, Shihui Yin, Eriko Nurvithadhi, Aravind Dasu, Yu Cao, and Jae-Sun Seo. "Automatic Compiler Based FPGA Accelerator for CNN Training." In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 166–172, 2019.

[KSH12a] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks." In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[KSH12b] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks." In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[KW16a] Thomas N Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks." *arXiv preprint arXiv:1609.02907*, 2016.

[KW16b] Thomas N Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks." *arXiv preprint arXiv:1609.02907*, 2016.

[Lan00] P. Langley. "Crafting Papers on Machine Learning." In Pat Langley, editor, *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

[LCL15] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. "Pudiannao: A polyvalent machine learning accelerator." In *ACM SIGARCH Computer Architecture News*, volume 43, pp. 369–381. ACM, 2015.

[LCM15] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. "Neural networks with few multiplications." *arXiv preprint arXiv:1510.03009*, 2015.

[LDS89] Yann LeCun, John S. Denker, and Sara A. Solla. "Optimal Brain Damage." In *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*, pp. 598–605, 1989.

[LDT16a] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. "Cambricon: An instruction set architecture for neural networks." In *ACM SIGARCH Computer Architecture News*, volume 44, pp. 393–405, 2016.

[LDT16b] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. "Cambricon: An instruction set architecture for neural networks." In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 393–405. IEEE, 2016.

[LDT16c] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. "Cambricon: An Instruction Set Architecture for Neural Networks." In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 393–405, 2016.

[LG15] Nicholas D. Lane and Petko Georgiev. "Can Deep Learning Revolutionize Mobile Sensing?" In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, HotMobile '15, pp. 117–122, New York, NY, USA, 2015. ACM.

[LLH05] Fei Li, Yizhou Lin, Lei He, Deming Chen, and Jason Cong. "Power modeling and characteristics of field programmable gate arrays." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **24**(11):1712–1724, 2005.

[LLK21] Jiajun Li, Ahmed Louri, Avinash Karanth, and Razvan Bunescu. "GCNAX: A Flexible and Energy-efficient Accelerator for Graph Convolutional Neural Networks." In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 775–788, 2021.

[LLS19] Xiaocong Lian, Zhenyu Liu, Zhourui Song, Jiwu Dai, Wei Zhou, and Xiangyang Ji. "High-Performance FPGA-Based CNN Accelerator With Block-Floating-Point Arithmetic." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[LLX17] Liqiang Lu, Yun Liang, Qingcheng Xiao, and Shengen Yan. "Evaluating fast algorithms for convolutional neural networks on FPGAs." In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 101–108. IEEE, 2017.

[LM13] Steve Leibson and Nick Mehta. "Xilinx UltraScale: The Next-Generation Architecture for Your Next-Generation Architecture." *Xilinx White Paper WP435*, 2013.

[LMB14] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. "Microsoft coco: Common objects in context." In *European conference on computer vision*, pp. 740–755. Springer, 2014.

[Log12] IP LogiCORE. "Floating-Point Operator v6. 0." *Xilinx Inc*, 2012.

[LSC17] Liangzhen Lai, Naveen Suda, and Vikas Chandra. "Deep convolutional neural network inference with floating-point weights and fixed-point activations." *arXiv preprint arXiv:1703.03073*, 2017.

[LTA16] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. "Fixed point quantization of deep convolutional networks." In *International Conference on Machine Learning*, pp. 2849–2858, 2016.

[They used fix point to quantize CNNs, and they used the SQNR as the metric to evaluate the performance of the quantization. They also analyzed the influence of the quantization of each layer on the whole network with SQNR (This analysis is only used for uniform distribution inputs).]

[LWC18] Cheng Luo, Yuhua Wang, Wei Cao, Philip HW Leong, and Lingli Wang. "RNA: An Accurate Residual Network Accelerator for Quantized and Reconstructed Deep Neural Networks." In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 60–603. IEEE, 2018.

[LWL20a] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, LI Huawei, Dawen Xu, and Xiaowei Li. "EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks." *IEEE Transactions on Computers*, 2020.

