CO-DESIGNING MODEL COMPRESSION ALGORITHMS AND HARDWARE ACCELERATORS FOR EFFICIENT DEEP LEARNING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Ritchie Zhao

May 2020

CO-DESIGNING MODEL COMPRESSION ALGORITHMS AND HARDWARE ACCELERATORS FOR EFFICIENT DEEP LEARNING

Ritchie Zhao, Ph.D. Student

Cornell University, May 2020

Over the past decade, machine learning (ML) with deep neural networks (DNNs) has become extremely successful in a variety of application domains including computer vision, natural language processing, and game AI. DNNs are now a primary topic of academic research among computer scientists, and a key component of commercial technologies such as web search, recommendation systems, and self-driving vehicles. However, factors such as the growing complexity of DNN models, the diminished benefits of technology scaling, and the proliferation of resource-constrained edge devices are driving a demand for higher DNN performance and energy efficiency. Consequently, neural network training and inference have begun to shift from commodity general-purpose processors (e.g., CPUs and GPUs) to custom-built hardware accelerators (e.g., FPGAs and ASICs). In line with this trend, there has been extensive research on specialized algorithms and architectures for dedicated DNN processors. Furthermore, the rapid pace of innovation in DNN algorithm space is mismatched with the time-consuming process of hardware implementation. This has generated increased interest in novel design methodologies and tools which can reduce the human effort and turn-around time of hardware design.

This thesis studies how low-precision quantization and structured matrices can improve the performance and energy efficiency of DNNs running on specialized accelerators. We co-design both the DNN compression algorithms and the accelerator architectures, enabling us to evaluate the impact of our ideas on real hardware. In the process, we examine the use of high-level synthesis tools in reducing the hardware design effort. This thesis represents a cross-domain research effort at efficient deep learning. First, we propose specialized architectures for accelerating binarized neural networks on FPGA. Second, we study novel high-level synthesis techniques to reduce

the manual effort in FPGA accelerator design. Third, we show a fundamental link between group convolutions and circulant matrices — two previously disparate lines of research in DNN compression. Using this insight we propose HadaNet, an alternative to circulant compression which achieve identical accuracy with asymptotically fewer multiplications. Fourth, we present outlier channel splitting, a technique to improve DNN weight quantization by removing outliers from the weight distribution without arduous retraining. Finally, we show preliminary results on overwrite quantization, a technique which address outliers in DNN activation quantization using extremely lightweight architectural extensions to a spatial accelerator template.

BIOGRAPHICAL SKETCH

Ritchie Zhao was born in the city of Wuhan, China on March 16, 1991. His family would move a number of times during his childhood, from China to Singapore, then Ottawa, and finally settling in Toronto. Ritchie's parents cultivated his interest in math and sciences from an early age, and he was fortunate to have met a number of friends and mentors in high school who further inspired his natural curiosity.

Ritchie received his Bachelors in Electrical and Computer Engineering from University of Toronto in 2014, and afterwards went on to do a PhD in the School of Electrical and Computer Engineering at Cornell University under Professor Zhiru Zhang. He started his PhD researching algorithms for FPGA-targeted high-level synthesis (HLS). In his third year, his focus switched to hardware specialization for deep neural networks and redesigning DNN algorithms to better suit emerging hardware architectures.

Ritchie has also had extensive experience working in industry. During undergrad, he interned at Altera (now part of Intel) in the timing modeling team, as well as at IBM working on compilers for heterogeneous systems. During his PhD, he interned twice at Microsoft Research in Redmond, working with Daniel Lo and Eric Chung on neural network quantization for the Brainwave project.

This document is dedicated to my parents, mentors, and all who believed in me.

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support, encouragement, and advice from a great number of people here at Cornell and otherwise. I would like to thank my family, teachers, mentors, friends, and all who believed in me for helping me through my PhD both professionally and personally.

First and foremost, I would like to thank my advisor Prof. Zhiru Zhang for being a great mentor, motivator, and role model throughout my time at Cornell University. Zhiru supported me in all of my projects, put tremendous time into helping my publications, and gave me freedom to pursue my own ideas in my graduate career. Zhiru's expertise and guidance was instrumental in helping me switch course from pure EDA research to the emerging field of machine learning systems; his rigor and attention to detail also left a huge influence on my own work. Thanks to Zhiru, I had a extremely fruitful PhD experience at Cornell and was exposed to many opportunities in both academia and industry.

I would also like to thank the other members of my graduate thesis committee, Prof. Christopher Batten, Prof. Christoph Studer, and Prof. Christopher De Sa. Chris Batten was responsible for teaching me almost everything I know about computer architecture, and provided both encouragement and advice in both our joint projects and in the rest of my thesis work. Chris was also pivotal in helping the CSL move across three facilities over my time at Cornell, and was responsible for many improvements to student life at the CSL lab. Christoph Studer not only taught me a tremendous amount about VLSI, but also helped me with many of the mathematical details in the UGConv project. I am grateful for his invaluable advice on my research direction at a time when I was most unsure of the course of my thesis. Last and not least, Chris De Sa taught me a tremendous amount on both the algorithm and system side of machine learning. I am indebted to him for kick-starting my ML projects, showing me many of the finer points of ML theory, and instilling the confidence in me to publish in an unfamiliar venue.

Furthermore, I would like to thank my many friends and colleagues at the CSL. I am grateful to Steve Dai, Gai Liu, and Dr. Mingxing Tan for being my first collaborators and companions, as

well as the other members of the Zhang Research Group for the discussions, projects, and fun we shared. I am also incredibly thankful to Dr. Ji Kim, Dr. Charles Jeon, Ecenur Ustun, Khalid Al-Hawaj, Ryan O'Hern, Dr. Mark Buckler, Philip Bedoukian for their friendship and encouragement over the years.

Last but certainly not least, I am grateful to my parents, Qing Xiang and Qing Zhao. While they were not physically with me through most of my PhD, they watched over, encouraged, and inspired me in their own way. Their dedication was what made me who I am, and their wisdom continues to guide me today.

This thesis was supported in part by DARPA Award HR0011-16-C-0037, DARPA Young Faculty Award D15AP00096, NSF Awards #1337240, #1453378, #1512937, the Semiconductor Research Corporation (SRC), and research gifts from Xilinx, Inc and NVIDIA Corporation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

BNN	binarized neural network
CNN	convolutional neural network
DNN	deep neural network
ASIC	application-specific integrated circuit
BRAM	block random-access memory
CDFG	control data flow graph
CPU	Central processing unit
DFG	data flow graph
DSL	domain-specific language
EDA	electronic design automation
FF	flip-flop
FPGA	field-programmable gate array
GPU	graphical processing unit
HaD	Hadamard-diagonalizable
HDL	hardware description language
HLS	high-level synthesis
II	initiation interval
ILP	integer linear programming
LUT	look-up table
MAC	multiply-accumulate
MILP	mixed-integer linear programming
ML	machine learning
NLP	natural language processing
OCS	outlier channel splitting
QoR	quality of results
RTL	register-transfer level
SDC	system of integer difference constraints

CHAPTER 1

INTRODUCTION

1.1 Hardware Specialization for Deep Neural Networks

Over the past decade, machine learning (ML) with deep neural networks (DNNs) has evolved from a niche academic discipline to become an important class of algorithms widely used across many different research domains and practical technologies. Although neural networks have been an integral part of ML research since the 1960's, for much of history they were outclassed by simpler techniques such as support vector machines and decision trees. Neural networks would experience a massive resurgence in the early 2010s owing to three factors: (1) advancements in network algorithms and design; (2) the development of massively parallel compute devices, especially general-purpose GPUs; and (3) the availability of large quantities of training data on the internet. All three factors would eventually converge during the development of AlexNet [KSH12], a convolutional neural network (CNN) which won the 2012 ImageNet large scale classification challenge [DDS+09] by a wide margin. This victory is frequently cited as the start of the modern "deep learning revolution". Today, deep neural nets dominate many computer vision tasks such as image recognition [KH09, DDS+09], object detection [EVGW+10, XZY+18], and semantic segmentation [COR+16, LMB+14]. Deep learning is also state-of-the-art in natural language processing (NLP) tasks such as text translation [BCB14, SVL14, LPM15, WSC+16] and question answering [VL15, RZLL16, LYBP16]. Beyond supervised learning, DNNs have led to the emergence of deep generative models like autoencoders [Doe16] and adversarial networks [GPAM+14, RMC15] being applied to tasks such as image in-painting [XXC12, YCYL+17], style transfer [GEB16, JAFF16], super-resolution [JAFF16, LTH+17], and content creation [OOS17, ZXL+17]. Deep learning has also proven remarkably transferable. Advances in network architectures, training algorithms, and feature representations generalize well to different application domains. Recent years have seen the successful expansion of DNNs into scientific disciplines where machine learning had no traditional foothold, with

Table 1.1: Time, Energy, and Monetary cost of training contemporary NLP models — figure is reproduced from [SGM19]. *kWh is calculated by the authors using a power usage effectiveness (PUE) penalty of 1.58 that is standard for US datacenters.

Model	Hardware	Power	Hours	kWh*	Cloud Server Cost
Transformer (base)	P100 x8	1416	12	27	$41–$140
Transformer (big)	P100 x8	1515	84	201	$289–$981
ELMo	P100 x3	518	336	275	$433–$1472
BERT (base)	V100 x64	12041	79	1507	$3751–$12,571
BERT (base)	TPUv2 x16	-	96	-	$2074–$6912
NAS	P100 x8	1515	274120	656347	$942,973–$3,201,722
NAS	TPUv2 x1	-	32623	-	$44,055–$146,848

examples including precision medicine, integrated circuit design, wireless communication, and financial analysis. Last and not least, deep learning has been heavily adopted in a number of real-world technologies include web search [SHG+14, WWY15, CAS16, CKH+16], language translation [SVL14, WSC+16], autonomous 3D navigation [KC15, ZMK+17, BDTD+16], robotic manipulation [LFDA16, MKS+15, LHP+15], and game AI [SHM+16, SSS+17, VBC+19]. These examples serve to demonstrate both the enormous practical capability and rapidly expanding application of deep neural networks in both academia and industry.

Growing DNN compute requirements – Although neural networks have advanced the quality of results in many domains, they have a critical drawback: training and serving a modern DNN model requires enormous amounts of computational power and electrical energy. Table 1.1 reproduced from Strubell et al. [SGM19] provides examples of the time, energy, and monetary cost of training a state-of-the-art NLP model. The same work also discusses the environmental impact of DNN training, reporting that training the state-of-the-art language model BERT [DCLT18] releases the equivalent quantity of CO_2 as a trans-American flight. It is not merely the case that deep neural networks are inherently computationally expensive compared to prior techniques; their depth and complexity has only increased over time. Figure 1.1 shows the exponential increase in DNN training cost since AlexNet. Note that during the timespan depicted, DNN compute and parameter efficiency have actually improved dramatically. However, greater algorithmic efficiency

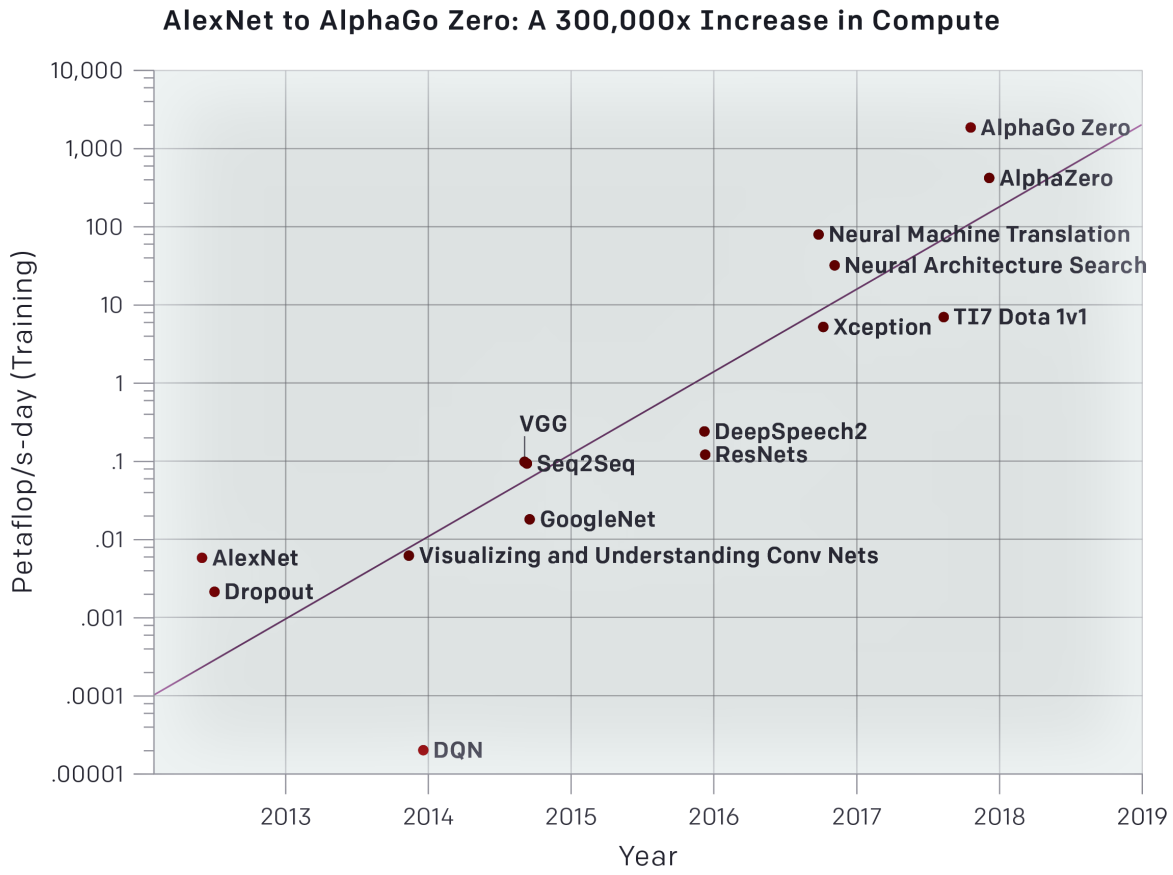


Figure 1.1: Growing compute cost of DNN training over time — training cost is measured in Peta-floating-point operations per second (FLOPS) $\times$ days of training, taken from an OpenAI blog post [AH18].

also leads to greater scalability. Early networks like AlexNet [KSH12] and VGG [SZ15] have difficulty scaling beyond 10 layers while contemporary architectures such as ResNet [HZRS15] and DenseNet [HLWvdM17] can scale to over 1000 layers and still see accuracy improvement. Training a bigger and more powerful network is usually a foolproof way to improve results, and consequently these algorithmic innovations have only encouraged machine learning practitioners to deploy ever larger models on bigger datasets. For example, the latest advancements in language modeling features unsupervised *pre-training* on huge amounts of unlabeled web-scraped text [DCLT18, RWC$^+$19], requiring days to weeks even on 64 GPUs.

Breakdown in technology scaling – Since the 1970's, integrated circuits saw periodic increases to the number of transistors per area. This phenomenon, named Moore's Law, was ac-

companied by a design ruled called Dennard Scaling [DGR+74] which described how to scale transistor size as to reduce the switching delay while keeping power density constant. These two scaling laws together provided substantial improvements to processor clock frequency and storage density each year while the manufacturing cost and power draw of a chip remained constant. However, by the mid 2000s, Dennard scaling had slowed considerably. Transistors stopped becoming cheaper, and chip manufacturing costs have steadily increased at each new technology node since 2012. Issues with leakage current prevented the scaling of power, and clock frequency was forced to level off to limit thermal dissipation. Unfortunately, the slowdown of scaling occurred around the same time deep learning began to gain traction. While transistors are still shrinking (at a slower pace than before), technology scaling today cannot provide the performance and efficiency improvements necessary to keep pace with the growing demands of DNN workloads.

Specialized hardware DNNs acceleration – In response to the previous two trends, computer architecture has moved towards specialized hardware for deep neural networks. Hardware acceleration for DNNs date back to AlexNet, which was first to utilize multiple GPUs to perform training in a reasonable timespan. In the years following, GPUs would supplant general-purpose CPUs as the go-to platform for DNN training [CHW+13, CL14]. General-purpose GPUs can efficiently parallelize linear algebra operations such as matrix multiplies, which make up the bulk of DNN workloads in the form of convolutional and fully-connected layers. Over the years, GPUs have become ubiquitous in deep learning and enjoy widespread support in popular ML experimentation frameworks. However, recent years see both academia and industry shift towards dedicated hardware for neural networks. DNN accelerators have become a major topic in the computer architecture research community, with works such as the DianNao variants [CDS+14, CLL+14], Eyeriss [CKES17], Cambricon [ZDZ+16], EIE [HLM+16], CirCNN [DLW+17, WDL+18], and OLAccel [PKY18]. Many of these architectures specifically target the complex, compute-heavy convolutional layers found in CNNs. In the commercial space, major cloud providers have began to offer ML services such as image recognition, automated speech, and targeted advertising. These cloud companies were the first to develop mass production systems dedicated solely to DNN

execution, with examples such as Google's TPU [JYP+17] and Microsoft's Brainwave [CFO+18].

A similar trend has also emerged in the mobile and embedded space. Deep neural networks have become a fundamental algorithm in computer vision and language analysis. These two technologies are in turn critically important to a variety of edge devices including mobile phones, camera drones, autonomous vehicles, and IoT sensors. Unlike in a datacenter, edge devices are limited by their small physical form factor, short battery life, and basic means of heat dissipation. These factors impose hard constraints on the compute and storage available for DNN execution (typically around 500 Mega-floating-point operations (MFLOPs) and 10s of Megabytes [HZC+17, ZZLS17]). Recently, the largest mobile handset vendors have also begun to incorporate dedicated neural processors into their mobile SoCs; examples include Apple's A12 (iPhone X), Samsung's Exynos 9820 (select Galaxy S10s), and Huawei's Kirin 970 (Honor 10, P20, others). Embedded DNN accelerators have also been extensively studied in academia, with particular focus on FPGAs (e.g., Zhang'15 [ZLS+15], AngelEye [QWY+16], FINN [UFG+17], ESE [HKM+17], C-LSTM [WLD+18], SkyNet [ZLH+19, XZY+18]) and ASICs (e.g., Origami [CGM+15], BinarEye [MBY+18], YodaNN [ACRB16]).

Algorithms for specialized DNN hardware – Throughout most of the 2010's deep learning revolution, ML researchers were constrained by the computer hardware they had access to. Any novel neural network algorithm must be well-suited to the best commodity training device (i.e., general-purpose GPU) because otherwise, it will likely be outperformed by an existing network which fits more weights into GPU memory and trains on more data. In this manner, the algorithm must be specialized for the available hardware. AlexNet once again provides an instructive example. The network was tailor-built to train on a two-GPU system (a novelty at the time) and replaced sigmoid activation functions with the simpler rectified linear unit (ReLU) to speed up GPU training [KSH12]. These GPU-specific features were a large part of AlexNet's success. Another important case of algorithm specialization for hardware is parallel stochastic gradient descent [ZWLS10, RRWN11], which enables DNN training to scale across multiple machines.