[LWL20b] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, LI Huawei, Dawen Xu, and Xiaowei Li. "Engn: A high-throughput and energy-efficient accelerator for large

graph neural networks." *IEEE Transactions on Computers*, **70**(9):1511–1525, 2020.

[LYL17] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks." In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pp. 553–564. IEEE, 2017.

[LYL18] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. "FP-BNN: Binarized neural network on FPGA." *Neurocomputing*, **275**:1072–1086, 2018.

[LZP21] Yi-Chien Lin, Bingyi Zhang, and Viktor K. Prasanna. "HP-GNN: Generating High Throughput GNN Training Implementation on CPU-FPGA Heterogeneous Platform." *CoRR*, **abs/2112.11684**, 2021.

[Max60] Joel Max. "Quantizing for minimum distortion." *IRE Transactions on Information Theory*, **6**(1):7–12, 1960.

[MBB08] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, et al. "Junior: The stanford entry in the urban challenge." *Journal of field Robotics*, **25**(9):569–597, 2008.

[MCG17] Jing Ma, Li Chen, and Zhiyong Gao. "Hardware implementation and optimization of tiny-yolo network." In *International Forum on Digital TV and Wireless Multimedia Communications*, pp. 224–234. Springer, 2017.

[MCM83] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors. *Machine Learning: An Artificial Intelligence Approach, Vol. I.* Tioga, Palo Alto, CA, 1983.

[MCV17a] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. "An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks." In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pp. 1–8. IEEE, 2017.

[MCV17b] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks." In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 45–54. ACM, 2017.

[MCV18] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. "Optimizing the convolution operation to accelerate deep neural networks on FPGA." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **26**(7):1354–1367, 2018.

[MDH12] Abdel-rahman Mohamed, George E Dahl, and Geoffrey Hinton. "Acoustic modeling using deep belief networks." *Audio, Speech, and Language Processing, IEEE Transactions on*, **20**(1):14–22, 2012.

[MDL18] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. "Nvidia tensor core programmability, performance & precision." In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 522–531. IEEE, 2018.

[MGA16] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. "Design space exploration of fpga-based deep convolutional neural networks." In *Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific*, pp. 575–580. IEEE, 2016.

[Mic14] DRAM Micron. "System Power Calculators.", 2014.

[Mig17] Szymon Migacz. "8-bit inference with TensorRT." In *GPU Technology Conference*, 2017.

[Mit80] T. M. Mitchell. "The Need for Biases in Learning Generalizations." Technical report, Computer Science Department, Rutgers University, New Brunswick, MA, 1980.

[MLN17] Chunsheng Mei, Zhenyu Liu, Yue Niu, Xiangyang Ji, Wei Zhou, and Dongsheng Wang. "A 200MHZ 202.4 GFLOPS@ 10.8 W VGG16 accelerator in Xilinx VX690T." In *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pp. 784–788. IEEE, 2017.

[MMN18] Asuka Maki, Daisuke Miyashita, Kengo Nakata, Fumihiko Tachibana, Tomoya Suzuki, and Jun Deguchi. "Fpga-based cnn processor with filter-wise-optimized bit precision." In *2018 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, pp. 47–50. IEEE, 2018.

[MTR03] Sayan Mukherjee, Pablo Tamayo, Simon Rogers, Ryan Rifkin, Anna Engle, Colin Campbell, Todd R Golub, and Jill P Mesirov. "Estimating dataset size requirements for classifying DNA microarray data." *Journal of computational biology*, **10**(2):119–142, 2003.

[NKL20] Duy Thanh Nguyen, Hyun Kim, and Hyuk-Jae Lee. "Layer-specific Optimization for Mixed Data Flow with Mixed Precision in FPGA Design for CNN-based Object Detectors." *IEEE Transactions on Circuits and Systems for Video Technology*, 2020.

[NMS19] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. "Deep learning recommendation

model for personalization and recommendation systems." *arXiv preprint arXiv:1906.00091*, 2019.

[NR81] A. Newell and P. S. Rosenbloom. "Mechanisms of Skill Acquisition and the Law of Practice." In J. R. Anderson, editor, *Cognitive Skills and Their Acquisition*, chapter 1, pp. 1–51. Lawrence Erlbaum Associates, Inc., Hillsdale, NJ, 1981.

[Nvi08] CUDA Nvidia. "Cublas library." *NVIDIA Corporation, Santa Clara, California*, **15**(27):31, 2008.

[NYF18] Hiroki Nakahara, Haruyoshi Yonekawa, Tomoya Fujii, and Shimpei Sato. "A lightweight yolov2: A binarized cnn with a parallel support vector regression for an fpga." In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 31–40. ACM, 2018.

[OHL08] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. "GPU computing." *Proceedings of the IEEE*, **96**(5):879–899, 2008.

[ON15] Keiron O'Shea and Ryan Nash. "An introduction to convolutional neural networks." *arXiv preprint arXiv:1511.08458*, 2015.

[ORK15a] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. "Accelerating deep convolutional neural networks using specialized hardware." *Microsoft Research Whitepaper*, **2**, 2015.

[ORK15b] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. "Accelerating deep convolutional neural networks using specialized hardware." *Microsoft Research Whitepaper*, **2**(11):1–4, 2015.

[PAY17] Eunhyeok Park, Junwhan Ahn, and Sungjoo Yoo. "Weighted-entropy-based quantization for deep neural networks." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5456–5464, 2017.

[PBP17] Adrien Prost-Boucle, Alban Bourge, Frédéric Pétrot, Hande Alemdar, Nicholas Caldwell, and Vincent Leroy. "Scalable high-performance architecture for convolutional ternary neural networks on FPGA." In *2017 27th International Conference on Field Programmable Logic and Applications*, pp. 1–7, 2017.

[PG72] M Paez and T Glisson. "Minimum mean-squared-error quantization in speech PCM and DPCM systems." *IEEE Transactions on Communications*, **20**(2):225–230, 1972.

[PGM19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. "Pytorch: An imperative style, high-performance deep learning library." In *Advances in neural information processing systems*, pp. 8026–8037, 2019.

[PKY18] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. "Energy-efficient neural network accelerator based on outlier-aware low-precision computation." In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, pp. 688–698, 2018.

[PSM13] Maurice Peemen, Arnaud Setio, Bart Mesman, Henk Corporaal, et al. "Memory-centric accelerator design for convolutional neural networks." In *ICCD*, pp. 13–19. IEEE, 2013.

[PYV18] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. "Value-aware quantization for training and inference of neural networks." In *Proceedings of the European Conference on Computer Vision*, pp. 580–595, 2018.

[QWY16] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. "Going deeper with embedded fpga platform for convolutional neural network." In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 26–35. ACM, 2016.

[RBK14] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. "FitNets: Hints for Thin Deep Nets." *CoRR*, **abs/1412.6550**, 2014.

[RDG16a] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. "You only look once: Unified, real-time object detection." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[RDG16b] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. "You only look once: Unified, real-time object detection." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[RDS15a] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. "ImageNet Large Scale Visual Recognition Challenge." *IJCV*, **115**(3):211–252, 2015.

[RDS15b] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. "ImageNet Large Scale Visual Recognition Challenge." *International Journal of Computer Vision*, **115**(3):211–252, 2015.

[@articleILSVRC15, Author = 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, Title = ImageNet Large Scale Visual Recognition Challenge, Year = 2015, journal = International Journal of Computer Vision (IJCV), doi = 10.1007/s11263-015-0816-y, volume=115, number=3, pages=211-252 .]

[Red16] Joseph Redmon. "Darknet: Open Source Neural Networks in C." `http://pjreddie.com/darknet/`, 2013–2016.

[RF17a] Joseph Redmon and Ali Farhadi. "YOLO9000: better, faster, stronger." *arXiv preprint*, 2017.

[RF17b] Joseph Redmon and Ali Farhadi. "YOLO9000: better, faster, stronger." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7263–7271, 2017.

[RF18] Joseph Redmon and Ali Farhadi. "Yolov3: An incremental improvement." *arXiv preprint arXiv:1804.02767*, 2018.