The emergence of dedicated DNN accelerators means that DNN algorithms are less constrained

by existing standards and instruction set architectures (ISAs). This has in turn driven research into novel DNN compression algorithms which can take full advantage of emerging hardware. One popular technique is quantization, which seeks to reduce the complexity of the numerical format used in DNN computations. Typically, 32-bit float is used for all values in a DNN. Research efforts in DNN quantization can be divided into two broad camps: inference-only quantization (which only needs to quantize the forward pass) and training quantization (which must quantize both forward and backwards passes). For inference-only, a number of works have demonstrated the use of 8-bit fixed-point [WLCS18, JKC+18, HMD16], with some going to even fewer bits [ZWN+16]. Early efforts in this space made use of fine-tuning (further training the quantized model for a few more epochs) to recover some of the lost accuracy. More recent work has established the feasibility of data-free quantization: quantizing a pre-trained model without additional training [Mig17, ZHD+19a, NvBBW19, MFAG19]. Training quantization is significantly more challenging. Gradient accumulation requires a larger dynamic range, and can furthermore introduce issues of numerical instability. While some models can be trained in fixed-point, this is not a widely accepted practice. Some number formats for training include float16 [MNA+17, JSH+18], bfloat16 [KMM+19] (first appeared in Google's DistBelief framework [DCM+12]), and some narrower non-standard floats [TYW+19, SCC+19]. Even more exotic formats have been proposed for DNNs such as binary and ternary networks for inference [CHS+16, RORF16, LL16], and the posit format for training [CLK+19, LLW+19]. Quantization also enjoys broad support among industry in both DNN frameworks and accelerators (e.g., float16 in newer NVIDIA GPUs [Mig17] and int8 in Google's TPU [JYP+17]).

There are also major classes of DNN algorithms that were proposed with hardware in mind, but have yet to gain significant traction among computer architects. One important example is sparse network pruning. Pruning takes a pre-trained model and removes weights or activations that are deemed unimportant (e.g., based on magnitude [HPTD15, HPN+17]). The pruned model can be fined-tuned for a small number of additional epochs to recover some of the accuracy lost during pruning process [HPN+17]. Use these techniques, VGG-16 [SZ15] has been compressed

to just 11% of its original size without impacting accuracy [HPTD15]. Although sparse pruning is heavily researched in the ML community and achieves impressive memory/compute savings, it is challenging to implement in real hardware. A pruned DNN model requires a sparse matrix representation (e.g., compressed sparse row), which stores non-zero values and their locations in the form of indices. Sparse representations incur additional storage overhead and memory indirection, which directly translate to performance and area penalties in a hardware accelerator. The accelerator must also implement sparse linear algebra operations which are only efficient when the matrices are highly sparse (i.e., $> 90\%$ zeros); it cannot efficiently target dense models. Sparse DNN accelerators have been recieved significant attention in academia [HLM$^+$16, HKM$^+$17, ZDZ$^+$16], but the prevailing industry trend is for hardware to focus on dense computation.

Methodologies for DNN hardware specialization – The proliferation of specialized DNN accelerators present serious challenges to the existing hardware design methodology. Currently, neural networks are typically trained in a pure ML framework such as Caffe [JSD$^+$14], TensorFlow [ea15], or PyTorch [PGC$^+$17], which enjoys ease of programming, GPU support, and optimized libraries such as cuDNN [CWV$^+$14]. Meanwhile, hardware design is done in register-transfer level (RTL) with the help of functional, cycle-accurate, and timing simulators. Two major problems exist with this approach. First, there is a substantial gap in the level of abstraction between the high-level ML frameworks and the low-level RTL tools. This greatly complicates the process of preparing and optimizing an ML model for hardware, and places barriers to any sort of model-architecture co-design. Second, the pace of innovation in ML space is extremely fast (the number of ML-related arXiv papers *doubles* every 2 years [DPY18]). Meanwhile, the design turn-around time for an accelerator in RTL can be over a year. By the time a DNN accelerator is taped-out, it may have been sidelined or even superseded by better algorithms. These issues already manifest themselves as a deep division between ML and architecture research communities. ML research almost never performs evaluation on a custom-built hardware, and architecture publications often focus on out-of-date benchmarks (e.g., AlexNet and VGG-16). Various approaches for addressing this productivity gap have been advanced, including high-level syn-

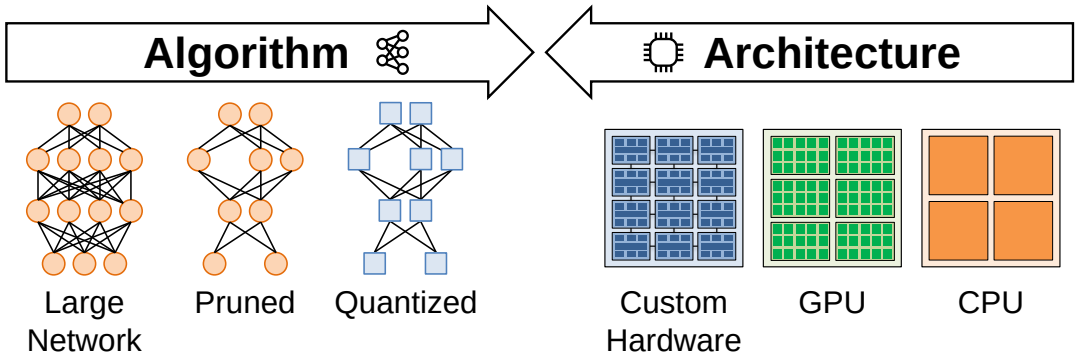


Figure 1.2: A co-design methodology involves specializing both algorithm and architecture with each other in mind for mutual benefit.

thesis [ZLS+15, ZSZ+17, SCD+16], domain-specific languages, and specialized design generators [WXH+16].

1.2 Thesis Overview

At a high level, this thesis explores deep neural network acceleration using algorithm and architecture co-design. Figure 1.2 illustrates the co-design approach, in which both algorithm and hardware are specialized with each other in mind for mutual benefit. We make contributions to multiple levels along the computer architecture stack and combines these innovations to demonstrate how cross-stack synergies can help address the computational challenges in DNN inference and training. We also make contributions to design automation tools to reduce the human effort of hardware specialization; such tools are key enablers for a tightly-coupled co-design methodology.

More specifically, this thesis examines how quantization and structured matrices can be used to specialize neural networks for efficient execution on custom hardware architectures. It also studies how high-level synthesis tools can reduce the effort of designing such hardware on FPGAs and ASICs. Each chapter of the thesis describes a research project which has been published to a peer reviewed conference. Figure 1.3 illustrates the contribution of each chapter to different parts of the architecture stack (machine learning algorithm, electronic design methodology, and hardware architecture), emphasizing the cross-domain nature of our work.

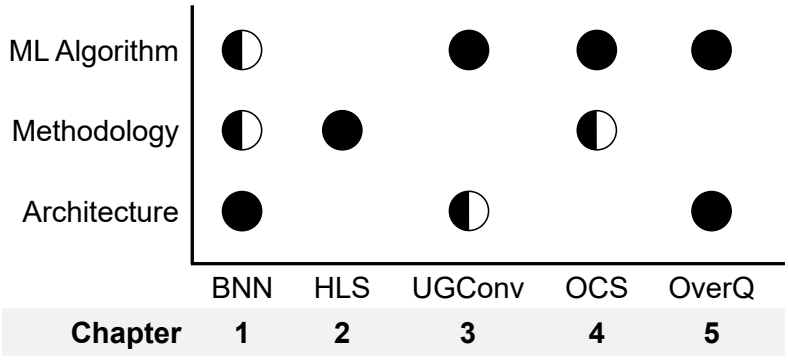


Figure 1.3: Thesis chapters organized with respect to the architectural stack — solid circle indicates significant technical contribution, while half circle indicates light contribution.

A detailed overview of the thesis is as follows:

- Chapter 2 presents an FPGA accelerator for binarized neural networks (BNNs) using a high-level synthesis (HLS) design methodology. The chapter proposes novel architectural elements for hardware BNN processing, and demonstrates how BNNs map more efficiently to FPGAs compared to conventional neural nets. The use of HLS illustrates the potential of such a methodology in rapidly constructing and testing ML accelerators. This work was published at FPGA'17 [ZSZ+17].
- Chapter 3 presents a cross-layer, mapping-aware approach to HLS pipeline scheduling which improves the synthesized hardware for logic operations. This technique has potential applicability in very low-precision DNN accelerator design. This work was published at DAC'15 [ZTDZ15].
- Chapter 4 presents unitary group convolutions, a conceptual framework which unifies group convolutions and structured matrices. Prior to this work, these were two disparate topics in efficient DNN layer design. Leveraging insights from this framework, the chapter proposes using the Hadamard transform for DNN compression. Hadamard outperforms prior art in Fourier and shuffle transforms under certain conditions. This work was published at CVPR'19 [ZHD+19b].
- Chapter 5 presents outlier channel splitting (OCS), a technique which improves the accuracy

of DNN quantization by removing outliers from the weight distribution. OCS is one of the few methods that target the problem of data-free quantization (i.e. quantizing a DNN without additional training epochs). The chapter also presents the an evaluation of various data-free clipping techniques in literature, showing that OCS outperforms clipping at lower precisions. This work was published at ICML'19 [ZHD+19a].

- Chapter 6 presents preliminary results on overwrite quantization (OverQ), a technique which addresses activation outliers efficiently through lightweight architectural changes to a commonly used spatial accelerator template. This research is work in progress, with a pre-print manuscript available on arXiv [ZDSZ19].

1.3 Collaborations, Funding, and Previous Publications

This thesis would not be possible without the contributions of colleagues within the Zhang Research Group and Batten Research Group in the Computer Systems Lab at Cornell University, as well as collaborators from Peking University, University of California Los Angeles (UCLA), and University of California San Diego (UCSD). My advisor and committee chair, Professor Zhiru Zhang, provided unparalleled guidance and assistance on all five projects presented in this thesis. The BNN project in Chapter 2 was done in collaboration with Jeng-Hau Lin and Prof. Rajesh Gupta at UCSD, as well as Tianwei Xing and Prof. Mani Srivasatava at UCLA. For the BNN project, Weinan Song and Wentao Zhang from Peking University built and collected latency data on the GPU baselines. The BNN design on FPGA was later ported to an ASIC named Celerity as part of the DARPA CRAFT program. The ASIC design was done in collaboration with Prof. Christopher Batten at Cornell. Khalid Al-Hawaj, Christopher Torng, and Dr. Steve Dai did nearly all of the ASIC HLS implementation, gate-level optimization and testing, and tape-out. The HLS project in Chapter 3 was an extension of a mapping-aware scheduling paper headlined by Dr. Mingxing Tan and Dr. Steve Dai [TDGZ15], and features code and benchmarks adapted from their work. The initial idea for UGConv emerged from discussions with Dr. Charles Jeon. Both the UGConv

project in Chapter 4 and the OCS project in Chapter 5 was done in collaboration with Prof. Chris De Sa at Cornell University, and with assistance from Yuwei Hu and Jordan Dotzel. UGConv also references technical details and code from Prof. Yanzhi Wang at Northeastern University and Prof. Bo Yuan at Rutgers University. Yuwei Hu wrote the MXNet code for ImageNet experiments, and Jordan Dotzel ran exploratory studies on different transforms. For OCS, Yuwei Hu built the first version of the code using PyTorch-Distiller and performed the RNN experiments. Jordan Dotzel devised and implemented the code for quantization-aware splitting. The OverQ project features data and insights gathered by Vidya Ramesh.

Many of my projects owe a great deal to discussions with Prof. Christoph Studer at Cornell, as well as with Daniel Lo and Eric Chung at the Microsoft Azure Advanced Intelligent Architectures group.

This thesis was supported in part by DARPA Award HR0011-16-C-0037, DARPA Young Faculty Award D15AP00096, NSF Awards #1337240, #1453378, #1512937, the Semiconductor Research Corporation (SRC), and research gifts from Xilinx, Inc and NVIDIA Corporation. My complete list of publications during my PhD in reverse chronological order is as follows:

1. Austin Rovinski, Chun Zhao, Khalid Al-Hawaj, Paul Gao, Shaolin Xie, Christopher Torng, Scott Davidson, Aporva Amarnath, Luis Vega, Bandhav Veluri, Anuj Rao, Tutu Ajayi, Julian Puscar, Steve Dai, Ritchie Zhao, Dustin Richmond, Zhiru Zhang, Ian Galton, Christopher Batten, Michael B Taylor, Ronald G Dreslinski, "A 1.4 GHz 695 Giga Risc-V Inst/s 496-Core Manycore Processor With Mesh On-Chip Network and an All-Digital Synthesized PLL in 16nm CMOS", *Symposium on VLSI Circuits (VLSI)*, Jun 2019
2. Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Christopher De Sa, Zhiru Zhang, "Improving Neural Network Quantization without Retraining using Outlier Channel Splitting", *International Conference on Machine Learning (ICML)*, Jun 2019
3. Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Christopher De Sa, Zhiru Zhang, "Building Efficient Deep Neural Networks with Unitary Group Convolutions", *Computer Vision and Pattern Recognition (CVPR)*, Jun 2018

4. Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawaj, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, Aporva Amarnath, Bandhav Veluri, Paul Gao, Anuj Rao, Gai Liu, Rajesh K. Gupta, Zhiru Zhang, Ronald G. Dreslinski, Christopher Batten, Michael Bedford Taylor, "The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips", *IEEE Micro (Vol 38, Issue 2)*, Apr 2018

5. Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengil, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Christian Boehn, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Tamas Juhasz, Ratna Kumar Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Steve Reinhardt, Adam Sapek, Raja Seera, Balaji Sridharan, Lisa Woods, Phillip Yi-Xiao, Ritchie Zhao, Doug Burger, "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave", *IEEE Micro (Vol 38, Issue 2)*, Apr 2018

6. Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, Zhiru Zhang. "Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs", *Int'l Symp. On Field-Programmable Gate Arrays (FPGA)*, Feb 2018

7. Jeng-Hau Lin, Tianwei Xing, Ritchie Zhao, Zhiru Zhang, Mani Srivastava, Zhuowen Tu, Rajesh K. Gupta, "Binarized Convolutional Neural Networks with Separable Filters for Efficient Hardware Acceleration", *Computer Vision and Pattern Recognition Workshops (CVPRW)*, Jul 2017

8. Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, Zhiru Zhang, "Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs", *Int'l Symp. On Field-Programmable Gate Arrays (FPGA)*, Feb 2017

9. Chang Xu, Gai Liu, Ritchie Zhao, Stephen Yang, Guojie Luo, Zhiru Zhang, "A Paral-

lel Bandit-Based Approach for Autotuning FPGA Compilation", *Int'l Symp. On Field-Programmable Gate Arrays (FPGA)*, Feb 2017

10. Ritchie Zhao, Gai Liu, Shreesha Srinath, Christopher Batten, Zhiru Zhang, "Improving High-Level Synthesis with Decoupled Data Structure Optimization", *Design Automation Conference (DAC)*, Jun 2016

11. Mingxing Tan, Gai Liu, Ritchie Zhao, Steve Dai, Zhiru Zhang, "ElasticFlow: A Complexity-Effective Approach for Pipelining Irregular Loop Nests", *Int'l Conf. on Computer Aided Design (ICCAD)*, Nov 2015

12. Ritchie Zhao, Mingxing Tan, Steve Dai, Zhiru Zhang, "Area-Efficient Pipelining for FPGA-Targeted High-Level Synthesis", *Design Automation Conference (DAC)*, Jun 2015

CHAPTER 2

HARDWARE ACCELERATION FOR BINARIZED NEURAL NETWORKS

2.1 Preliminaries

This chapter presents our work on hardware specialization for binarized neural networks (BNNs), an extreme form of low-precision neural network where weights and activations are restricted to $+1$ or -1. As discussed in the thesis introduction, one major challenge to the widespread deployment of CNNs is their significant demands for computation and storage capacity. The VGG-19 network, for instance, contains over 140 million floating-point (FP) parameters and performs over 15 billion FP operations to classify one image [SZ15]. A number of efforts in the academic community has demonstrated the potential of FPGA-based CNN accelerators [ZLS+15, QWY+16] as well as tools for generating such accelerators automatically [SCD+16, WXH+16]. Recent work by Microsoft has explored DNN acceleration on FPGAs at datacenter scale [ORK+15, CFO+18].

One major advantage of FPGAs (and ASICs) as compared to conventional CPUs and GPUs is the ability to compute using simpler, reduced-precision number representations. Where as DNNs on CPUs and GPUs use 32 or 16-bit float (with rare cases of int16 or int8), dedicated hardware such as FPGAs can can compute at arbitrary precisions such as int4, int2, or even lower. Figure 2.1 shows the throughput of the Brainwave FPGA-based DNN accelerator across different numerical precisions, over three modern FPGA devices. Note the logarithmic y-axis, and the use of non-standard numerical representations custom-tailored to DNN workloads. DNN quantization is critical for extracting DNN performance from specialized hardware. It is an extremely active area of research [ZWN+16, SFRO17, WLCS18, JKC+18, CWV+18, BNHS18] and sees deployment in many commercial systems such as Google's TPU [JYP+17], NVIDIA's TensorRT [Mig17], and Microsoft's Brainwave [CFO+18].

Recently, research in neural network algorithms have demonstrated the potential of *binarized* convolutional neural networks, CNNs where every weight and activation is represented as just $+1$ or -1. While simple to understand, training a BNN is mathematically non-trivial as conventional

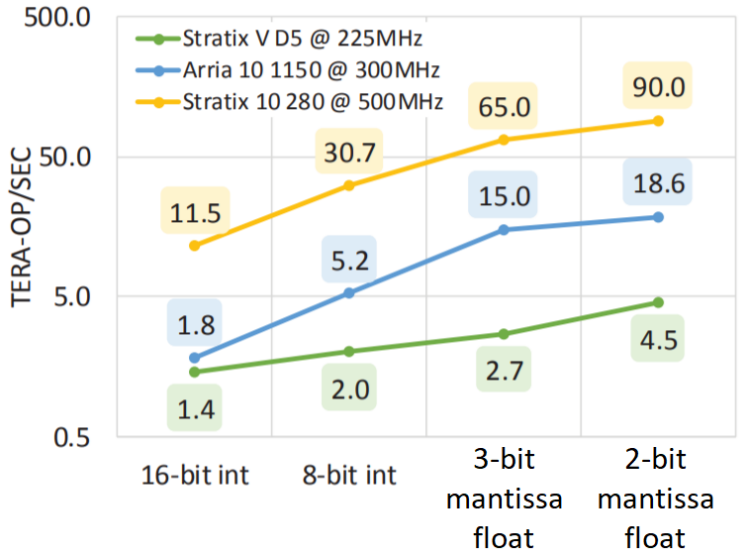


Figure 2.1: Throughput of the Brainwave FPGA-based DNN accelerator – throughput is computed by assuming full saturation of all processing engines realized on the device. *N*-bit mantissa float is a specialized numerical representation for DNN workloads. Taken from [CFO+18].

backpropagation stochastic gradient descent (SGD) fails at this level of quantization. Two key issues emerge: (1) the `sign` function used to binarize values has a derivative of zero almost everywhere; (2) even if an approximate derivative is used, each individual gradient update is generally too small to flip the sign of a weight and therefore no training progress can be made. A significant breakthrough was made on this front by Corbariaux et al. in 2016. The solution, presented in [CBD15] and refined in [CHS+16], is to treat binarization as a noise process which is only applied on the forward pass. The weights being updated in each step remains in floating-point precision, enabling SGD to run as normal. Courbariaux et al.'s network architecture (which we call the BinaryNet model) achieved near state-of-the-art results on CIFAR-10 and SVHN datasets at the time of publication for this work, but could not match the accuracy of high-precision networks on the larger ImageNet dataset. Since then, there enormous advancement in both BNN (e.g., XNOR-net [RORF16], Bi-Real net [LWL+18]) as well as related low-precision networks (e.g., binary-weight networks [LZP17], ternary nets [LL16]) have greatly closed this accuracy gap.

BNNs may be one key to efficient deep learning on FPAGs. The dot product between two binary vectors can be computed using bitwise `xors` and a popcount, two operations which can be

implemented very efficiently on the FPGA look-up table (LUT) fabric. A customized BNN accelerator on FPGA can offer significant improvements in energy efficiency and power dissipation compared to conventional neural nets. These factors may be critical in enabling the increased use of binary networks in low-power settings such as unmanned drones or embedded computing. Though Courbariaux et al. made seminal contributions to the construction and training of BNNs, they only presented back-of-the-envelope estimations of run time and energy savings on CPU/GPU, and did not consider other platforms. Furthermore, the algorithmic differences between BNN and conventional CNN necessitates substantial differences in accelerator architecture. In this chapter, we present the design and implementation of a specialized accelerator for BNN inference on embedded FPGAs. In addition, the accelerator was migrated to a 16nm ASIC as part a DARPA CRAFT tapeout named Celerity [A$^+$18, D$^+$18]. This work was among the first to explore and evaluate specialized architectures for BNN acceleration and show empirical results at the silicon level.

One drawback to using dedicated hardware as compared to commodity CPUs and GPUs is the lack of compatibility with deep learning frameworks such as Caffe [JSD$^+$14], Theano [The16], or TensorFlow [ea15]. These frameworks allow users to make use of the latest models or to train a custom network with little engineering effort, and as a result, there is a sizable gap in design effort in creating an ML application for GPU versus FPGA platforms. This gap is especially distressing given the rate of algorithmic innovation in deep learning. An FPGA-based CNN accelerator (or CNN design compiler) is unlikely to support the most up-to-date models, putting them at a severe competitive disadvantage. To partially address this issue, we designed our BNN accelerator using high-level synthesis (HLS) from C++, and open-sourced our code to the research community. HLS tools such as Xilinx Vivado HLS [CLN$^+$11], LegUp [CCA$^+$13], and Intel FPGA SDK for OpenCL [CAD$^+$12] allows users to specify the design at a higher abstraction than conventional register-transfer level (RTL) design, and some offer further automation features for generating the hardware-software interface and on-chip memory network. In the context of deep learning, these tools have the potential to critically reduce time-to-market on new accelerator designs and thus reduce the aforementioned productivity gap. While it nowhere near closes the gap on a full-fledged

ML framework, the use of HLS represents one step to making hardware accessible to algorithm designers and data scientists. The next present improvements to HLS scheduling algorithms which further reduce human effort and hardware know-how required to create a high-quality design.

2.1.1 Building and Training a Binarized CNN

In this section we describe algorithmic changes necessary to build and train a CNN which uses binarized weights and activations. A typical CNN consists of a pipeline of connected *layers*. Each layer takes as input a set of *feature maps* (*fmaps*), performs some computation on them, and produces a new set of fmaps to be fed into the next layer. The input fmaps of the first layer are the channels of the input image. Common layer types found in CNNs include convolutional (*conv*) layers, dense or fully-connected (*FC*) layers, and pooling (*pool*) layers.

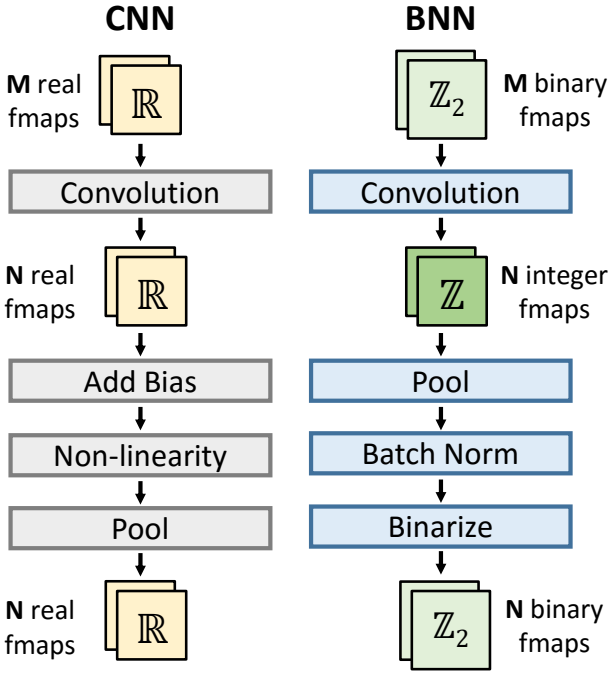


Figure 2.2: Comparison of CNNs and BNNs — Left: the order of operations in a CNN for a conv and pool layer. Right: the (modified) order of operations in the BinaryNet BNN [CHS+16]. Pooling is performed early and a batch normalization precedes the binarization to minimize information loss. Biases have been removed from the BNN.

As stated previously, a BNN is essentially a neural network whose weights and activations are

binarized to $+1$ or -1. Figure 2.2 illustrates the typical stack of layers in (1) a conventional CNN and (2) a BNN. In the CNN, all weights and calculations use full-precision floating-point. In the BNN, the intermediate data go from binary to integer after convolution until it is binarized again. Biases have been removed, and pooling is always performed on the integer intermediates.

The BNN introduces the batch normalization or (*batch norm*) layer, which perform a linear transformation (scale and shift) on the input data to keep the input distribution (nominally a Gaussian) close to zero mean and unit variance. Batch norm layers have been shown to improve training speed and accuracy in CNNs [IS15]. In BNNs however, they gain additional importance as binarizing a centered and properly-scaled distribution leads to lower quantization error compared to an arbitrary distribution. The batch norm transformation is given below,

$$y = \frac{x-\mu}{\sqrt{\sigma^2+\varepsilon}}\gamma+\beta \tag{2.1}$$

where x and y are respectively input and output, μ, σ, γ, and β are parameters fixed during inference, and ε is to avoid round-off problems.

2.1.2 CIFAR-10 BNN Model

Our work focuses on implementing the *BinaryNet* BNN architecture introduced by Courbariaux et al. [CHS$^+$16] for the CIFAR-10 dataset [KH09]. This dataset contains sixty thousand 32×32 3-channel images drawn from 10 classes consisting of vehicles and animals (airplane, truck, cat, etc.) divided into a training set of 50000 and a test set of 10000. While the dataset is small by ML standards it is sufficient to demonstrate the potential of the BinaryNet BNN for computer vision tasks. The Courbariaux et al. architecture is heavily inspired by the VGG models [SZ15], and consists of six conv layers followed by three FC layers. All conv layers use 3×3 filters and edge padding, and all conv/FC layers apply batch norm before binarization. There is a 2×2 max pooling layer after the 2nd, 4th, and 6th conv layers. The first conv layer is different from the rest as its input is the (floating-point) image (its weights are still binary). The architecture is summarized in Table 2.1. The size of the feature maps diminish deeper into the network, and the first two

Table 2.1: Architecture of the BinaryNet CIFAR-10 BNN — The weight bits exclude batch norm parameters, whose total size after optimization is less than 1% of the weights.

Layer	Input Fmaps	Output Fmaps	Output Dim	Output Bits	Weight Bits
Conv1	3	128	32	128K	3456
Conv2	128	128	32	128K	144K
Pool	128	128	16	32K	
Conv3	128	256	16	64K	288K
Conv4	256	256	16	64K	576K
Pool	256	256	8	16K	
Conv5	256	512	8	32K	1.1M
Conv6	512	512	8	32K	2.3M
Pool	512	512	4	8192	
FC1	8192	1024	1	1024	8.0M
FC2	1024	1024	1	1024	1.0M
FC3	1024	10	1	10	10K
Total					13.4M
Conv					4.36M
FC					9.01M

dense layers contain most of the weights. We trained this network using open-source Python code provided by the authors [1], and reached 11.58% test error out-of-the-box, in line with their results.

2.1.3 Algorithmic Modifications for Hardware

A number of optimizations were made to the BNN algorithm to make it more efficient for FPGA inference. As with conventional CNNs, one key optimization is data quantization. While the weights are already binarized, the biases and batch norm parameters in the BinaryNet model are real numbers. For biases, we noticed that nearly all biases were much smaller than 1 (recall that activations have magnitude 1). We removed biases entirely and retrained the network reaching a test error of 11.32%. We take this number as the baseline error rate.

A second optimization involved noting that batch norm (Equation (2.1)), being a linear trans-

[1] https://github.com/MatthieuCourbariaux/BinaryNet

Table 2.2: BinaryNet BNN accuracy with various optimization — `no-bias` refers to retraining after removing biases from all layers and `fixed-point` refers to quantization of the inputs and batch norm parameters.

Source	Model	Padding	Test Error
From [CHS+16]	-	0	11.40%
Python	Default	0	11.58%
Python	no-bias	0	11.32%
Python	no-bias	+1	11.82%
C++	no-bias, fixed-point	0	11.46%
C++	no-bias, fixed-point	+1	12.27%

formation, can be formulated as $y = kx + h$, where:

$$k = \frac{\gamma}{\sqrt{\sigma^2 + \varepsilon}} \qquad \text{and} \qquad h = \beta - \frac{\mu\gamma}{\sqrt{\sigma^2 + \varepsilon}} \tag{2.2}$$

This reduces the number of operations and cuts the stored parameters to two. Furthermore, the BinaryNet always binarizes immediately after each batch norm. Thus we do not need the magnitude of y, only the sign. This allows us to scale both k and h into the range of our fixed-point implementation. Empirical testing showed that k and h can be quantized to 16 bits with negligible accuracy loss while being a good fit for power-of-2 word sizes. We also quantized the floating point input images to 20-bit fixed point. Table 2.2 summarizes the impact of each algorithmic modification on test error. C++ refers to the HLS code and the synthesized accelerator, both of which achieve the same test accuracy.

One complication in the BinaryNet model is the interaction between binarization and edge padding. The model binarizes each activation to -1 or +1, but each input fmap is edge padded with zeros, meaning that a convolution can see up to 3 values: -1, 0, or +1. Thus the BinaryNet model actually requires some 2-bit operators (though the fmap data can still be stored in binary form). We managed to modify and retrain the BinaryNet model to pad with +1, eliminating the zeros and creating a truly binarized CNN. This +1 padded BNN achieves a test error of 11.82% in Python and 12.27% in C++/FPGA, only slightly worse than the original. For our FPGA implementation we used the 0 padded BNN as the resource savings of the +1 padded version was not particularly relevant for the target device.

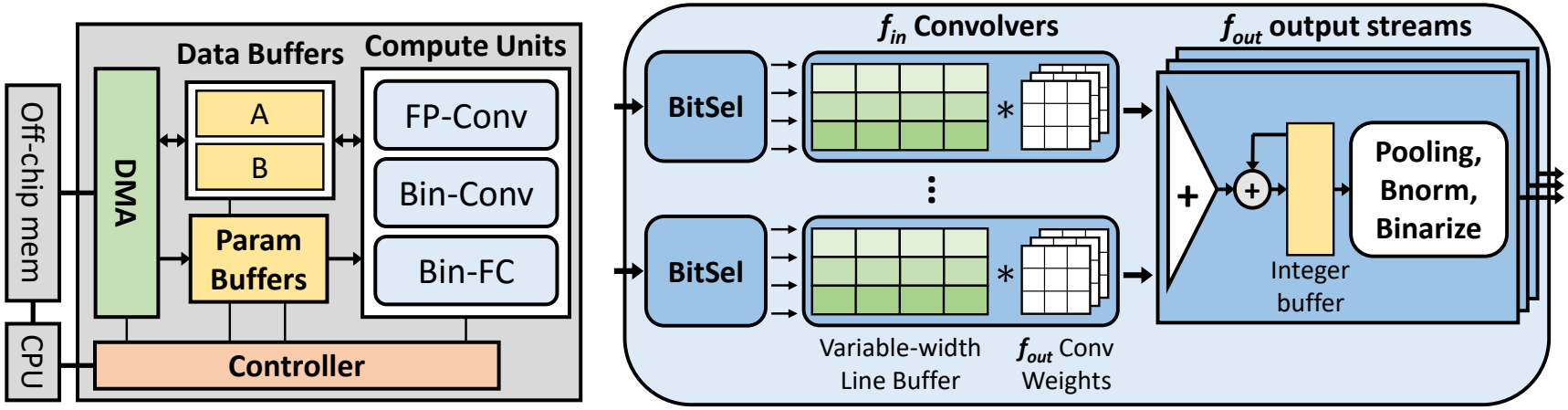


Figure 2.3: Architectural diagram of the BNN accelerator — (a) system-level block diagram showing the three compute units, buffers, and how the accelerator is connected to the CPU and off-chip memory hierarchy; (b) architecture of the Bin-Conv unit with input and output parallelization factors $f_{in} = 2$ and $f_{out} = 3$. The unit can stream in two words per cycle and produce three output fmaps per invocation.

2.2 BNN Hardware Architecture

2.2.1 System Architecture

Figure 2.3(a) illustrates our overall system architecture, which consists of three compute units, data and weight buffers, a direct memory access (DMA) system for off-chip memory transfer, and an FSM controller. The three compute units work on different types of layers: the *FP-Conv* unit for the (non-binary) first conv layer, the *Bin-Conv* unit for the five binary conv layers, and the *Bin-FC* unit for the three binary FC layers. Of the three, the Bin-Conv and Bin-FC units must be configurable to handle different numbers of input and output feature maps (fmaps), and in the case of Bin-Conv different fmap sizes.

The storage of intermediate data in our accelerator differs from most existing designs. In full-precision CNN accelerators, the output fmaps of any single layer typically exceed the size of on-chip storage. This necessitates the transfer of fmaps to and from off-chip RAM. However, as Table 2.1 shows, the size of the largest set of fmaps in our BNN is only 128K bits, which easily fits in on-chip SRAM. Our design uses two in-out data buffers A and B of equal size. One layer reads from A and write its outputs to B. Then, without any external data transfers, the next layer can read from B and write to A. Off-chip memory transfers are only needed for the input image, output prediction, and loading each layer's weights.

Unlike the fmaps, there is only enough memory on-chip to store a portion of a layer's weights. Multiple accelerator invocations may be needed for a layer. In each invocation we load in a new set of weights and produce a new set of fmaps. We keep invoking the accelerator until all output fmaps have been generated and stored in the on-chip buffers.

2.2.2 Compute Unit Architecture

In our accelerator, each compute unit must store *binarized* data to the on-chip RAMs at the end of its execution. As Figure 2.2 reveals, the first operation of a conv or FC layer transforms the

binary inputs to integers. We make sure each unit will also perform the subsequent batch-norm, pooling, and binarization before writing data out to the buffers. One of our design goals is to limit the amount of integer-valued intermediate data buffered inside each compute unit.

FP-Conv – The fixed-point conv unit utilizes the well-known line buffer architecture for 2D convolutions. Because this unit only targets a single layer, we hard-wire it to handle a 3-channel 32×32 input. While the input pixels are 20-bit fixed-point, the weights are binarized, so we can replace the multiplies in the conv operation with sign inversions. We fully parallelize across the three input channels: in each cycle we stream in three input pixels, add them to three line buffers, and compute a 3×3×3 convolution. The result is put through batch norm and binarization to produce one output bit per cycle. Greater parallelism in this unit is achievable, but the first conv layer takes up a very small portion of the overall run time, and we focused design efforts into the other units.

Bin-Conv – The binary conv unit is the most critical component of the accelerator, as it is responsible for the five binary conv layers which take up the vast majority of run time on CPU/GPU. The unit must maintain high throughput and resource efficiency while handling different input widths *at run time*. Our design targets input widths of 8, 16, or 32, and can support larger power-of-two widths with minor changes. To efficiently compute a convolution, multiple rows of input pixels need to be buffered for simultaneous access. However, a standard line buffer (e.g., from video processing applications) is unsuitable as (1) it must be sized for the largest input width and will underutilized for smaller widths; (2) a standard line buffer shifts in one pixel per cycle, but with binarized data we can expect to stream in 32 or more bits per cycle. Tiling can help resolve the input width problem, but fmaps in a BNN can be as small as 4 bits, rendering tiling ineffective.

To address the above issues, we propose a *variable-width line buffer* (VWLB). For brevity we do not fully describe the VWLB's operation, but importantly it allows us to process an entire data word's worth of pixels each cycle regardless of input width. This essentially exploits the crucial sub-word parallelism in a BNN's binary feature data.

Figure 2.3(b) shows the complete Bin-Conv unit. f_{in} is the input parallelization factor (the

unit takes in f_{in} parallel input words each cycle) and f_{out} is the output parallelization factor (the unit produces f_{out} parallel partial sums for different fmaps each cycle). Each input word stream connects to a Convolver, which contains a VWLB and circuits to perform f_{out} conv filters per cycle on the data buffered in the VWLB. The resulting partial sums are stored in f_{out} integer buffers. When an output fmap is finished, it is put through max-pooling, batch norm, and binarization (the latter two operations are non-linear and cannot be applied to partially accumulated fmaps).

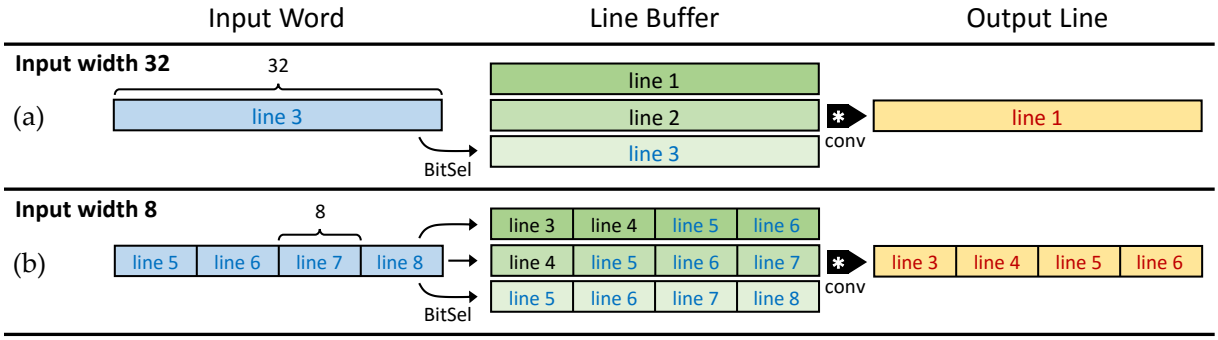


Figure 2.4: Example usage of the variable-width line buffer — a 32-bit input word is divided up by BitSel and inserted into the VWLB. The line buffer has height 3 and width 32. Sliding a 3×3 conv filter across the VWLB produces 32 output pixels, ignoring edge effects. (a) for a 32-wide input fmap, each row of the VWLB stores one line and applying the conv filter produces one 32-wide output line; (b) for an 8-wide input fmap, each row of the VWLB stores four lines and applying the conv filter produces four consecutive 8-wide output lines given the mapping of input lines to banks shown.

Figure 2.4 explains the operation of the BitSel and VWLB in greater detail. The diagram assumes we have a word size of 32 bits and a 3×3 conv filter, which requires a VWLB with three rows and 32 elements per row. We demonstrate how the VWLB works for input fmap widths 32 and 8 and ignore edge padding for the sake of simplicity.

1. For a 32-wide input, each word contains exactly one line. Each cycle, the VWLB shifts up and the new 32-bit line is written to the bottom row. We can then slide the 3×3 conv window across the VWLB to generate one 32-bit line of conv outputs.

2. For an 8-wide input, each word contains four lines. We split each VWLB row into four banks, and map each input line to *one or more* VWLB banks. The mapping is done in such a

way that sliding the conv window across the VWLB produces four consecutive 8-bit output lines. Each cycle the VWLB shifts both up and to the left.

The BitSel is responsible for slicing the input word and mapping the slices to the row banks. Because the smallest input width is 8, each slice and VWLB bank is sized at 8 bits. For a 32-wide input, BitSel maps four contiguous 8-bit slices to the bottom row. For an 8-wide input, the mapping is more complex, but still highly regular and can be computed in hardware with just adds and shifts. Each pixel in the output lines in Figure 2.4 is an integer conv sum, and each sum is accumulated at a different location in the integer buffer.

The BitSel and VWLB provides three primary advantages: (1) the VWLB achieves full hardware utilization regardless of input width, (2) a new input word can be buffered every cycle, and (3) the BitSel deals with various input widths by itself, allowing the actual buffer and convolution logic to be fixed. Note that the VWLB used in our design differs from Figure 2.4 in a few details. First, we have neglected edge padding. The actual VWLB contains two additional elements per bank to hold horizontal pad bits. Vertical padding is handled by inserting lines of zeros. Second, because the pad bits are 0 rather than +1 or -1, we must make each element in the VWLB two bits instead of one. The conv operation is performed between the 2-bit data and 1-bit weights, and can be implemented as sign inversion and accumulate.