[ROR16] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. "Xnor-net: Imagenet classification using binary convolutional neural networks." In *European Conference on Computer Vision*, pp. 525–542, 2016.

[RZW19] S. Rasoulinezhad, H. Zhou, L. Wang, and P. H. W. Leong. "PIR-DSP: An FPGA DSP Block Architecture for Multi-precision Deep Neural Networks." In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 35–44, 2019.

[Sam59] A. L. Samuel. "Some Studies in Machine Learning Using the Game of Checkers." *IBM Journal of Research and Development*, **3**(3):211–229, 1959.

[SBD18] Sean O Settle, Manasa Bollavaram, Paolo D'Alberto, Elliott Delaye, Oscar Fernandez, Nicholas Fraser, Aaron Ng, Ashish Sirasao, and Michael Wu. "Quantizing convolutional neural networks for low-power high-throughput inference engines." *arXiv preprint arXiv:1805.07941*, 2018.

[SCC19] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Viji Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. "Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks." *Advances in neural information processing systems*, **32**, 2019.

[SCD16] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks." In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 16–25. ACM, 2016.

[Sch10] Mike Schuster. "Speech recognition for mobile devices at Google." In *PRICAI 2010: Trends in Artificial Intelligence*, pp. 8–10. Springer, 2010.

[SCS22] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. "Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication." In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–77, 2022.

[SGT08] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. "The graph neural network model." *IEEE transactions on neural networks*, **20**(1):61–80, 2008.

[SHK14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: a simple way to prevent neural networks from overfitting." *The Journal of Machine Learning Research*, **15**(1):1929–1958, 2014.

[SHZ18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. "Mobilenetv2: Inverted residuals and linear bottlenecks." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

[SIV17] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. "Inception-v4, inception-resnet and the impact of residual connections on learning." In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[SJC09] Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, Srimat Chakradhar, Igor Durdanovic, Eric Cosatto, and Hans Peter Graf. "A massively parallel coprocessor for convolutional neural networks." In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pp. 53–60. IEEE, 2009.

[SKB18] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. "Modeling relational data with graph convolutional networks." In *European semantic web conference*, pp. 593–607. Springer, 2018.

[SLJ15a] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[SLJ15b] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going Deeper with Convolutions." In *CVPR 2015*, 2015.

[SLJ15c] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[SLW18] Zhourui Song, Zhenyu Liu, and Dongsheng Wang. "Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design." In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[SMH11] Ilya Sutskever, James Martens, and Geoffrey E Hinton. "Generating text with recurrent neural networks." In *ICML*, 2011.

[SPM16] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. "From high-level deep neural models to FPGAs." In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 17. IEEE Press, 2016.

[SPS18a] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network." In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 764–775. IEEE, 2018.

[SPS18b] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks." In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pp. 764–775, 2018.

[SS08] Huifang Sun and Yun Q Shi. "Image and video compression for multimedia engineering: Fundamentals, algorithms, and standards.", 2008.

[SVI16a] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. "Rethinking the inception architecture for computer vision." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

[SVI16b] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. "Rethinking the inception architecture for computer vision." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

[SZ14a] K. Simonyan and A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." *CoRR*, **abs/1409.1556**, 2014.

[SZ14b] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556*, 2014.

[SZ14c] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556*, 2014.

[SZ14d] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556*, 2014.

[SZF22] Xinkai Song, Tian Zhi, Zhe Fan, Zhenxing Zhang, Xi Zeng, Wei Li, Xing Hu, Zidong Du, Qi Guo, and Yunji Chen. "Cambricon-G: A Polyvalent Energy-Efficient Accelerator for Dynamic Graph Neural Networks." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **41**(1):116–128, 2022.

[SZG20a] Mengying Sun, Sendong Zhao, Coryandar Gilvary, Olivier Elemento, Jiayu Zhou, and Fei Wang. "Graph convolutional networks for computational drug development and discovery." *Briefings in bioinformatics*, **21**(3):919–935, 2020.