Our Bin-Conv design exploits all forms of parallelism in a BNN conv layer: parallelism within the 3x3 convolution, parallelism across pixels (via the VWLB), and parallelism across both input and output feature maps (via f_{in} and f_{out}).

Bin-FC – The binary FC unit is comparatively simple. Each cycle we read in f_{in} data words and an equal number of weight words. Here f_{in} is the input parallelization factor in Bin-Conv. We perform a dot product between the data and weight words by applying a bitwise XOR operation and then summing the resulting bits with a popcount. Similar to the Bin-Conv unit, we accumulate the sum in an integer buffer and apply binarization after all inputs have been processed. Note that the FC layers are typically bound by memory bandwidth of the off-chip connection, rather than the throughput of the accelerator.

Data Buffers – To accommodate multiple reads per cycle, the data buffers are partitioned into f_{in} banks, and feature maps are interleaved across the different banks. Figure 2.5 shows an example with $f_{in} = 2$ and four words per fmap. The data words are read sequentially by address, so a compute unit always accesses f_{in} consecutive fmaps in parallel.

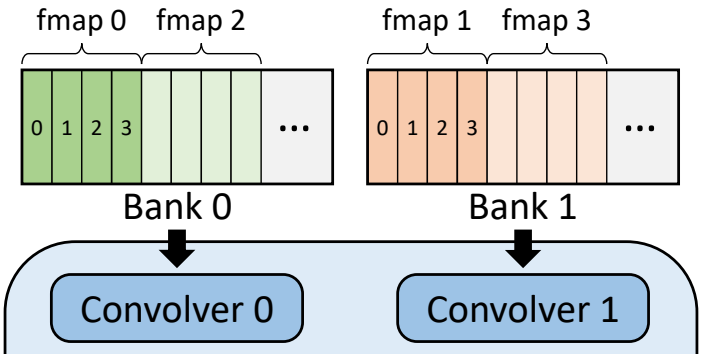


Figure 2.5: Memory banking in the BNN data buffer — The compute unit and memory system have $f_{in} = 2$. Each fmap contains four words which are laid out sequentially. The fmaps are interleaved across banks, and both Bin-Conv and Bin-FC benefit from this banking.

2.2.3 Implementation Using High-Level Synthesis

Figure 2.6 shows the HLS pseudocode for the front half of the Bin-Con unit, and demonstrates a key difference between BNN and CNN hardware design. For a CNN the code typically loops over an fmap processing one pixel at a time, and key design decisions include loop ordering and unroll factors [ZLS+15]. In our BNN accelerator, the basic atom of processing is not a pixel but a word. The example code is designed to sustain one word per cycle throughput over the entire input feature map set. Each fmap consists of `words_per_fmap` words, a number which differs between layers. As it processes the input set, the code updates the weights on each new fmap and accumulates the conv results in `outbuf`. We call `BitSel` and `conv` inside the loop to instantiate the BitSel units and conv logic as shown in Figure 2.3(b). To increase the number of input streams we can tile the loop and unroll the inner loop body.

A key design decision here is the input word size, which controls the level of parallelism across the pixels of an fmap. To guarantee correctness, `words_per_fmap` must be an integer greater than

zero. This constrains the word size to at most the size of the smallest input fmap ($8 \times 8 = 64$ bits in our case). The word size restriction is not a significant limiting factor in our design, as 64 is already a very large parallelization factor (it means we perform 64 convolutions per cycle), and there are other sources of parallelism to exploit in the BNN. We chose a word size of 64 bits for the data buffers and sized each data buffer A and B at 2048 words, which is just enough to store the largest set of fmaps in the BNN.

We also explored different values for f_{in} and f_{out} in Bin-Conv. It was observed that both have roughly similar effects on execution time, but increasing f_{out} has a more severe effect on total area. f_{in} controls the number of BitSels and VWLBs while f_{out} controls the number of pooling/batch norm units and integer buffers. In terms of logic a BitSel and a pooling/batch norm unit is similar, but each VWLB contains 32×3 2-bit registers while each integer buffer contains 32×32 12-bit registers. Thus all else being equal it is better to increase f_{in}. This result shows the importance of minimizing the storage of intermediate values and only committing binarized data to memory.

We target our hardware implementation for a low-power embedded SoC with FPGA co-processor. We use Xilinx SDSoC as the primary design tool for our BNN application. SDSoC takes as input a software program with certain functions marked as "hardware". It invokes Vivado HLS under the hood to synthesize the "hardware" portion into RTL. In addition, it automatically generates the data motion network and DMA necessary for memory transfer between CPU and FPGA based on the specified software-hardware partitioning. We selected a DMA engine built for contiguous memory since it has the highest throughput, and a neural network's data and weights can be laid out contiguously. We used directives to ensure that data is only transferred on the first and last accelerator invocation. Weights are transferred on every invocation.

```
VariableLineBuffer linebuf;
ConvWeights wts;
IntegerBuffer outbuf;

for (i = 0; i < n_input_words; i++) {
  #pragma HLS pipeline

  // read input word, update linebuffer
  WordType word = input_data[i];
  BitSel(linebuf, word, input_width);

  // update the weights each time we
  // begin to process a new fmap
  if (i % words_per_fmap == 0)
    wts = weights[i / words_per_fmap];

  // perform conv across linebuffer
  for (c = 0; c < LINE_BUF_COLS; c++) {
  #pragma HLS unroll
    outbuf[i % words_per_fmap][c] +=
      conv(c, linebuf, wts);
  }
}
```

Figure 2.6: HLS pseudocode for part of the Bin-Conv unit — the pseudocode implements a pipeline which reads and performs convolution on one input word each cycle. Many details are left out; the goal is to illustrate how our design can be expressed in high-level code.

2.3 Experimental Evaluation

2.3.1 Experiment Setup

We evaluate our design on a ZedBoard, which uses a low-cost Xilinx Zynq-7000 SoC containing an XC7Z020 FPGA alongside an ARM Cortex-A9 embedded processor. We make use of Xilinx SDSoC 2016.1 as the primary design tool, which leverages Vivado HLS and Vivado to perform the actual HLS compilation and FPGA implementation. We compared our design against two server-class computing platforms: an Intel Xeon E5-2640 multi-core processor (**CPU**) and an NVIDIA Tesla K40 GPU (**GPU**). We also compared against an NVIDIA Jetson TK1 embedded GPU board (**mGPU**). As BNNs are a recent development, our baseline applications will not be as well optimized compared to CNN baselines. The CPU and GPU baselines are adapted from code provided in [CHS+16], which uses Theano with OpenBLAS. The code does not perform bitwise

Table 2.3: Comparison of different BNN configurations — Last row shows the resources available on the device; Run time is in milliseconds. * indicates our chosen configuration.

f_{in}	LUT	FF	BRAM	DSP	Runtime
1	25289	28197	86	3	17.5
2	35291	37125	87	3	10.8
4	38906	36771	87	3	7.98
8*	46900	46134	94	3	5.94
Device	53200	106400	140	220	-

optimizations since they are not natively supported in Theano, and instead uses floating-point values binarized to -1 and +1. For the baselines we used the BNN model with no biases and k and h, and on GPU we used the largest possible batch size. Power measurement is obtained via a power monitor. We measured 4.5W idle and 4.7W max power on the Zedboard supply line when running our BNN, so the dynamic power consumption of the accelerator is very low. Our CPU baseline was written from scratch and uses bitwise logic for the FC layers. But due to the edge-padding issue, it uses 8-bit arithmetic instead for the conv layers (similar to how our FPGA uses 2-bit arithmetic).

2.3.2 Design Parameter Study and Tuning

Table 2.3 shows the performance and resource utilization of our accelerator using different values of f_{in} for the Bin-Conv and Bin-FC units. All numbers are post place and route. In our experiments f_{out} is set to 1. Performance scaling is below unity as f_{out} only improves parallelism in the binary conv layers. We use $f_{in} = 8$ in the rest of the experiments.

We compare the performance of our accelerator to the baselines in Table 2.4. As raw throughput depends heavily on device size, we also show the power consumption and the throughput per Watt. The FPGA design obtains 15.1x better performance and 11.6x better throughput per Watt over **mGPU**, which has a similar power envelope. Against the x86 processor, it achieves a 2.5x speedup. While the binary conv layers were faster, the FC layers were slower, which is unsurprising as the FC layers are bound by external memory bandwidth. Versus **GPU**, the FPGA is 8.1x worse in performance. But as expected, it has much lower power consumption and better throughput per

Table 2.4: BNN performance comparison — Conv1 is the first FP conv layer, **Conv2-5** are the binary conv layers, **FC1-3** are the FC layers. A – indicates a value we could not measure. Numbers with * are sourced from datasheets. The last row shows power efficiency in throughput per Watt.

	Execution time per image (ms)			
	mGPU	**CPU**	**GPU**	**FPGA**
Conv1	–	0.68	0.01	1.13
Conv2-5	–	13.2	0.68	2.68
FC1-3	–	0.92	0.04	2.13
Total	90	14.8	0.73	5.94
Speedup	1.0x	6.1x	123x	15.1x
Power (Watt)	3.6	95*	235*	4.7
imgs/sec/Watt	3.09	0.71	5.83	35.8

Watt.

To show that the FC layers are indeed limited by memory bandwidth, we tested a design where the FC computations are removed but the memory transfers are retained. The new FC execution time was within 5% that of the original, demonstrating that further compute parallelization would result in very little gain.

2.3.3 Comparison Against CNN Accelerators

Table 2.5 compares our implementation against state-of-the-art FPGA accelerators found in literature. All numbers are retrieved from the respective papers. Note that two of the comparisons are against larger FPGAs while one is against the same device. Throughput is shown in giga-operations-per-second (GOPS), and we count adds and multiplies following [ZLS+15]: each binary xor, negation, or addition counts as one operation. Our BNN accelerator beats the best known FPGA accelerators in pure throughput, and is also much more resource and power efficient. BNNs save especially on the number of DSPs since multiplication/division is only needed for batch norm and not for the compute-intensive conv or FC calculations. The metrics of throughput per kLUT and throughput per Watt are especially important — due to the relative novelty of the BNN, we were only able to obtain a suitable network for CIFAR-10 while previous work shows results

Table 2.5: BNN resource comparison against state-of-the-art FPGA accelerators — GOPS counts multiplies and adds per second. * indicates values approximated from charts.

	[SCD+16]	[QWY+16]	[VB16]	Ours
Platform	Stratix-V GSD8	Zynq 7Z045	Zynq 7Z020	Zynq 7Z020
Capacity (kLUTs)	695	218.6	53.2	53.2
Clock(MHz)	120	150	100	143
Power(W)	19.1	9.6	-	4.7
Precision	8-16b	16b	-	1-2b
GOPS (conv)	136.5	187.8	-	318.9
GOPS (all)	117.8	137.0	12.73	207.8
kLUTs	120*	182.6	43.2	46.9
DSPs	760*	780	208	3
GOPS / kLUT	0.98	0.75	0.29	4.43
GOPS / Watt	6.17	14.3	7.27	44.2

for larger ImageNet networks. However, our data provides evidence that BNN is a better algorithm for FPGA than CNN, enabling far more efficient usage of resource and power.

While it may not be completely fair to compare GOPS between a binarized and conventional network, it is standard practice for hardware accelerator studies to compare reduced and full-precision implementations that use different data types. Indeed, previous work we cite all use different data types so a fair comparison is impossible.

2.4 Conclusions and Future Work

In this chapter, we demonstrate an HLS-based FPGA accelerator for binarized neural networks on FPGA. BNNs feature potentially reduced storage requirements and binary arithmetic operations, making them well suited to the FPGA fabric. However, these characteristics also render CNN design constructs such as input tiles and line buffers ineffective. We introduce new design constructs such as a variable-width line buffer to address these challenges, creating an accelerator radically different from existing work. We leverage modern HLS tools to write our design

in productive, high-level code, and our accelerator outperforms existing work in raw throughput, throughput per area, and throughput per Watt.

Future work should focus both on algorithmic and architectural improvements. On the algorithmic side we would like to explore techniques to reduce model size. From the architectural side one action item is to implement a low-precision network for ImageNet, which would involve a much larger and more complicated accelerator design.

CHAPTER 3

MAPPING-AWARE PIPELINE SCHEDULING FOR HIGH-LEVEL SYNTHESIS

3.1 Preliminaries

High-level synthesis (HLS) has established itself as a powerful alternative to the conventional register-transfer level (RTL) design. HLS allows designers to automatically synthesize hardware from high-level specifications written in a software programming language and making microarchitectural optimizations (such as pipelining or memory banking) accessible in the form of pragmas. By raising the design abstraction to that of a software language, HLS tools for FPGAs such as LegUp [CCA+11], Vivado HLS [CLN+11], and Intel FPGA SDK for OpenCL [CAD+12] is able to greatly reduce engineering effort and turn-around time of an FPGA design.

HLS already has a proven track record in hardware design for signal processing and cryptographic applications. As Chapter 2 demonstrates, HLS can also be effective for building neural network accelerators, as shown by a number of recent publications utilizing HLS tools for DNN specialization [ZLS+15, SCD+16, DSSMS16, NSW18, MP18]. As DNNs are dominated by highly-regular loop nests, such accelerators frequently make use of an HLS optimization called *loop pipelining*. Pipelining produces a circuit which *partially* overlaps successive iterations of a loop, thus exploiting inter-iteration parallelism without duplicating the entire loop body. In this chapter of the thesis, we propose a technique to enhance HLS pipelining for logic operations, which is highly relevant to the HLS implementation of extremely low-precision neural networks where multiplies and adds can often be mapped to simple logic operations. The practicality of this technique is demonstrated by applying it as a proof-of-concept to a module from the binary CNN accelerator from Chapter 2.

3.1.1 Pipelining in High-Level Synthesis

HLS synthesizes a pipeline in multiple distinct steps. First, it schedules the operations in the loop body, creating the hardware register boundaries. Then, it allocates hardware resources for the operations and binds each operation to a resource instance. For pipelining to succeed, the loop body must be scheduled such that it can be repeated at some *initiation interval* (II) which is less than the loop body's latency. HLS typically uses a software compiler technique known as modulo scheduling [Rau94] to schedule pipelines, which assumes each operation incurs a fixed delay based on operation type (add, mul, etc). Critically, while this assumption mostly holds when targeting general-purpose processors, it fails in the case of simple logic operations for LUT-based FPGAs. Below we illustrate how static modulo scheduling can generate an overly conservative schedule:

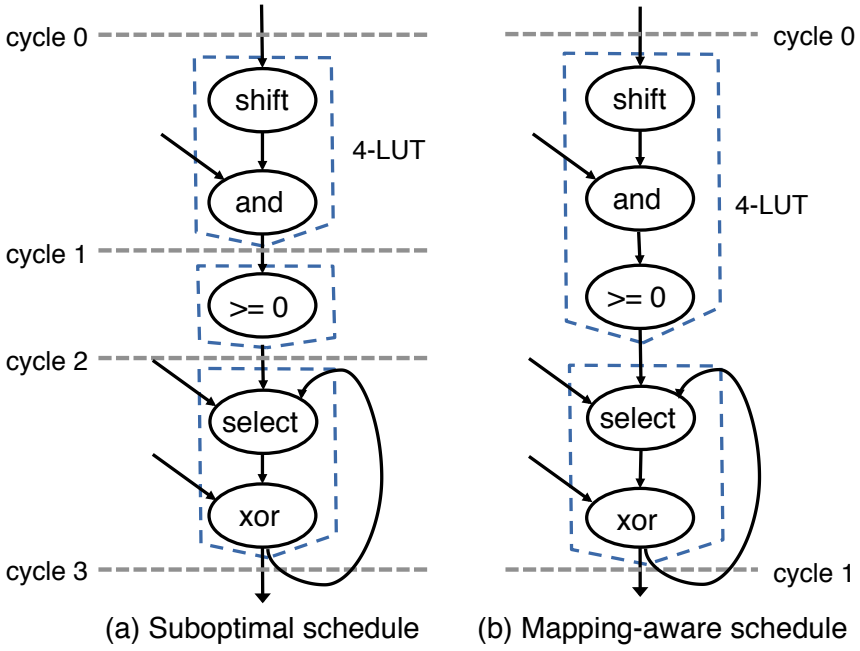


Figure 3.1: Pipeline schedule for Reed-Solomon encoder – target clock period is 5ns; each logic operation or LUT incurs a 2ns delay. (a) Suboptimal schedule requires 3 LUTs and 3 pipeline stages; (b) Optimal schedule requires 2 LUTs and 1 pipeline stage.

Figure 3.1 shows the data-flow graph (DFG) of a kernel in Reed-Solomon encoding [ANA10], a prominent error-correction algorithm. We assume the target FPGA uses 4-input LUTs and the target II is one cycle. Using pre-characterized delays, modulo scheduling generates a three-stage pipeline (left). In practice, however, it is possible to map the entire kernel to only two LUTs, which

can be chained combinationally in one cycle (right). The key issue is that the delay in a complex logic network is not simply additive: a complex chain of logic operations might be implemented in just a few LUTs. The mapping-agnostic scheduler is conservative and inserts extra registers. At the same time, the downstream technology mapping cannot shorten the pipeline as it must respect register boundaries. To bridge this QoR gap, the scheduling algorithm must be made aware of the underlying LUT-based hardware and potential mapping optimizations.

In this chapter, we present a mixed integer linear programming (MILP) formulation of modulo scheduling to perform mapping-aware pipeline synthesis. Our method allows the MILP to select the optimal mapping for each operation to minimize LUT and register utilization. While an ILP is inherently unscalable to large designs, we successfully demonstrate the benefits of a cross-layered pipelining approach compared to a state-of-the-art commercial HLS tool. This research is inspired by the work of fellow student Mingxing Tan [TDGZ15], though Tan does not address pipelined designs. We published this work in *DAC'15* [ZTDZ15].

3.1.2 Background and Related Works

Modulo scheduling is a well-known compiler optimization technique to realize software pipelining [Rau94]. It is also extensively used in the HLS context for enabling loop and function pipelining. For example, state-of-the-art HLS tools such as LegUp [CCA+11] and Vivado HLS [CLN+11] make use of a mathematical programming formulation known as system of difference constraints (SDC) to support modulo scheduling for hardware pipelining synthesis [ZL13, CCA+13]. Other developments to incorporate memory port reduction [BAMR10], pipeline flushing [DTHZ14], polyhedral analysis [MDQ13], and multithreading [TLDZ14] continue to advance the pipelining capabilities of HLS.

For an FPGA, technology mapping is the process of covering a network of logic gates with LUTs [CCP06]. While some prominent mapping techniques include FlowMap [CD94], CutMap [CH95], and DaoMap [CC04], there exists numerous mapping techniques that optimize for LUT depth [CD94] or area [CWD99, CC04]. Meanwhile, Pan et al. proposed a retiming-based

technology mapping technique that considers mapping for register repositioning to achieve the minimum clock period [PL98, PKL98]. Area-efficient mapping is an important step in achieving good QoR in the physical implementation flow. Unfortunately, while latency-optimal technology mapping can be achieved in polynomial time [CD94], area-optimal mapping is NP-hard [FS94]. An ILP-based algorithm for minimum-area LUT mapping is proposed in [CH05]. Unlike our approach, however, it eschews the usage of cuts. Recent research has focused on integrating mapping with the upstream tool flow to explore additional optimizations in high-level analysis. In particular, Zheng et al. propose a flow that iterates between upstream scheduling and downstream mapping and place-and-route and uses post-physical implementation timing for re-scheduling [ZGRC14].