[SZG20b] Mengying Sun, Sendong Zhao, Coryandar Gilvary, Olivier Elemento, Jiayu Zhou, and Fei Wang. "Graph convolutional networks for computational drug development and discovery." *Briefings in bioinformatics*, **21**(3):919–935, 2020.

[SZH18] Mingcong Song, Jiechen Zhao, Yang Hu, Jiaqi Zhang, and Tao Li. "Prediction based execution on deep neural networks." In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 752–763. IEEE, 2018.

[TCP19] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. "Mnasnet: Platform-aware neural architecture search for mobile." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2820–2828, 2019.

[TFL20a] Shyam A Tailor, Javier Fernandez-Marques, and Nicholas D Lane. "Degree-quant: Quantization-aware training for graph neural networks." *arXiv preprint arXiv:2008.05000*, 2020.

[TFL20b] Shyam A Tailor, Javier Fernandez-Marques, and Nicholas D Lane. "Degree-quant: Quantization-aware training for graph neural networks." *arXiv preprint arXiv:2008.05000*, 2020.

[TL19] Mingxing Tan and Quoc Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks." In *International Conference on Machine Learning*, pp. 6105–6114, 2019.

[TWL21] Zhuofu Tao, Chen Wu, Yuan Liang, and Lei He. "LW-GCN: A Lightweight FPGA-based Graph Convolutional Network Accelerator." *ArXiv*, **abs/2111.03184**, 2021.

[TWO18] Kiran K Thekumparampil, Chong Wang, Sewoong Oh, and Li-Jia Li. "Attention-based graph neural network for semi-supervised learning." *arXiv preprint arXiv:1803.03735*, 2018.

[UFG17] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. "Finn: A framework for fast, scalable binarized neural network inference." In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74, 2017.

[URS18] Yaman Umuroglu, Lahiru Rasnayake, and Magnus Själander. "Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing." In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 307–3077. IEEE, 2018.

[VCC17] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. "Graph attention networks." *arXiv preprint arXiv:1710.10903*, 2017.

[VLG18] Sebastian Vogel, Mengyu Liang, Andre Guntoro, Walter Stechele, and Gerd Ascheid. "Efficient hardware acceleration of CNNs using logarithmic data representation with arbitrary log-base." In *2018 IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–8, 2018.

[VRR14] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. "AxNN: energy-efficient neuromorphic systems using approximate computing." In *Proceedings of the 2014 international symposium on Low power electronics and design*, pp. 27–32, 2014.

[VSY20] Shreyas K. Venkataramanaiah, Han-Sok Suh, Shihui Yin, Eriko Nurvitadhi, Aravind Dasu, Yu Cao, and Jae-sun Seo. "FPGA-Based Low-Batch Training Accelerator for Modern CNNs Featuring High Bandwidth Memory." In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery.

[WCB18] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. "Training deep neural networks with 8-bit floating point numbers." In *Advances in neural information processing systems*, pp. 7686–7695, 2018.

[WCC18] Di Wu, Jin Chen, Wei Cao, and Lingli Wang. "A Novel Low-Communication Energy-Efficient Reconfigurable CNN Acceleration Architecture." In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 64–643. IEEE, 2018.

[WDZ19a] Erwei Wang, James J Davis, Ruizhe Zhao, Ho-Cheung Ng, Xinyu Niu, Wayne Luk, Peter YK Cheung, and George A Constantinides. "Deep Neural Network Approximation for Custom Hardware: Where We've Been, Where We're Going." *arXiv preprint arXiv:1901.06955*, 2019.

[WDZ19b] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 10734–10742, 2019.

[WDZ19c] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search." In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10734–10742, 2019.

[WLD18] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. "C-lstm: Enabling efficient lstm using structured compression techniques on fpgas." In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 11–20. ACM, 2018.

[WLL19a] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. "HAQ: Hardware-Aware Automated Quantization with Mixed Precision." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 8612–8620, 2019.

[WLL19b] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. "Haq: Hardware-aware automated quantization with mixed precision." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8612–8620, 2019.

[WLL19c] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. "Haq: Hardware-aware automated quantization with mixed precision." In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8612–8620, 2019.