3.2 A Mapping-Aware Pipeline Scheduler

The key idea behind our approach is to perform pipeline scheduling while taking LUT mapping into consideration. To accomplish this, we extend cutenumeration from conventional technology mapping algorithms into the HLS domain. A brief description of traditional cut enumerations is as follows.

Let v be a node on a graph representing a bit-level logic network. Then we define O_v, a *cone* of v, as a sub-graph of v and its predecessors such that there exists a path from any node in O_v to v that is entirely contained within O_v. The cut of O_v, denoted as C_v, is then defined to be the set of nodes not in O_v with an edge pointing to a node in O_v. A cone and its associated cut is defined to be K-feasible if the cut contains K or fewer nodes. Because a K-input LUT can implement any K-bounded logic network, a K-feasible cone can be mapped to a single K-input LUT.

Cut enumeration is the process of identifying the set of all K-feasible cuts for every node in the graph. After this is done, the technology mapping problem becomes one of selecting a set of cuts whose cones cover each node in the graph, while minimizing some objective such as total latency or LUT area.

Typically, cut enumeration is done on a bit-level directed acyclic graph (DAG). However, the

pipeline scheduling problem operates on a word-level control data flow graph (CDFG). Our approach is to use a modified cut enumeration algorithm to find the cut set of each operation, and construct an MILP using the cut information which simultaneously schedules each operation while selecting an optimal set of cuts which covers the CDFG. More formally, here is our area-minimizing modulo scheduling problem formulation:

Given: (1) A CDFG for a function or loop whose edges capture inter-iteration and intra-iteration data dependences between operations; (2) A target clock period T_{cp}; (3) A target initiation interval II; (4) A set of constraints C including latency constraints, cycle time constraints, and resource constraints; (5) Characterized delays for operations on a target FPGA device using K-input LUTs.

Goal: Find a minimum area modulo schedule for the operations so that no constraints in C are violated, and within each cycle, there exists a feasible K-input LUT mapping that meets T_{cp}.

3.2.1 Word-Level Cut Enumeration

An intuitive approach to word-level cut enumeration is to break down the word-level DFG into a bit-level graph [ZZZ+10] and use a traditional method. Tan et al. [TDGZ15] proposed a word-level cut enumeration algorithm, but only for simplicity since the scheduling approach in their paper can handle both bit-level and word-level graphs. In this work, bit-level decomposition would generate an enormous number of cuts and make an MILP approach intractable. A word-level cut enumeration algorithm is therefore necessary.

Bitwise operations such as `AND/OR/XOR` are straightforward because the operations on different bits are completely independent. The key challenge arises for non-bitwise operations, such as shifting or arithmetic, where a single bit of the output might depend on multiple bits of each input operand. For instance, given an addition operation $out[1:0] = in_1[1:0] + in_2[1:0]$, the most significant output bit $out[1]$ would depend on four input bits: $in_1[0]$, $in_1[1]$, $in_2[0]$, and $in_2[1]$, coming from two nodes in_1, in_2 on the DFG. To address this problem, we use a bit-level dependence tracking technique on the word-level DFG. Instead of just identifying dependent values, our algorithm

also tracks all dependent bits of each value. To limit our analysis to those operations which are mapped to LUTs, we define a *black box (BB)* operation as one which does not map to LUTs, (i.e., memory access operations). The following applies then to all non-BB nodes in the DFG.

Let $v[j]$ denote the a bit j of the operand v. We then classify all operations into three classes, and define the DEP function for each class of operations as follows:

- Bit-wise operations (`AND/OR/XOR`): each output bit only depends on a single bit of each input operand. For example, the DEP for operation $out = in_1 \ \& \ in_2$ is defined as: $DEP(out[j]) = \{in_1[j], in_2[j])\}$.
- Shifting operations (`LSFHIT/RSHIFT`): each output bit depends on a shifted single bit of each input operand. For example, the DEP for operation $out = in_1 >> s$ is defined as: $DEP(out[j]) = \{in_1[j+s]\}$.
- Arithmetic operations (`ADD/SUB/CMP`): each output bit can depend on multiple bits of each input operand. For example, the DEP for operation $out = in_1 + in_2$ is defined as follows: $DEP(out[j]) = \{in_1[j], in_1[j-1], ..., in_1[0], in_2[j], in_2[j-1]..., in_2[0]\}$.

The DEP function over a word-level value $DEP(v)$ is further defined as the union set of $DEP(v[j])$ for each bit $v[j]$. Based on the DEP function, we compute the K-feasible cut set for each node in the DFG by merging the K-feasible cuts for all of its inputs. Suppose v's inputs are u_1, u_2, ..., u_p, with associated cut sets CUT_{u_1}, CUT_{u_2}, ..., CUT_{u_p}, where CUT_u is a collection of cuts and each cut $C_i \in CUT_u$ is K-feasible. The K-feasible cut set for v can be computed as follows:

$$
\begin{aligned}
&CUT_v = mergeCuts(u_1, ..., u_p) = \\
&\{C' = \bigcup_{C_i \in CUT_{u_i}} \{DEPS(C_i)\}, \text{ if } |C'| < K\} \\
&\text{where } DEPS(C_i) = \bigcup_{t \in C_i} \{DEP(t)\}
\end{aligned}
\tag{3.1}
$$

Our cut enumeration procedure (Algorithm 1) iteratively applies Equation (3.1) to each node until all K-feasible cuts are obtained. We maintain a work list for nodes that need to be updated. Initially, the work list contains all operations, and the cut set for each node v is the trivial cut $\{\{v\}\}$.

For each node in the work list, we apply Equation (3.1) to compute the new cut set. If a new cut is added for a node, we update its cut set and add all its successors to the work list. We remove a node from the work list each time it is visited. The algorithm terminates when the work list becomes empty. For each black-box operation, we simply force its cut set to be its trivial cut. Previous studies have shown that cut enumeration is an exponential algorithm with respect to K [CWD99]. Fortunately, cut enumeration is typically very fast as the value of K is small in practice ($K \leq 6$).

Algorithm 1: *CutGen*$(CDFG)$

```
   input  : CDFG – control data flow graph
   output: CUT – cut set for all CDFG nodes
   // Initialize the trivial cut for each node.
1  foreach node v in CDFG do
2      CUT_v = {{v}}
3  L ← list of CDFG nodes in topological order
   // Iteratively update cut set
4  while L ≠ Φ do
5      get the head node v from L
6      if v is not a primary input or black box operation then
7          newCutSet = mergeCuts(v)
8          if newCutSet ≠ CUT_v then
9              CUT_v ← newCutSet
10             append v's successors to L
```

Figure 3.2 demonstrates the cut enumeration for the example listed in Figure 3.1. The original Reed-Solomon application uses 32-bit operations, but for simplicity, we use 2-bit operations in this figure. Cut enumeration for A and B are relatively simple. Each bit of A depends on a single shifted bit of input s, while each bit of B depends on a single bit of each input operand t and A. In general, operation C would be treated as an arithmetic operation, but in this example, the comparison "`B >= 0`" is actually testing whether the most significant bit is zero or one. In this case, our algorithm will identify that the output of C only depends on the highest bit of each input based on bit-level dependence tracking. Our algorithm can also handle the cycle which arises from a loop-carried dependence when processing nodes D and E.

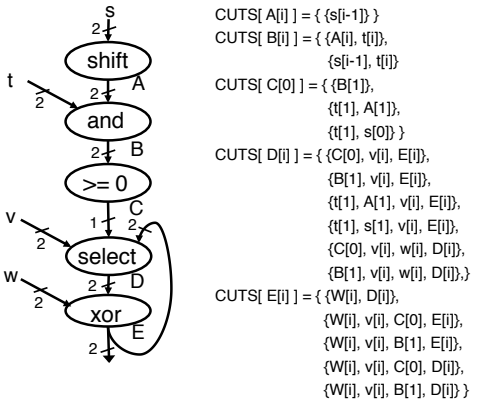


Figure 3.2: Cut enumeration for the Reed-Solomon decoder.

3.2.2 MILP Formulation of Modulo Scheduling

We formulate the modulo scheduling problem as a mixed integer linear program (MILP). Given a word-level dependence graph G and the cut set CUT_v of each graph node v computed via Algorithm 1, our formulation aims to compute an area-efficient pipelined schedule while respecting the following constraints:

LUT cover constraints – For each cut of v, we create a binary variable $c_{v,i}$, denoting whether cut i is selected for node v. We also define the binary variable $root_v$ as the sum of $c_{v,i}$ for all i. Conceptually, if $root_v = 1$ then v is the root of a cone that will be mapped to a LUT. Otherwise v will be mapped inside the cones of other nodes. Note that by the definition of $root_v$, the MILP can only select one cut for each node. Equation (3.3) ensures that each primary output (PO) is a root node, while Equation (3.4) ensures that the inputs of a selected cut are themselves root nodes.

$$root_v \in \{0,1\}$$

$$root_v = \sum_i c_{v,i} \quad \forall v \tag{3.2}$$

$$\sum_i c_{v,i} = 1 \quad \forall v \in \{POs\} \tag{3.3}$$

$$c_{v,i} \leq root_u \quad \forall u \in CUT_v[i] \tag{3.4}$$

Dependence constraints – The MILP uses a set of binary scheduling variables $s_{v,t}$, $0 \leq t \leq M$ where M is the bound on pipeline latency, to denote whether operation v is assigned to cycle t.

Equation (3.5) below constrains each operation to a single clock cycle, and Equation (3.6) defines S_v, an integer variable whose value is the actual cycle assigned to v.

$$\sum_t s_{v,t} = 1 \tag{3.5}$$

$$S_v = \sum_t t * s_{v,t} \tag{3.6}$$

Given a dependence between operations u and v with a distance $dist_{u,v}$[1] , we need to guarantee that operation u in iteration k is finished before we start the operation v in iteration $k + dist_{u,v}$. This can be captured by Equation (3.7):

$$S_u - S_v - II \cdot dist_{u,v} \leq 0 \tag{3.7}$$

Cycle time constraints – To consider the possibility of combinationally chaining multiple operations in a clock cycle, the MILP also assigns to each variable a start time L_v within the cycle, Note that L_v is a real number with bounds $0 \leq L_v \leq T_{CP}$, where T_{CP} is the target clock period. Suppose d_v is the delay of operation v in nanoseconds, Equation (3.8) ensures a combinational circuit will not across clock boundaries:

$$L_v + d_v \leq T_{CP} \tag{3.8}$$

Conceptually, if cut i is chosen for node v then there must be dependence constraints ensuring that each node u in $CUT_v[i]$ finishes before v begins. This is captured by adding the constraint in (3.9) for each u in $CUT_v[i]$:

$$\begin{aligned} &(T_{CP} * S_v + L_v) - (T_{CP} * S_u + L_u) \geq \\ &\quad c_{v,i} * (d_u - T_{CP} * II * dist_{uv}) \\ &\quad -(1 - c_{v,i}) * T_{CP} * CyclesMax \end{aligned} \tag{3.9}$$

Here $dist_{uv}$ is the dependence distance from u to v in number of iterations.

[1] A distance of 0 indicates an intra-iteration dependence, while a non-zero distance indicates an inter-iteration dependence.

For each dependence between nodes u and v, we must make sure u will not start later than v if they are scheduled in the same cycle. Equation (3.10) ensures this constraint:

$$\begin{aligned}&(S_u - S_v - II \cdot dist_{u,v}) * T_{CP} + (L_u - L_v + c_{v,i} \cdot d_u) \leq 0\\&\text{where } u \text{ is in } CUT_v[i].\end{aligned} \tag{3.10}$$

Essentially, Equation (3.10) enforces a difference between L_u and L_v if and only if u and v are scheduled in the same cycle (i.e., when $S_v - S_u - II \cdot dist_{u,v} = 0$). If the selected cut of v does not contain u ($c_{v,i} = 0$) then u and v will be constrained to have the same start time in the same cycle (i.e., $L_u = L_v$ and $S_u = S_v$), and will thus be mapped into the same LUT.

Register constraints – Register usage is captured by considering the liveness of each operation. We define binary variables $live_{v,t}$ to denote whether the result of operation v is live on cycle t. We shall show that $live_{v,t}$ can be calculated using binary variables $def_{v,t}$ and $kill_{u,t}$ and adding the constraint in Equation (3.13) for each u in $CUT_v[i]$:

$$def_{v,t} = \sum_{z=0}^{t-\lfloor d_v/T_{CP} \rfloor} s_{v,z} \tag{3.11}$$

$$kill_{v,t} = \sum_{z=0}^{t} s_{v,z} \tag{3.12}$$

$$def_{u,t} - kill_{v,t} - (1 - c_{v,i}) \leq live_{u,t} \tag{3.13}$$

Here $def_{v,t} = 1$ if the results of v is available on or before cycle t, and $kill_{v,t} = 1$ if the inputs of v are killed on or before cycle t. Thus Equation (3.13) ensures $live_{u,t} = 1$ on each cycle t where the results of u is defined and at least one fanout of u has not yet executed. The term $(1 - c_{v,i})$ ensures that if $CUT_v[i]$ is not selected, the liveness constraint between u and v is inactivated.

Using $live$ we can define the maximum number of registers needed on each cycle m, taking into account that operations separated by exactly II cycles will execute concurrently in a pipeline:

$$\begin{aligned}Reg(m) &= \sum_{t \in T'} \sum_{v \in V} Bits(v) * live_{v,t}\\&\text{where } T' = \{t \mid t \bmod II = m\}\end{aligned} \tag{3.14}$$

Here $Bits(v)$ is the number of bits in the value produced by v.

Resource constraints – In this work we consider resource constraints only for black-box operations. Because the cut set for such operations will always contain only the trivial cut, we do not need to consider mapping for them. The resource constraints in our MILP are therefore identical to those in a canonical modulo pipelining formulation [Rau94]. Let r denote a resource type, X_r the number of that resource used for the design, N_r the number of that resource available, and $R(v)$ the resource type for a black-box operation v. Then for each r:

$$\begin{gathered}\sum_{t\in T'}\sum_{v\in V'(r)} s_{v,t} - X_r \leq 0, \forall m : 0 \leq m < II \\ X_r \leq N_r \\ \text{where } T' = \{t \mid t \bmod II = m\} \\ \text{and } V'(r) = \{v \mid R(v) = r\}\end{gathered} \tag{3.15}$$

Objectives – The objective of the MILP is to minimize the weighted sum of the number of root nodes and the number of registers:

$$\textit{minimize} \quad \alpha \cdot \sum_{v\in V} Bits(v) * root_v + \beta \cdot \sum_{m=0}^{II-1} Reg(m) \tag{3.16}$$

Here α and β are user-defined parameters to trade-off the optimizations on LUT and register usage. Note that our MILP formulation can be easily extended to consider other type of resources such as embedded DSP blocks.

3.3 Experimental Evaluation

To implement our proposed idea, we leverage Vivado HLS, a popular commercial C-based HLS tool targeting Xilinx FPGAs. The tool uses LLVM as its front-end compiler, and we inject our technique as an additional LLVM pass applied before scheduling. By reordering instructions and inserting wait statements, we are able to enforce a custom schedule. We used Vivado HLS 2013.3 as the HLS tool, IBM ILOG CPLEX as the MILP solver, and Xilinx Vivado as the tool to implement the generated RTL. All timing and area numbers below were obtained post place and route.

To model the delays of each operation, we back annotated delay values parsed from the schedule report of the HLS tool for black-box operations. We restricted the MILP solver to run for at most 60 minutes. For all benchmarks, a feasible (but not necessarily optimal) solution was found in this amount of time. The values of α and β were set to 0.5 in all experiments.

To fairly evaluation of our mapping-aware approach, we include three sets of results for each benchmark: the commercial HLS tool, the MILP without mapping consideration (MILP-base), and the full MILP (MILP-map). MILP-base is implemented by skipping the cut enumeration step, and assigning to each node only the trivial cut. We show results for two MILPs to understand how much of the improvement is due to mapping consideration and not the differences between ILP and the heuristics employed by the HLS tool.

Table 3.1: Resource usage comparison — target clock period is 10ns. `CP` = achieved clock period; `LUT` = # of look-up tables; `FF` = # of flip-flops. The percentages next to each column is calculated relative to the HLS tool.

Design	Domain	Description	Method	CP(ns)	LUT	%	FF	%
CLZ	Kernel	Count the number of leading zeros in a 64-bit value	HLS Tool	5.43	171		221	
			MILP-base	4.29	152	(-11.1%)	226	(+2.3%)
			MILP-map	5.55	99	(-42.1%)	43	(-80.5%)
XORR	Kernel	XOR reduction for an array of elements	HLS Tool	5.55	3394		257	
			MILP-base	5.55	3394	(+0.0%)	257	(+0.0%)
			MILP-map	4.59	3264	(-3.8%)	0	(-100.0%)
GFMUL	Kernel	Efficient Galois field multiplication	HLS Tool	1.64	41		27	
			MILP-base	1.69	44	(+7.3%)	28	(+3.7%)
			MILP-map	3.36	39	(-4.9%)	0	(-100.0%)
CORDIC	Scientific Computing	Coordinate Rotation Digital Computer	HLS Tool	8.24	1313		631	
			MILP-base	5.19	1663	(+26.7%)	646	(+2.4%)
			MILP-map	7.58	1220	(-7.1%)	298	(-52.8%)
MT	Scientific Computing	Mersenne Twister pseudorandom number generation	HLS Tool	5.70	681		843	
			MILP-base	6.03	623	(-8.5%)	842	(-0.1%)
			MILP-map	7.17	640	(-6.0%)	526	(-37.6%)
AES	Cryptography	Advanced Encryption Standard	HLS Tool	5.27	4860		4720	
			MILP-base	5.33	4564	(-6.1%)	5316	(+12.6%)
			MILP-map	5.55	4475	(-7.9%)	2441	(-48.3%)
RS	Communication	Reed-Solomon decoder	HLS Tool	8.71	6493		8206	
			MILP-base	6.45	7308	(+12.6%)	7114	(-13.3%)
			MILP-map	9.26	6656	(+2.5%)	3856	(-53.0%)
DR	Machine Learning	Digit recognition using k-nearest neighbours algorithm	HLS Tool	5.29	1264		1365	
			MILP-base	5.55	1070	(-15.3%)	1088	(-20.3%)
			MILP-map	5.15	963	(-23.8%)	999	(-26.8%)
GSM	Communication	Global system for mobile communications	HLS Tool	7.76	1706		1231	
			MILP-base	8.53	1543	(-9.6%)	1074	(-12.8%)
			MILP-map	9.83	1766	(+3.5%)	493	(-60.0%)

3.3.1 Kernel and Small Application Results

Table 3.1 shows each of our benchmarks, what are divided into `kernels` (small loop bodies) and `applications` (complete real-life programs). Every benchmark (or its main loop) is pipelined to an *II* of 1.

Kernel results — The kernels saw essentially no improvement from HLS tool to MILP-base, but significant reductions in resource from the HLS tool to MILP-map. In `CLZ` MILP-map cut down the pipeline latency from 7 to 1, reducing the number of FFs required by 81% compared to MILP-base. In `GFMUL` and `XORR`, MILP-map was able to recognize that the entire pipeline can be implemented in a single combinational stage, eliminating registers altogether. MILP-map was also able to decrease the number of LUTs used by 19% over the three kernels.

To see how MILP-map obtained such enormous FF savings, we examine `XORR` in detail. `XORR` specifies an xor reduction over an array, which was synthesized into a depth 9 reduction tree. From scheduling reports we found that the HLS tool assigns a delay of 1.37ns to each xor operation and generated a 2-stage pipeline. MILP-base generated an identical schedule. By considering mapping, MILP-map was able to map multiple xors to a single LUT and fit the critical path into one cycle. This eliminates all pipeline registers and results in the observed FF savings.