[WMS18] Yap June Wai, Zulkalnain bin Mohd Yussof, Sani Irwan bin Salim, and Lim Kim Chuan. "Fixed point implementation of tiny-yolo-v2 using opencl on fpga." *International Journal of Advanced Computer Science and Applications*, **9**(10):506–512, 2018.

[WWC20] Chen Wu, Mingyu Wang, Xinyuan Chu, Kun Wang, and Lei He. "Low Precision Floating-point Arithmetic for High Performance FPGA-based CNN Acceleration." *arXiv preprint arXiv:2003.03852*, 2020.

[WWC21] Chen Wu, Mingyu Wang, Xinyuan Chu, Kun Wang, and Lei He. "Low-Precision Floating-Point Arithmetic for High-Performance FPGA-Based CNN Acceleration." *ACM Trans. Reconfigurable Technol. Syst.*, **15**(1), nov 2021.

[WWL20] Chen Wu, Mingyu Wang, Xiayu Li, Jicheng Lu, Kun Wang, and Lei He. "Phoenix: A Low-Precision Floating-Point Quantization Oriented Architecture for Convolutional Neural Networks." *arXiv preprint arXiv:2003.02628*, 2020.

[WWZ18] Bichen Wu, Yanghan Wang, Peizhao Zhang, Yuandong Tian, Peter Vajda, and Kurt Keutzer. "Mixed precision quantization of convnets via differentiable neural architecture search." *arXiv preprint arXiv:1812.00090*, 2018.

[WYZ17a] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs." In *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 29. ACM, 2017.

[WYZ17b] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs." In *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 29, 2017.

[WZW21] Chen Wu, Jinming Zhuang, Kun Wang, and Lei He. "MP-OPU: A Mixed Precision FPGA-based Overlay Processor for Convolutional Neural Networks." In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 33–37, 2021.

[XHL18] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[Xil] Xilinx. "Vivado2018.2.".

[Xil15] Xilinx. "LogiCORE IP Distributed Memory Generator v8.0 Product Guide." *Xilinx Product Guide*, 2015.

[Xil18] Xilinx. "7 Series DSP48E1 Slice User Guide (UG479).", 3 2018.

[Xil20] Xilinx. "UltraScale Architecture DSP Slice (UG579).", 2020.

[XLL17a] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs." In *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 62. ACM, 2017.

[XLL17b] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs." In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6. IEEE, 2017.

[XYL14] Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. "Distributed Power-law Graph Computing: Theoretical and Empirical Analysis." In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[YDH20a] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. "Hygcn: A gcn accelerator with hybrid architecture." In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 15–29. IEEE, 2020.

[YDH20b] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. "Hygcn: A gcn accelerator with hybrid architecture." In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 15–29. IEEE, 2020.

[YHC18a] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. "Graph convolutional neural networks for web-scale recommender systems." In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 974–983, 2018.

[YHC18b] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. "Graph convolutional neural networks for web-scale recommender systems." In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 974–983, 2018.

[YHL20] Haibao Yu, Qi Han, Jianbo Li, Jianping Shi, Guangliang Cheng, and Bin Fan. "Search what you want: Barrier panelty nas for mixed precision quantization." In *European Conference on Computer Vision*, pp. 1–16. Springer, 2020.

[YWZ19a] Yunxuan Yu, Chen Wu, Tiandong Zhao, Kun Wang, and Lei He. "OPU: An FPGA-Based Overlay Processor for Convolutional Neural Networks." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[YWZ19b] Yunxuan Yu, Chen Wu, Tiandong Zhao, Kun Wang, and Lei He. "Opu: An fpga-based overlay processor for convolutional neural networks." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **28**(1):35–47, 2019.

[YWZ19c] Yunxuan Yu, Chen Wu, Tiandong Zhao, Kun Wang, and Lei He. "OPU: An FPGA-based overlay processor for convolutional neural networks." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **28**(1):35–47, 2019.