Application results — The most convincing results appear in `CORDIC`, `MT`, and `AES`, where MILP-map was able to reduce FFs by an average of 46% over the HLS tool. In contrast, MILP-base was unable to achieve any register savings on these designs, and in fact incurred an average penalty of 4.8%. MILP-base was able to show some FF improvement in `RS`, `DR`, and `GSM` compared to baseline, but even on these designs, MILP-map achieved a reduction of 39% over MILP-base. Some variation in MILP-base can be attributed to imprecise delay estimates, as we could not back annotate a delay for every operation.

```
#pragma HLS array_partition variable=sum_m complete dim=0
#pragma HLS array_partition variable=dt_mem complete dim=1
#pragma HLS array_partition variable=wt_mem complete dim=1

AccumType sum_m[CONVOLVERS];
AccumType sum = 0;

for (int i = 0; i < n_inputs; i+=CONVOLVERS*WORD_SIZE) {
    #pragma HLS pipeline

    // Each convolver performs a binary dot product
    for (int j = 0; j < CONVOLVERS; ++j) {
        const Word in_wrd = dt_mem[j][...]
        const Word wt_wrd = wt_mem[j][...];

        Word x = wt_wrd ^ in_wrd;

        // count_set bit for 64 bits
        AccumType sum_b = 0;
        for (int b = 0; b < WORD_SIZE; ++b) {
            sum_b += (x[b]==0) ? 1 : -1;
        }

        sum_m[j] = sum_b;
    }

    // Sum products from across convolvers
    for (int j = 0; j < CONVOLVERS; ++j)
        sum += sum_m[j];
}

return sum;
```

Figure 3.3: HLS code for binary dot product engines in the Bin-FC unit — the code computes $N =$ CONVOLVERS number of binary dot products per cycle in a pipelined fashion.

3.3.2 MILP Practicality

Table 3.2 shows the size of each benchmark as well as the runtime of the MILP for each benchmark using both MILP-base and MILP-map. We noted that the runtime scaled primarily with the number of unique constraints, which is based on the total number of cuts enumerated in the DFG. Thus MILP-map is much slower than MILP-base.

Resource-constrained scheduling is known to be NP-hard, and all commercial algorithms are heuristic in nature. Our data supports the idea that exact formulations are difficult to scale. Nevertheless, it seems reasonable use our mapping-aware MILP to improve design quality for small,

Table 3.2: CPLEX MILP solve time for each benchmark – runtime does not include cut enumeration or ILP construction. Zeros indicate runtime too short to be measured.

		MILP Solver Runtime (s)	
Design	**LLVM Instrs**	**MILP-base**	**MILP-map**
CLZ	387	9.0	28
XORR	2047	0.0	1622
GFMULT	86	0.0	0.0
CORDIC	304	3.7	42
MT	236	5.3	3602
AES	1809	314	3600
RS	2503	0.2	1.8
DR	282	3600	3603
GSM	324	108	3603
Mean	886	449	1789

performance critical kernels which are isolated from the rest of the design.

3.4 Conclusions and Future Work

We present an area-efficient pipeline synthesis approach for HLS which uses a word-level cut enumeration technique and an MILP formulation to perform mapping-aware modulo scheduling. We test our algorithm on a variety of kernel benchmarks and practical applications, obtaining improved LUT and FF usage compared to both a state-of-the-art commercial HLS tool and a mapping-agnostic MILP approach. While an MILP formulation is unscalable in general, we have limited the runtime of the solver to 60 minutes, which is tolerable given the non-trivial improvements in QoR. Future work includes incorporating mapping awareness into a scalable heuristic pipeline scheduling algorithm as well as investigating other logic synthesis optimizations such as exploring different logic decompositions of the circuit during mapping.

CHAPTER 4

EXPLORING STRUCTURED MATRICES FOR DNN COMPRESSION

4.1 Preliminaries

Chapter 2 discussed how low-precision fixed-point quantization can be used to make neural networks less resource-intensive, especially as applied to specialized hardware. In this chapter, the focus is DNN pruning via the removal of redundant weight connection from a neural network. Unlike quantization which reduces the precision of each weight, pruning instead reduces the total number of weights and operations. The two approaches are not completely orthogonal, as they both exploit the fact that DNNs are typically overparameterized containing many redundant features. However, DNN pruning is very different in implementation from DNN quantization. To introduce the reader to this topic we give an overview of three popular approaches to DNN pruning below.

4.1.1 Sparse Pruning

Sparse weight pruning is one of the most popular techniques for model compression. Pruning typically takes a pre-trained model and removing weights based on a metric such as magnitude [HPTD15, HPN+17], energy consumption [YCS16], or importance to the loss function [YLC+18]. Usually, the pruned model is fined-tuned for a small number of additional epochs to recover some of the accuracy lost during the pruning process [HPN+17]. Use such techniques, models such as AlexNet and VGG-16 have been compressed to just 9% and 11% of their original size, respectively, with little impact on accuracy [HPTD15]. To complement these developments, hardware accelerators targeting pruned networks have also been proposed [HLM+16, HKM+17, ZDZ+16]. While sparse compression techniques have achieved impressive parameter savings, a major drawback is that they introduce sparsity in an *unstructured* manner. This necessitates a sparse matrix representations which stores both the values of non-zeros as well as their locations in the form of indices. Unstructured sparse matrices incur additional storage and memory indirection overhead,

which directly translate to performance and area penalties in a hardware accelerator. In addition, they require their own set of sparse linear algebra packages/modules and are generally incompatible with dense implementations.

4.1.2 Depthwise Separable and Group Convolutions

One alternative approach to DNN pruning is to remove weight connections in a structured and principled manner. Early networks such as AlexNet [KSH12] and VGG [SZ15] exclusively utilize dense mappings such as convolutional (conv) or fully-connected (FC) layers that form a weight connection between every input and every output feature. A *depthwise separable convolution* decouples this into two steps: a *depthwise convolution* which only performs 2D spatial filtering on each channel, and a *pointwise convolution* which only learns to combine different channels using a 1×1 conv. cross-channel mappings. The decoupled layers together contain far fewer parameters than a traditional conv layer. In addition, depthwise separable convs divide the burden of spatial and cross-channel learning into two distinc steps to more efficiently make use of parameters and multiplies. The idea originated in a thesis by Laurent Sifre in 2014 [Sif14], and was subsequently popularized by network architectures like Xception [Cho16] and MobileNets [HZC$^+$17]. Currently, depthwise separable convs are widely used in mobile-targeted DNN models.

Within a depthwise separable, over 90% of parameters exist in the pointwise step [ZZLS17]. This is because the pointwise conv is still a dense mapping with a single 1×1 weight connecting each pair of input and output feature maps. Additional work has focused on eliminating redundant weights in this layer, with one proposal being the *group convolution*. Group convs divide input and output feature maps into equally-sized, mutually independent groups and performs a dense convolution in each group. Depthwise convs are specific cases of group convs with group size 1. Group convolutions were part of the original AlexNet, but only to facilitate training on multiple GPUs [KSH12]; their first as a building block of efficient CNNs was in ResNeXt [XGD$^+$17].

A group convolution reduces the parameter count of a pointwise conv, but the independent groups prevent the learning of certain cross-channel correlations resuling in reduced network ac-

curacy. This is an especially serious problem when a network stacks multiple group conv layers, as a channel in the first layer will never be connected to other groups in any of the subsequent layers. ShuffleNet [ZZLS17] proposes to remedy this by adding a *channel shuffle* to mix channels amongst groups, thereby improving cross-group connectivity in the network. Shuffling increases accuracy over vanilla group convs. Furthermore, it is extremely lightweight as it requires no parameters or arithmetic operations and can be implemented as a tensor transpose. Further works on channel shuffling include Interleaved Group Convolutions [ZQXW17, XWZ+18, SLLW18], which showed that when group convs are interleaved with shuffling, a specific combination of layer width and number of groups can maximize accuracy. Another work, Deep Roots [IRCC17], uses group convolutions with increasing group size deeper into the network to improve numerous existing models.

4.1.3 Structured Matrix Compression

A second alternative to DNN pruning is a line of research on neural networks with circulant or block-circulant [1] weights [CYF+15, SSK15, DLW+17, WLD+18]. An unstructured $n \times n$ matrix contains n^2 parameters, while an $n \times n$ circulant matrix contains only n unique parameters. Moreover, a circulant matrix $\mathbf{C}$ can be diagonalized by the normalized discrete Fourier matrix $\mathbf{F}$ via the *circulant convolution theorem* [Pan01]:

$$\mathbf{C} = \mathbf{F}^* \mathbf{D} \mathbf{F} \tag{4.1}$$

This result enables matrix-vector multiplies with $\mathbf{C}$ to be computed using two fast Fourier transforms (FFTs) and an element-wise product, which amounts to just $2n \log n + n$ multiplies. A circulant-weight DNN can thus achieve principled, asymptotic reductions in both storage size and computational complexity.

The circulant convolution theorem also provides a link between *structure* and *sparsity*. Equation 4.1 is essentially an eigendecomposition which shows us us that a circulant matrix (dense) is

[1] In this section, block-circulant, block-diagonal, etc. refers to matrices consisting of square sub-matrices which are circulant, diagonal, etc. This is different from the canonical definition of a block-diagonal matrix.

guaranteed to be diagonal (sparse) in the Fourier basis. The key advantage of exploiting sparsity in this manner is that the structure of the sparse matrix is known beforehand. There is no overhead to track non-zero locations, and computation can be performed with conventional dense linear algebra operations.

The circulant convolution theorem also provides a link between *structure* and *sparsity*. Equation 4.1 is essentially an eigendecomposition which shows us us that a circulant matrix (dense) is guaranteed to be diagonal (sparse) in the Fourier basis. The key advantage of exploiting sparsity in this manner is that the structure of the sparse matrix is known beforehand. There is no overhead to track non-zero locations, and computation can be performed with conventional dense linear algebra operations.

Publication	**Dataset**	**Comp. Ratio**	**Accuracy Loss**	**Model**
Cheng ICCV'15 [CYF+15]	CIFAR-10	3.75x	1.1%	3-layer CNN
	ImageNet	18.3x	0.7% Top-1/ 0.4% Top-5	AlexNet
Moczulski ICLR'16 [MDAdF16]	ImageNet	6.0x	0.76% Top-1	Caffe AlexNet
CirCNN MICRO'17 [DLW+17]	CIFAR-10	128x	<2%	"Medium-size DNN"
	ImageNet	90x	<2% Top-1	AlexNet
C-LSTM FPGA'18 [WLD+18]	TIMIT	7.9x	0.32% PER	Google LSTM
	TIMIT	15.9x	1.23% PER	Google LSTM

Table 4.1: Previous work on circulant DNNs – the first two publications applied circulant matrices to just the FC layers, while CirCNN applied block-circulant matrices to both conv and FC layers.

Imposing a circulant structure on DNN weights clearly restricts the space of learnable models, potentially degrading classification performance. However, previous work shows that circulant DNNs can in many cases reach comparable accuracy to baseline, or at least provide a compelling compression-accuracy tradeoff. We first summarize previous work on convolutional networks (Table 4.1). The earliest works to empirically test circulant CNNs were Cheng et al. [CYF+15] and Moczulski et al. [MDAdF16]; both showed that circulant weights were extremely effective at compressing the FC layers of AlexNet, with little accuracy loss. More recently, Ding et al.'s

CirCNN [DLW+17] proposed to us *block-circulant* matrices – matrices whose square sub-blocks are circulant. By applying block-circulant matrices to both conv and FC layers, they were able to increase the level of compression (but seemingly at an accuracy loss). CirCNN also provided the first hardware implementation of circulant CNNs, demonstrating improved energy efficiency over previous state-of-the-art of 16x for FPGAs and 6x for ASICs. A follow-up paper applied the same principles to LSTM compression and demonstrated results on FPGA which surpassed a state-of-the-art sparse LSTM accelerator [WLD+18].

4.1.4 Unitary Group Convolutions and HadaNet

In this chapter of the thesis, we propose the concept of *unitary group convolution* (UGConv), defined as a building block for neural networks that combines a weight layer (most commonly a group convolution) with unitary transforms in feature space. We show that group convs with channel shuffle (ShuffleNet) and block-circulant networks (CirCNN) are specific instances of UGConvs. By unifying two different lines of work in CNN literature, we gain a deeper understanding into the basic underlying idea, that group convolutions exhibit improved learning ability when performed in a transformed feature basis. Through a series of experiments, we then investigate how different transforms and UGConv structures affect the learning performance. Specifically, the contributions of this chapter are as follows:

1. We propose the concept of unitary group convolutions. We show that ShuffleNets and circulant networks, techniques from two disparate lines of research, are in fact both instances of UGConv networks. This lets us unify the conceptual insights of both works.
2. We evaluate how different unitary transforms affect learning performance. Our experiments show that when the weight layer is highly sparse (i.e., the number of groups is large), dense transforms outperform simple permutations.
3. We propose HadaNets, UGConv networks using the easy-to-compute Hadamard transform. HadaNets obtain similar accuracy as circulant networks at a lower computation complexity,

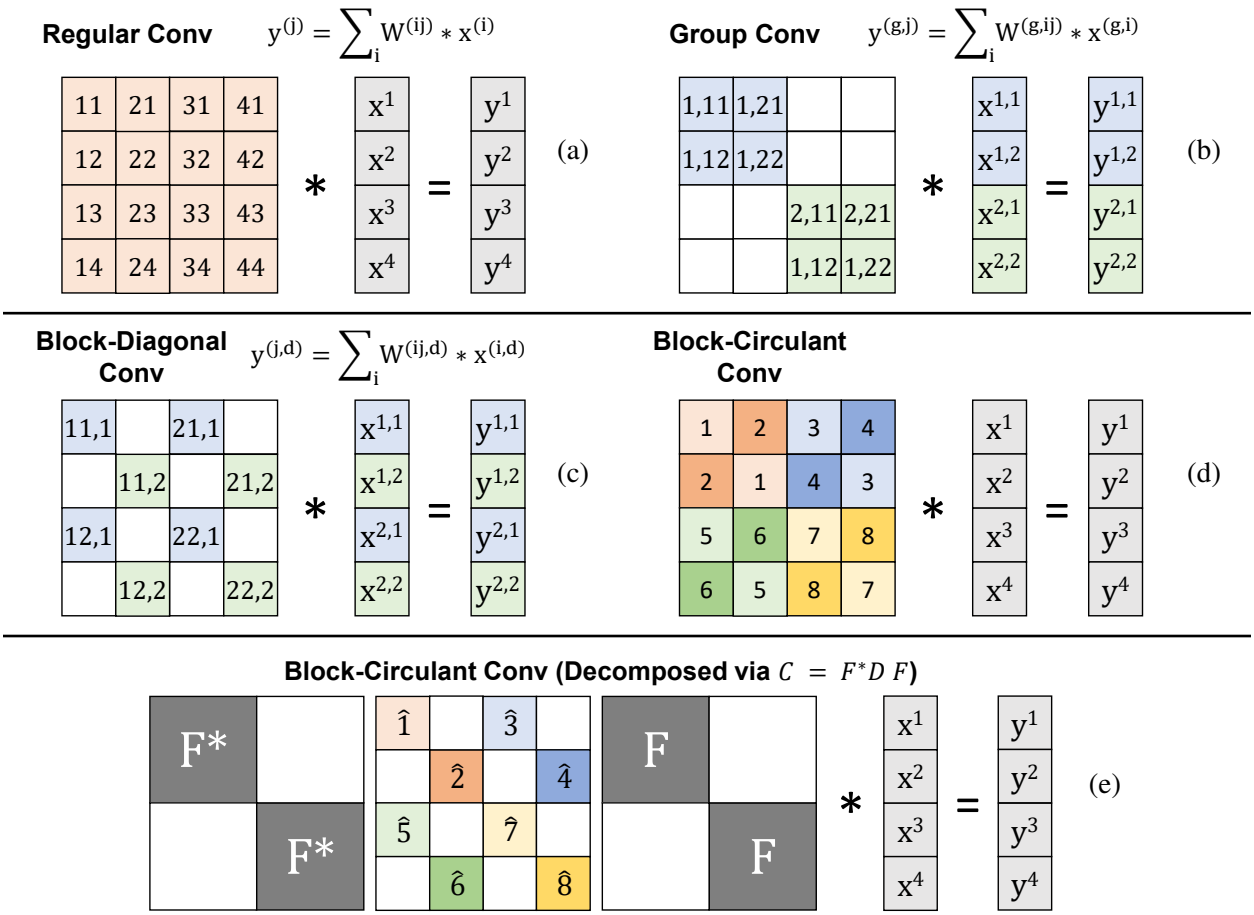


Figure 4.1: Relationship between group convs and circulant weights — each square represents a 2D feature/filter, which can be 1×1 for an FC layer. (a) regular conv layer; (b) group conv with 2 groups; (c) same group conv reordered to show the block-diagonal weight structure; (d) block-circulant conv layer; (e) same block-circulant conv decomposed into block-DFTs and a block-diagonal conv layer.

and outperform ShuffleNets with identical parameter and fpmul counts.

4.2 Unitary Group Convolutions

The basic idea of a UGConv is a group convolution sandwiched between two unitary transforms in feature space. Let $\mathbf{X}$ be an M-channel input tensor to a conv/FC layer. Each channel is a 2D feature map (for a dense layer the dimensions are 1×1). Let $\mathbf{x}^{(i)}$ denote the i'th channel in $\mathbf{X}$. Similarly, let $\mathbf{Y}$ be the N-channel output tensor, and let $\mathbf{W}$ be the weight tensor consisting of

$M \times N$ filters. We can now define an ordinary conv layer below:

$$\mathbf{y}^{(j)} = \sum_{i=1}^{M} \mathbf{x}^{(i)} * \mathbf{W}^{(ij)}, \qquad 1 \leq j \leq N$$

Figure 4.1(a) illustrates such a conv or dense layer. Note that although the figure looks like matrix multiplication, each square represents a 2D weight filter or feature map.

A group convolution is simply a collection of G disjoint convolutions (G is the number of groups). Each conv takes M/G input channels and produces N/G output channels.

$$\tilde{\mathbf{y}}^{(g,j)} = \sum_{i=1}^{M/G} \tilde{\mathbf{x}}^{(g,i)} * \tilde{\mathbf{W}}^{(g,ij)}, \quad 1 \leq j \leq N/G \tag{4.2}$$

Here g denotes the group ($1 \leq g \leq G$), and we re-index $\mathbf{x}$ and $\mathbf{y}$ with two indices (*group*, *channel in group*). Figure 4.1(b) illustrates how non-zero weights in a group conv form $\frac{M}{G} \times \frac{N}{G}$ blocks along the main diagonal. A group conv reduces parameter size and fpmuls by a factor of G relative to an ordinary conv. However, this is achieved by removing all weight connections between groups and negatively impacts learning behavior.

A UGConv recovers this lost learning ability by sandwiching the group conv between two cross-channel unitary transforms $\mathbf{P}$ and $\mathbf{Q}$ (Figure 4.2(a)). More formally, we can define a UGConv is:

$$\begin{gathered} \tilde{\mathbf{X}}_k = \mathbf{P}\mathbf{X}_k \qquad \forall k \\ \tilde{\mathbf{y}}^{(g,j)} = \sum_{i=1}^{M/G} \tilde{\mathbf{x}}^{(g,i)} * \tilde{\mathbf{W}}^{(g,ij)}, \quad 1 \leq j \leq N/G \\ \mathbf{Y}_l = \mathbf{Q}\tilde{\mathbf{Y}}_l \qquad \forall l \end{gathered} \tag{4.3}$$

For a tensor $\mathbf{X}$ containing M channels, $\mathbf{X}_k$ is defined as the M-length vector formed by taking the k'th element/pixel from each channel. $\mathbf{P} \in \mathbb{C}^{M \times M}$ and $\mathbf{Q} \in \mathbb{C}^{N \times N}$ are unitary matrix transforms applied element-wise over the input and output channels. We use tilde ($\tilde{\mathbf{x}}, \tilde{\mathbf{y}}, \tilde{\mathbf{W}}$) to indicate tensors in the transformed feature space. Note that: (1) $\mathbf{P}$ and $\mathbf{Q}$ can be identity transforms, and thus UGConv includes group convolutions; (2) unitary transforms preserve inner products, thus they should not diminish gradient magnitudes in the network; (3) UGConv can also be applied to FC layers (using 1×1 feature maps and a 1×1 group conv).