[YWZ20] Yunxuan Yu, Chen Wu, Tiandong Zhao, Kun Wang, and Lei He. "OPU: An FPGA-Based Overlay Processor for Convolutional Neural Networks." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **28**(1):35–47, 2020.

[YZW20a] Yunxuan Yu, Tiandong Zhao, Kun Wang, and Lei He. "Light-OPU: An FPGA-based Overlay Processor for Lightweight Convolutional Neural Networks." In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 122–132, 2020.

[YZW20b] Yunxuan Yu, Tiandong Zhao, Kun Wang, and Lei 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*, pp. 122–132, 2020.

[YZW20c] Yunxuan Yu, Tiandong Zhao, Kun Wang, and Lei 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*, FPGA '20, p. 122–132, New York, NY, USA, 2020. Association for Computing Machinery.

[YZW20d] Yunxuan Yu, Tiandong Zhao, Mingyu Wang, Kun Wang, and Lei He. "Uni-OPU: An FPGA-Based Uniform Accelerator for Convolutional and Transposed Convolutional Networks." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2020.

[ZCA08] Dan Zuras, Mike Cowlishaw, Alex Aiken, Matthew Applegate, David Bailey, Steve Bass, Dileep Bhandarkar, Mahesh Bhat, David Bindel, Sylvie Boldo, et al. "IEEE standard for floating-point arithmetic." *IEEE Std 754-2008*, pp. 1–70, 2008.

[ZDG18] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. "Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach." In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 15–28, 2018.

[ZDZ16a] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. "Cambricon-x: An accelerator for sparse neural networks." In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1–12, 2016.

[ZDZ16b] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. "Cambricon-X: An accelerator for sparse neural networks." In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12. IEEE, 2016.

[ZFZ16] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. "Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks." In *Proceedings of the 35th International Conference on Computer-Aided Design*, p. 12. ACM, 2016.

[ZGG19] Yiren Zhao, Xitong Gao, Xuan Guo, Junyi Liu, Erwei Wang, Robert Mullins, Peter YK Cheung, George Constantinides, and Cheng-Zhong Xu. "Automatic generation of multi-precision multi-arithmetic CNN accelerators for FPGAs." In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pp. 45–53. IEEE, 2019.

[ZHM16a] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. "Trained ternary quantization." *arXiv preprint arXiv:1612.01064*, 2016.

[ZHM16b] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. "Trained ternary quantization." *arXiv preprint arXiv:1612.01064*, 2016.

[ZL16] Barret Zoph and Quoc V Le. "Neural architecture search with reinforcement learning." *arXiv preprint arXiv:1611.01578*, 2016.

[ZLS15a] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. "Optimizing fpga-based accelerator design for deep convolutional neural networks." In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170. ACM, 2015.

[ZLS15b] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks." In *FPGA*, pp. 161–170. ACM, 2015.

[ZLS15c] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. "Optimizing fpga-based accelerator design for deep convolutional neural networks." In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170, 2015.

[ZNL18] Ruizhe Zhao, Xinyu Niu, and Wayne Luk. "Automatic Optimising CNN with Depthwise Separable Convolution on FPGA: (Abstact Only)." In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 285–285, 2018.

[ZP20] Hanqing Zeng and Viktor Prasanna. "GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms." In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 255–265, 2020.

[ZSF16] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks." In *2016 IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–8, 2016.

[ZVS18] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. "Learning transferable architectures for scalable image recognition." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.

[ZWZ18a] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. "Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas." In *Proceedings of the International Conference on Computer-Aided Design*, p. 56. ACM, 2018.

[ZWZ18b] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. "DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs." In *Proceedings of the International Conference on Computer-Aided Design*, p. 56. ACM, 2018.

[ZZL15] Xiang Zhang, Junbo Zhao, and Yann LeCun. "Character-level convolutional networks for text classification." *arXiv preprint arXiv:1509.01626*, 2015.

[ZZS19a] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. "Graphsaint: Graph sampling based inductive learning method." *arXiv preprint arXiv:1907.04931*, 2019.

[ZZS19b] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. "Graphsaint: Graph sampling based inductive learning method." *arXiv preprint arXiv:1907.04931*, 2019.