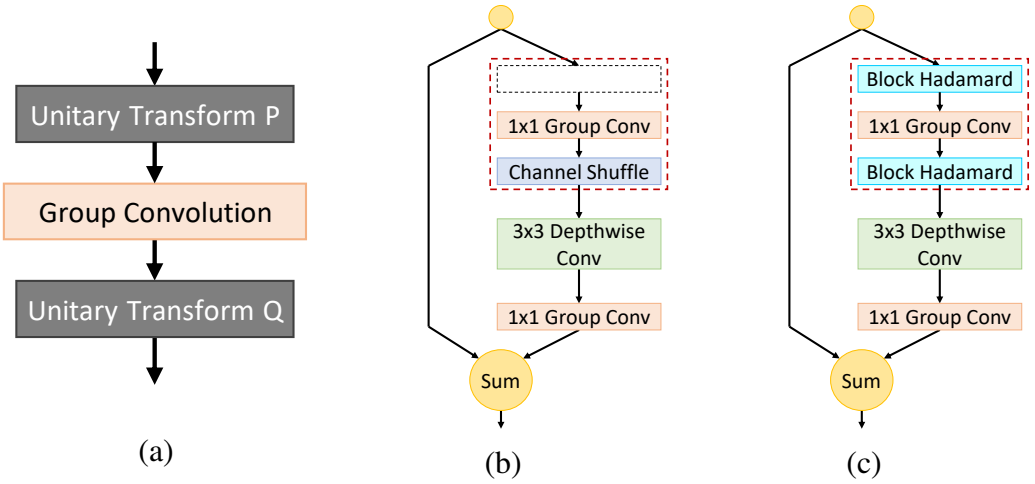


Figure 4.2: UGConv, ShuffleNet, and HadaNet block architectures — (a) a general block for unitary group convolutions; (b) a ShuffleNet block reproduced from the original paper [ZZLS17]; (c) our proposed HadaNet variation. Note that both ShuffleNet and HadaNet blocks contain the UGConv pattern.

One key point to make is the equivalence between a *group conv* and a convolution with *block-diagonal weights* (weights that consist of sub-blocks of square diagonal matrices). Figure 4.1(c) shows a block-diagonal conv, which visually already looks identical to the group conv in Figure 4.1(b). More formally, divide $\mathbf{X}$ and $\mathbf{Y}$ into size $D \times 1$ sub-blocks, and W into $D \times D$ sub-blocks which are diagonal. Let i index the input sub-blocks ($0 \leq i \leq M/D-1$), j index the output sub-blocks ($0 \leq j \leq N/D-1$), and d index the channels within each sub-block ($1 \leq d \leq D$). We can express the block-diagonal conv as follows:

$$\mathbf{y}^{(j*D+d)} = \sum_{i=0}^{M/D-1} \mathbf{x}^{(i*D+d)} * \mathbf{W}^{(i*D+d \;\; j*D+d)}$$

Only D convs need to be performed for each $D \times D$ sub-block because they are diagonal. Similar to Equation 4.2, we can simplify notation by re-labeling using a tuple (*sub-block*, *channel in sub-block*). This removes the multiplies by D and allows i and j to start from 1. Then:

$$\mathbf{y}^{(j,d)} = \sum_{i=1}^{M/D} \mathbf{x}^{(i,d)} * \mathbf{W}^{(ij,d)}, \quad 1 \leq j \leq N/D \tag{4.4}$$

It is easy to see that Equation 4.4 matches Equation 4.2.

4.2.1 UGConv and ShuffleNet

ShuffleNet is a variant of the MobileNets architecture in which the pointwise convolutions (which take up 93.4% of the multiply-accumulate operations [ZZLS17]) are converted into group convolutions. However, when multiple group convs are stacked together, the lack of connections between groups over many layers prevents the learning of cross-group correlations. To address this, ShuffleNet shuffles the output channels groups in a fixed, round-robin manner. For each group, the first channel is shuffled into group 1, the second channel into group 2, etc. This shuffle can be expressed as a permutation in feature space, and ShuffleNets are thus an example of of UGConvNets where **P** is identity and **Q** is a fixed permutation matrix.

ShuffleNet shows experimentally that it is beneficial to shuffle information across groups when stacking group convs. However, shuffling channels is not the only way to accomplish such information mixing.

4.2.2 UGConv and Circulant Networks

Circulant and block-circulant neural networks [DLW+17, WDL+18] utilize layers that impose a block-circulant structure on their weight tensors. For an FC layer, the 2D weight matrix is made to be circulant. For a conv layer, the circulant structure is applied over the input and output channels axes. That is to say, given a 4D convolutional weight tensor with shape `(height, width, in_channels, out_channels)`, each 2D slice of this tensor `[i,j,:,:]` becomes circulant.

Figure 4.1(d) shows a block-circulant layer where each 2×2 sub-block of the weight tensor is circulant. By Equation (4.1), each $D \times D$ circulant matrix can be decomposed into a D-length DFT, a diagonal matrix, and a corresponding IDFT. In Figure 4.1(e), each $D \times D$ sub-block is diagonalized in this fashion. We use tilde to indicate weight values in the DFT-transformed space. The resulting weight structure is block-diagonal, and the weight layer sits between two block-DFT transforms. We know from the previous section that block-diagonal weights correspond to group convolutions. Therefore, *a block-circulant layer is just a group convolution in a transformed*

feature space. This of course falls within the definition of a UGConv, with $\mathbf{P}$ and $\mathbf{Q}$ being block-DFT/IDFT transforms. Note that these DFTs are applied along the channels, and so circulant networks are *not* examining the spatial frequency components of the image.

We make a few additional notes about block-circulant layers. First, the size of the circulant blocks D is equal to the *number of groups* in the equivalent group conv (not the group size). Thus each D-length DFT touches a single channel in every group, fully mixing information between groups. Second, though our example uses a "square" weight tensor (where $M = N$), non-square block-circulant tensors can be diagonalized as well. As long as both M and N are divisible by D, the 'rectangular' weight tensor can be divided into $D \times D$ blocks. In this case, $\mathbf{P} \in \mathbb{C}^{\mathbf{M}\times\mathbf{M}}$ is not the inverse of $\mathbf{Q} \in \mathbb{C}^{\mathbf{N}\times\mathbf{N}}$, but each sub-block along the diagonal of $\mathbf{P}$ is the inverse of the corresponding sub-block in $\mathbf{Q}$. We say that $\mathbf{P}$ is the *block-inverse* of $\mathbf{Q}$.

Because $\mathbf{P}$ and $\mathbf{Q}$ are block-inverses, if we directly stack multiple such blocks many of the transforms will cancel out. However, practical DNNs include batch norm and/or nonlinearities between linear layers. The block-DFTs (and orthogonal transforms in general) do not commute with channel-wise or pointwise operations, which prevents trivial cancellation. However, note that channel shuffles *do* commute and cancel out in this manner.

4.2.3 Discussion of UGConvs

We have provided two specific examples from literature (ShuffleNet [ZZLS17] and CirCNN [DLW$^+$17]) which combine a structured sparse weight layer (group convolution) with unitary transforms. The transforms help to improve cross-channel representation learning without adding additional parameters. However, the two techniques have important differences. ShuffleNet's permutations are very lightweight as they require no arithmetic operations. However, permutations do not affect the sparsity of weight layer. On the other hand, CirCNN composes block-DFTs with a group conv to create an effective weight structure (i.e., circulant weights) which is dense. Moreover, it does so with lower asymptotic computational complexity than unstructured dense weights.

We hypothesize that the representation learning capability of a UGConv layer is a function of

Table 4.2: Hadamard vs. Discrete Fourier transforms — The entries of the DFT matrix are the complex roots of unity. The entries of the Hadamard matrix are $+1$ or -1. The last column shows the structure of $\mathbf{P}^*\mathbf{DP}$ where $\mathbf{D}$ is a diagonal matrix and $\mathbf{P}$ is the transform; differences are bolded.

	Fourier	**Hadamard**
Transform $\mathbf{P}$	$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix}$	$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$
Structure of $\mathbf{P}*\mathbf{DP}$	$\begin{bmatrix} a & b & c & d \\ d & a & b & c \\ c & d & a & b \\ b & c & d & a \end{bmatrix}$	$\begin{bmatrix} a & b & c & d \\ \mathbf{b} & a & \mathbf{d} & c \\ c & d & a & b \\ \mathbf{d} & c & \mathbf{b} & a \end{bmatrix}$
FP Muls	$n \log n$	0
FP Adds	$n \log n$	$n \log n$

both the sparsity of the weights as well as that of the transform. An unstructured dense weight layer offers the best learning capability; grouping introduces sparsity and degrades cross-channel learning performance, some of which can be recovered via transforms. Because dense transforms create dense weight structures, we believe they enable learning a richer set of representations compared to sparse transforms (i.e., channel shuffling). When the weight sparsity is low, the difference between the two may be negligible in terms of network accuracy. However, we expect dense transforms to outperform shuffling when the number of groups is large.

Another difference is that ShuffleNet applies channel shuffle on only one side of the weight layer, while CirCNN effectively applies transformations on both sides. We use the terms 1-sided and 2-sided UGConvs to refer to these two cases, and test both in our experiments.

4.2.4 The Hadamard Transform

One drawback of dense transforms such as DFT is that they require more computational overhead as compared to shuffling. Even using the 'fast' algorithm, each $n \times n$ DFT requires $O(n \log n)$ floating-point multiplies and adds. Furthermore, the fact that the DFT uses complex numbers may further complicate software/hardware implementations. Finally, the DFT is taken over the channels

Table 4.3: UGConv test error on a toy MNIST network — a 'G' in the layer width columns indicates a group layer. In the transform columns, **P** and **Q** denote 1-sided pre-conv and post-conv transforms, respectively; **PQ** denotes a 2-sided transform. All values are averaged over 5 runs and 90% confidence bounds for each value are at most ±5%.

Layer Width			**Transform**						
L2	**L3**	**L4**	**None**	**Rand Ortho**			**Rand Perm**		
Conv3x3	FC	FC		P	Q	PQ	P	Q	PQ
20	20,G	10	6%	**4%**	**4%**	**4%**	5%	6%	5%
20	20,G	10,G	27%	10%	8%	**4%**	27%	26%	25%
20,G	20,G	10	25%	**10%**	**10%**	**10%**	27%	20%	21%
20,G	20,G	10,G	60%	23%	**17%**	20%	57%	55%	57%

where there is no spatial structure; the transform exists purely to mix information across channels. Given this, we would like to find a more efficient alternative.

The Hadamard transform [PKA69] is defined as a matrix containing only $+1/-1$ elements and whose rows and columns are mutually orthogonal. Table 4.2 shows a 4×4 Hadamard matrix. Because all coefficients have magnitude 1, the transform can be computed without multiplies, using adds/subtracts only. This is extremely important as floating-point multiplies are typically the computational bottleneck for DNN computation on both GPUs and specialized hardware. By replacing multiplies with adds, the FHT will be much faster than the FFT in specialized hardware such as ASICs and FPGAs. In addition, the Hadamard transform can be generated recursively like the Fourier transform, meaning that a fast Hadamard transform (FHT) exists similar to the FFT to compute a n-length Hadamard transform in $O(n \log n)$ adds/subtracts [PKA69]. The recursive nature of FHT also enables Hadamard kernels to be implemented without explicitly storing the matrix itself; instead the matrix can be generated on the fly (similar to existing implementations of FFT kernels). This means neither FHT nor FFT requires storing additional parameters.

Hadamard is more efficient that DFT, but does it achieve the same learning performance? There is some high-level intuition that this would be the case: Table 4.2 compares the weight structure imposed by $\mathbf{P}^*\mathbf{DP}$ when **P** is DFT and Hadamard. DFT results in a circulant matrix; Hadamard results in a nearly identical weight matrix with only a few different elements. We hypothesize

that there will be no accuracy impact in replacing circulant weights with Hadamard-diagonalizable weights in neural nets.

We further speculate that dense unitary transforms in general, including DFT and Hadamard, achieve comparable learning performance. This is again because the ordering of channels in DNNs is essentially random (i.e., the channel order encodes no useful information), meaning there are no patterns that can be exploited by one particular cross-channel transform and not others. The transforms in UGConv exist solely to connect different channel groups, and any dense transform will work as well as another. To test this hypothesis, we experiment with randomly generated orthogonal transforms in addition to DFT and Hadamard.

One drawback of Hadamard transforms is that they have only been proven to exist for some specific power-of-two lengths [Wal76, Đok08, Hor12], while the DFT exists for all even lengths. This restricts the group size in a Hadamard UGConv. However, we believe this is not a serious issue in practice as power-of-two widths and group sizes are extremely common [HZRS15, XGD+17, HZC+17, Cho16, SZ15].

4.3 Experimental Validation

We first present ablation studies on a toy MNIST network followed by deeper CIFAR-10 models. These experiments build up insights on UGConv. We then demonstrate the utility of Hadamard using grouped ResNets and a ShuffleNet model from literature trained on ImageNet.

4.3.1 Dense Transforms vs. Shuffle

Our first experiment uses a toy MNIST network. This allows us to isolate the UGConv block and to compare dense orthogonal transforms versus permutations in a simple setting. We stress that the goal here is *not* to build a realistic classifier. The layer architecture is denoted below, where each layer is described as (*number of channels*)(*layer type*):

$$10\text{Conv3x3} - 20\text{Conv3x3} - \mathbf{20FC} - 10\text{FC}$$

We perform 2×2 max pooling before each 3×3 conv layer, and a global average pool before the first FC layer. Each layer is followed by batch normalization and ReLU.

We convert the first FC layer of the network (20FC1, shown in bold) into a UGConv block (it becomes a grouped FC with transforms). The group number is equal to the number of channels to maximize sparsity. From this base architecture we derive three variations: (1) convert the preceding Conv3x3 layer into group conv; (2) convert the following FC layer into group FC; (3) convert both surrounding layers into group layers. These test the performance of transforms in the context of stacked group layers. Two types of transform are evaluated: randomly generated dense orthogonal and random permutation transforms. We test with both 1-sided (using one of $\mathbf{P}$ or $\mathbf{Q}$ and setting the other to identity) and 2-sided UGConvs ($\mathbf{P} = \mathbf{Q}^{-1}$). All results are averaged over five runs, and we regenerate the random transformation matrices between runs.

Table 4.3 shows our results. Due to the small size of the network, the 90% confidence bound for these values can be as large as $\pm 5\%$. Nevertheless, differences between transforms are clearly demonstrated. When L3 is the only grouped layer in the network (row 1), transforms have little to no effect. However, when two or more group layers are stacked together, the dense orthogonal transforms achieve improved accuracy. Permutations did not improve accuracy in any experiment. This is a clear (albeit artificial) demonstration that when the number of groups is very large, dense transforms outperform permutations in learning ability.

Another interesting observation is that there is little difference between 1-sided and 2-sided transforms, regardless of whether the UGConv block is stacked before or after another group layer. For example, in Table 4.3 row 3, a dense orthogonal transform improves accuracy even when it is placed *after* both group layers. It may be surprising that a transform affects layers preceding it. But keep in mind that the transform also affects gradients on the backwards pass, allowing the same weights to 'see' more downstream activations during backpropagation. Alternatively, we can view the UGConv layer as a learnable structured weight layer. From this perspective, the weight structure is a function of transforms both before or after.

Table 4.4: UGConv test error on CIFAR-10 — The first three columns show the number of groups used in the three stages (S1-S3). The **Base** column shows the test error with no transforms, and the other columns show *improvement* in test error over this baseline. Some entries are blank due to insufficient time to complete the experiments.

	# of Groups			**Base**	**1-sided Transforms**			**2-sided Transforms**				**Params**
	S1	S2	S3		Shuffle	Hada	Ortho	Shuffle*	Fourier	Hada	Ortho	
ResNet-20	4	8	16	19.5%	3.3%	4.0%	4.0%	3.1%	4.1%	**4.2%**	3.8%	25K
	8	16	32	23.8%	2.9%	4.3%	3.9%	4.1%	**5.4%**	**5.4%**	5.3%	14K
ResNet-56	4	8	16	16.0%	4.0%	4.4%	4.2%	4.0%	**4.7%**	4.5%	4.6%	76K
	8	16	32	20.6%	5.4%	6.1%	6.4%	5.8%	7.1%	**7.2%**	6.8%	41K
ShuffleNet-29	4	8	16	18.3%	2.7%	2.4%	3.1%	3.8%	**4.9%**	4.5%	4.2%	23K
	8	16	32	22.1%	0.6%	3.4%	3.6%	3.8%	5.1%	5.0%	**5.3%**	17K
ShuffleNet-56	4	8	16	16.2%	3.6%	3.5%	3.4%	3.9%	4.6%	4.5%	**4.7%**	41K
	8	16	32	19.7%	4.3%	4.4%	4.9%	5.2%	**6.0%**	**6.0%**	**6.0%**	29K
Mean	4	8	16	17.5%	3.4%	3.6%	3.7%	3.7%	**4.6%**	4.4%	4.3%	
	8	16	32	21.5%	3.3%	4.6%	4.7%	4.7%	**5.9%**	**5.9%**	**5.9**%	

4.3.2 Evaluation of Different Transforms

We have shown that dense orthogonal transforms can improve over shuffles in small DNNs with large group sizes. To validate our results on more realistic architectures, we perform experiments on CIFAR-10 [KH09] using ResNet [HZRS15]. We use UGConvs to replace the two 3×3 convolutions in each ResNet block, and to replace the 1×1 projection layers. ResNets are divided into three stages (S1, S2, S3), with later stages having more channels. We use more groups in later stages, keeping the ratio of channels to groups constant. Two models are tested: **ResNet-20** (3 block per stage) and **ResNet-56** (9 blocks per stage). We also experiment with the same high-level architecture but using the building block from ShuffleNet [ZZLS17]. This block which contains two 1×1 convs and a 3×3 depthwise conv (see Figure 4.2(b)). Following ShuffleNet we apply transformations around the first 1×1 group conv only and make no changes to the second group conv. Again, two models are tested: **ShuffleNet-29** (3 block per stage) and **ShuffleNet-56** (6 blocks per stage).

We use layer widths and training hyperparameters from [HZRS15] and make use of standard data augmentations: padding 8 pixels on each side and randomly cropping back to original size, combined with a random horizontal flip [HZRS15, HSL$^+$16, LCY13]. Each network is trained for 200 epochs, and we report the mean test error over the last 5 epochs.

We test the following transforms: identity (None), ShuffleNet permutation (Shuffle), block-Hadamard (Hada), block-DFT (Fourier), and block-random-orthogonal (Ortho). The block transforms follow the same structure described in Section 4.2.2. For each transform, both 1-sided (letting $\mathbf{Q}$ be the transform and $\mathbf{P}$ identity) and 2-sided ($\mathbf{P}$ and $\mathbf{Q}$ are block-inverses) versions are tested where reasonable. The 1-sided DFT is left out because it introduced complex numbers into the network. For the 2-sided channel shuffle (Shuffle*), we set $\mathbf{P} = \mathbf{Q}$ to essentially perform additional shuffling; this is done since using block-inverse shuffles will lead to trivial cancellation. All results are displayed in Table 4.4. The error rate with no transforms is given first followed by the accuracy improvement achieved with each UGConv setup. Our base error rates are high for CIFAR-10 because group convolutions significantly compress the network

Table 4.5: ShuffleNet and HadaNet classification error on ImageNet — we include data on both the original ShuffleNet (with our own code) and our pre-activation variation. Our baseline ShuffleNet implementation is close to the literature results (52.7%). For each model we show the number of parameters and fpmuls, as well as the overhead in additions from the Hadamard transform.

	Shuffle	**Hada**	**Delta**	**Params**	**FPmuls**	**Hada Adds**
ResNet-18 g8	46.4%	**44.6%**	(-1.8%)	1.9M	330M	7.8M
ResNet-18 g16	55.8%	**52.3%**	(-3.5%)	1.2M	226M	10.4M
ShuffleNet-x0.25 g8	53.6%	**52.6%**	(-1.0%)	0.46M	17M	0.95M

A key result here is that dense orthogonal transforms perform similarly in accuracy. Fourier, Hada, and Ortho obtain results which are within a spread of 0.4% in both 1-sided and 2-sided settings. On the other hand, the shuffle transforms (1 and 2-sided) clearly perform worse for the larger group sizes. This confirms our hypothesis that Hadamard is comparable to DFT in learning performance while being much easier to compute. It also provides evidence that *all* dense UGConvs achieve comparable learning performance.

Another observation is that 2-sided transforms significantly outperform their 1-sided variants, which is different from the MNIST data. We currently do not have an explanation for this effect. One speculation was that 2-sided transforms perform better when the number of input and output channels did not match. However, further testing with the small MNIST network showed that this was not the case.

Finally, note that the accuracy trends remained the same whether the transforms were applied to 3×3 group convs in ResNet or 1×1 group convs in ShuffleNet. This is evidence that spatial and cross-channel dependencies are effectively decoupled in convolutional layers, and that the size of the filter does not significantly affect channel-space transforms.

4.3.3 HadaNet Evaluation on ImageNet

The data from previous sections point to two regimes: at low weight sparsity (i.e., small group numbers) a simple shuffle is sufficient to maximize accuracy. At large group numbers, however,

dense transforms outperform shuffles. This section evaluates the 2-sided block-Hadamard transform against shuffle on ImageNet. Hadamard was chosen as it is far more efficient than other dense unitary transforms, and ShuffleNet was used for comparison as it is highly related work and a strong baseline. We refer to networks using Hadamard UGConvs as HadaNets. Figure 4.2 compares the residual blocks of ShuffleNet and HadaNet.

Due to hardware constraints, we chose small models with fairly large group size. This is the setting where dense transforms should perform the best compared to shuffle. We evaluate ResNet-18 following the ImageNet architecture from [HZRS15] and using group sizes 8 and 16 throughout the network. We also test with the ShuffleNet-x0.25 g8, which is the smallest ShuffleNet variant from [ZZLS17]. This network has 50 layers and also uses 8 groups. Each network was trained with the hyperparameters and learning rate schedule described in their respective papers. We compare 1-sided shuffle to 2-sided block-Hadamard (note that ShuffleNet from literature already contains the 1-sided shuffle). All results are displayed in Table 4.5. Our reproduction of ShuffleNet-x0.25-g8 achieved a Top-1 error of 53.6%, which is close to the 52.7% reported in Table 2 of [ZZLS17].

The results demonstrate that the Hadamard transform can indeed outperform shuffling in terms of accuracy on large scale datasets. ResNet-18 with group convs is a non-standard model, but it serves to show that the trends observed in CIFAR-10 ResNets carry over to ImageNet. On the other hand, ShuffleNet is a well-optimized baseline which obtains good accuracy performance on a very tight parameter and fpmul budget. In addition, despite very little hyperparameter tuning, HadaNet was able to improve slightly over ShuffleNet.

4.3.4 Practicality of HadaNet

HadaNet slightly outperforms ShuffleNet on accuracy, but requires extra floating-point adds. An N-channel group conv with B groups requires N^2/B fpmuls for the weight layer and $2N \log B$ adds for the two block-Hadamard transforms. Compared to multiplies, additions are already much cheaper in hardware. The last column of Table 4.5 shows the number of additions needed for each network if the fast Hadamard transform is used. The relative overhead of HadaNet is fairly small:

the extra adds amount to only 2-5% of existing multiply-accumulates in those networks.

However, the overhead of the Hadamard transform depends on a well-optimized implementation. The reason we did not show runtime on GPU is that an $O(n \log n)$ fast Hadamard kernel operating along the channels is not currently available. As a result our own HadaNet implementation is fairly slow.

On the other hand, we believe the Hadamard transform might be useful for specialized DNN accelerators implemented with FPGAs [CFO+18] or ASICs [JYP+17]. Top computer hardware conferences already contain works demonstrating the use of circulant matrices for DNN compression in dedicated hardware [DLW+17, WLD+18, DLX+19]. These works show that DFTs can be very efficiently implemented in a dedicated module due to its recursive nature. We choose Hadamard because it also has the same recursive properties, meaning it should be even simpler in hardware due to lower computational complexity. All-in-all, this chapter reveals that in high weight sparsity regimes, dense transforms outperform simple shuffling. HadaNet is more efficient than the existing state-of-the-art dense transform (the DFT transform) while achieving similar accuracy performance in DNNs.

4.4 Conclusions and Future Work

We introduce the concept of unitary group convolutions, a composition of group convolutions with unitary transforms in feature space. We use the UGConv framework to unify two disparate ideas in CNN literature, ShuffleNets and block-circulant networks, and provide valuable insights into both techniques. UGConvs with dense unitary transforms demonstrate superior ability to learn cross-channel mappings versus ordinary and shuffled group convolutions. Based on these these observations we propose HadaNet, a variant of ShuffleNet that improves accuracy on the ImageNet dataset without incurring additional parameters or floating-point multiplies.

One direction for future work would be investigating *higher-order* Hadamard-diagonalizable matrices in deep neural networks. There are already a few studies which describe *low-displacement*

rank matrices (a generalized form of circulant matrices which admit fast algorithms using the DFT) and their potential for DNN compression [SSK15]. We believe it would be possible to similarly derive a more general class of matrices which utilize the fast Hadamard transform. Such matrices would have the same advantages of vanilla Hadamard networks over vanilla circulant networks shown in this chapter.

A different future direction is to replace the Hadamard transform with a trained $0, +1, -1$ transform; training may allow the transform to adapt to the weights, and introducing zeros could enable sparse compute reduction.

CHAPTER 5

OUTLIER CHANNEL SPLITTING FOR DATA-FREE DNN QUANTIZATION

5.1 Preliminaries

Fixed-point quantization is critical to the design of high performance specialized neural network accelerators. The majority of literature on DNN quantization involves training. A DNN model is either trained from scratch with quantized values [CBD15, WLCS18, JKC+18] or a pre-trained floating-point model is *fine-tuned* with quantized values for a small number of epochs [HMD16, ZYG+17]. Training is important because it essentially lets the each layer's weights adapt to the newly-quantized inputs. With fine-tuning, a variety of CNNs for ImageNet classification can run inference with 8-bit integers [Mig17]. Although training-based techniques are versatile and valuable, there are important real-world scenarios in which (re)training is not applicable. Consider an ML service provider (e.g., Amazon, Microsoft, Google) which wants to run a black-box client model on a specialized accelerator utilizing low-precision fixed-point. Machine learning research works almost exclusively with floating-point models trained on GPUs (save for a tiny fraction of hardware-oriented research), so one of the parties must quantize the model first. It would be natural for the service provider to perform quantization for its custom machines, but the client often does not want to share its proprietary training data. On the other hand, the client (possibly a small organization or an individual) may lack the the expertise or manpower for quantization training. Finally, the DNN may be an off-the-shelf model (for which data is not available) or part of a legacy data pipeline (for which the model is extremely difficult to modify). The above is just a small list of reasons why industry would prefer *data-free* (DF) quantization, i.e., quantization methods that do not require (re)training. The importance of DF quantization can be seen from NVIDIA's TensorRT, a product which specifically performs 8-bit integer quantization using just a small collection of sample inputs. The chapter of the thesis presents a technique which can generically improve data-free DNN quantization at the cost of slight model size overhead.

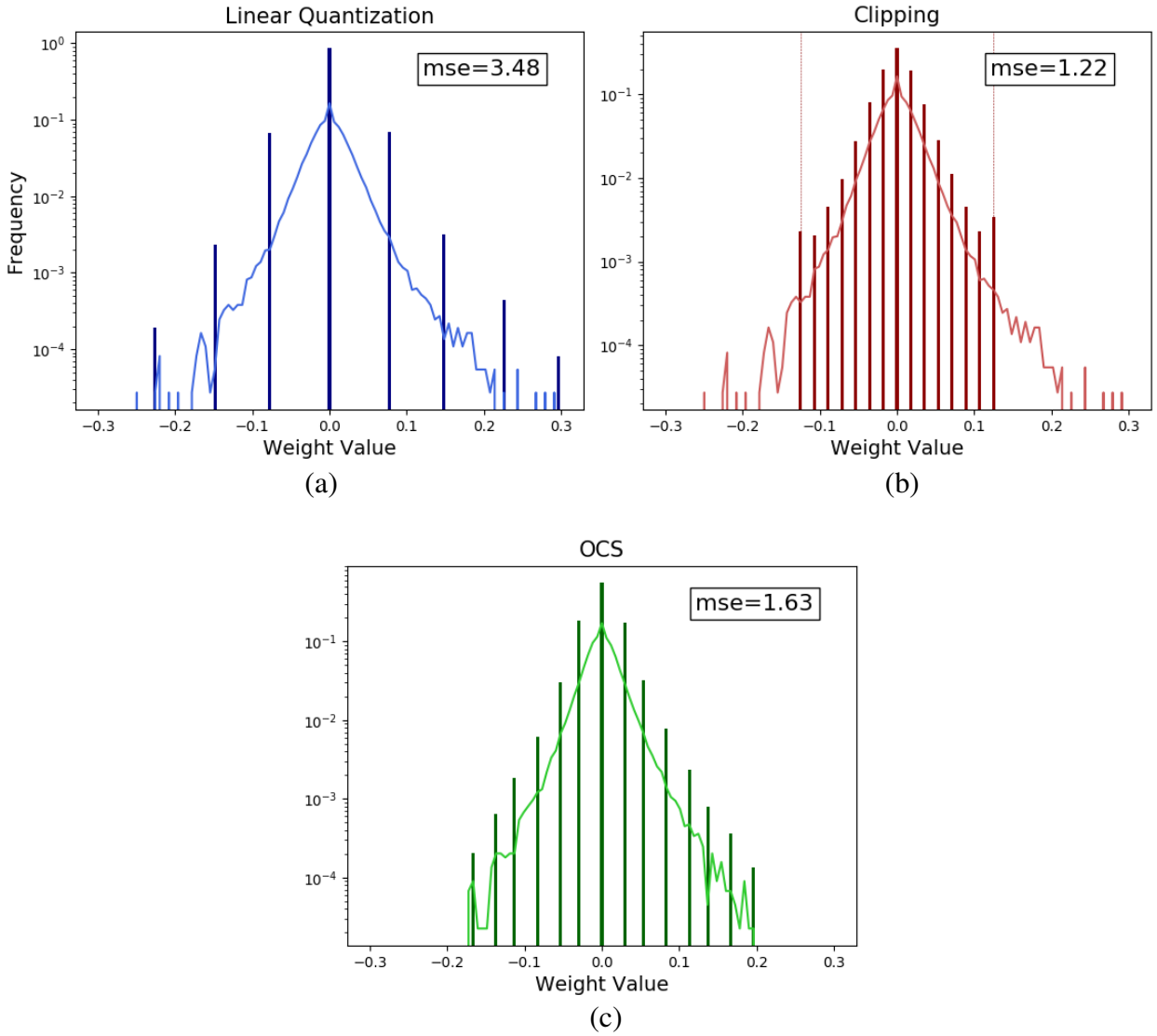


Figure 5.1: Weight histograms for linear, clipping, and OCS quantization — light color is for the floating-point histogram and dark color indicates is in the light color while the quantized weight histogram is in dark color. Both clipping and OCS reduces mean squared quantization error (MSE) by reducing the dynamic range of the distribution. Clipping reduces the overall MSE but greatly distorts the outliers. OCS avoids this distortion by splitting the outliers instead, moving them towards the center of the distribution at the cost of some model size overhead.

5.1.1 Data-Free Quantization

DNN weights and activations follow a bell-shaped distribution after training. However, commodity hardware uses a linear number representation with evenly-spaced grid points. The naïve approach is to linearly map the entire range of the distribution to the range of the quantization grid

(Figure 5.1(a)). Here the grid points extend to the maximum value in the distribution [HCS+17]. Clearly, this method over-provisions grid points for the rarely-occurring *outliers*. A better approach is to make the grid narrower than the distribution. This is known as *clipping*, as it is equivalent to thresholding the outliers before applying linear quantization (Figure 5.1(b)). Empirically, clipping can improve the accuracy of quantized DNNs, and many techniques exist to choose the optimal clip threshold [SSH15, ZST+18, Mig17]. Unfortunately, clipping can only reduce overall quantization error by increasing the distortion on the outliers — it is constrained by this tradeoff.

Another approach to handling outliers is to quantize them separately from the central values. Such outlier-aware quantization [PKY18, PYV18] is highly effective, but involves the use of dedicated non-commodity hardware.

5.1.2 Outlier Channel Splitting

In this chapter of the thesis, we propose *outlier channel splitting* (OCS). OCS identifies a small number of channels containing outliers, duplicates them, then halves the values in those channels. This creates a functionally identical network, but moves the affected outliers towards the center of the distribution (Figure 5.1 (c)). OCS takes inspiration from previous work on equivalence-preserving network transformations [CGS16]; it does not require retraining and can be used on commodity CPUs and GPUs. Complementary to clipping, OCS introduces a new tradeoff: it reduces quantization error at the expense of making the neural network larger. Experimental evaluation shows that for practical CNN and RNN models, OCS can significantly improve post-training quantization accuracy over state-of-the-art clipping methods with just a few percent overhead.

To present a comprehensive study of post-training quantization, this chapter also evaluates different techniques for optimizing the clip threshold on both weights and activations. To our best knowledge, we are the first to perform a detailed literature comparison of these techniques. The specific contributions of this chapter are as follows:

1. We propose outlier channel splitting, a technique to improve DNN model quantization that

does not require retraining and works with commodity hardware.

2. We present a comprehensive evaluation of post-training clipping techniques found in literature. To our best knowledge this is the first such study.

3. We demonstrate that OCS can outperform state-of-the-art clipping techniques on weight quantization, while incurring negligible overheads.

Code for both OCS and clipping is available in open source [1].

5.2 Prior Work on Data-Free Quantization

5.2.1 Linear Quantization

The simplest form of linear quantization maps the inputs to a set of discrete, evenly-spaced grid points which span the entire dynamic range of the inputs. The maximum quantization error for any single value is one-half of the increment. For symmetric k-bit quantization, we have $2^k - 1$ grid points:

$$\textbf{LinearQuant}(\mathbf{x}) = \text{round}\left(\frac{\mathbf{x}\,(2^{k-1}-1)}{\max(|\mathbf{x}|)}\right)\frac{\max(|\mathbf{x}|)}{2^{k-1}-1} \tag{5.1}$$

Because each value is scaled by $\max(\mathbf{x})$, **LinearQuant** is sensitive to the largest inputs, the outliers. Many existing works first clip the range of $\mathbf{x}$ prior to linear quantization. Clipping represents the state-of-the-art in post-training quantization, and in the paragraphs below we give a brief overview of different methods for optimizing the clip threshold in literature. We also present an evaluation of these methods in our experiments.

5.2.2 Mean Squared Error Clipping

This method chooses a clip threshold which minimizes the mean squared error (MSE) or L2-norm between the floating-point and quantized values [SSH15, SHS16]. It first constructs a his-

[1] https://github.com/cornell-zhang/dnn-quant-ocs

togram of the floating-point values. Let x_i and $h(x_i)$ be the bin values and frequencies, and $i = 1 \ldots n$ denote the n bins. The MSE is defined as:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} h(x_i) * (x_i - Q(x_i))^2 \tag{5.2}$$

where $Q(x)$ is the quantization function. In our experiments, we generate a large number of candidate clip thresholds evenly spaced between 0 and the max absolute value, and choose the one with minimal MSE.

5.2.3 ACIQ Clipping

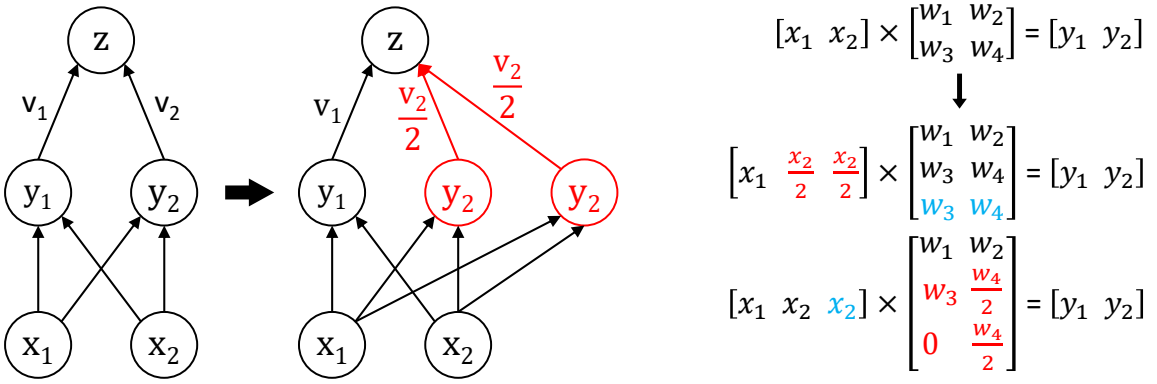
Proposed by Banner et al. [BNHS18], ACIQ first determines whether distribution is closer to a Gaussian or a Laplacian. Using statistics from the appropriate distribution, it uses an (approximate) closed-form solution for the clip threshold which minimizes MSE. Compared to the MSE method above, ACIQ avoids sweeping candidate thresholds and is much faster. This allows the clip threshold to be adjusted between input batches for activation quantization.

We used open-source code from the authors [2]. Banner et al. assumed that an m-bit fixed-point format contains 2^m grid points; this representation lacks a grid point at zero for signed values. We use $2^m - 1$ grid points (a sign-magnitude representation) instead as it is the default in our framework, and slightly adjusted the formulas from the paper to suit.

5.2.4 KL Divergence Clipping

This method chooses a clip threshold which (approximately) minimizes the KL divergence between the floating-point and quantized. Similar to the MMSE method, it works on the histogram of values and selects the optimal clip threshold from a set of candidates. The method was first proposed in a set of slides on NVIDIA's TensorRT [Mig17], which unfortunately does not contain enough technical detail for replication. Instead, we adapted an open-source implementation from Apache MXNet [CLL+15].

[2] https://github.com/submission2019/AnalyticalScaleForInteger Quantization



(a) (b)

Figure 5.2: OCS network transformation — after duplicating a neuron, we can divide either the neuron's output value or its outgoing weights in half to preserve functional equivalence. (a) figure taken from Net2Net [CGS16] where the weight v_2 is halved by duplicating y_2; (b) an example with multiple inputs and multiple outputs, showing how x_2 or w_3 can be halved while maintaining the same outputs. Here, an entire row must be added to the weight matrix to split a single value.

In general, floating-point and quantized distributions do not have the same support and the KL divergence is thus undefined. To get around this, the MXNet implementation smooths the quantized histogram slightly by moving some of the probability mass into zero-frequency bins.

5.3 Outlier Channel Splitting

5.3.1 Improving Quantization with `Net2WiderNet`

The core idea of OCS is to reduce the magnitude of outlier weights and/or activations in a DNN layer by duplicating a neuron, then either (1) halving its output; (2) halving the outgoing weight connections. This leaves the layer functionally equivalent but makes the weight/activation distribution narrower and thus more suitable for linear quantization. Such layer transformations were originally proposed as `Net2WiderNet` in Net2Net [CGS16]; we leverage them to improve quantization.

More formally, consider a linear layer in a DNN which takes as input the m-channel activation vector $\mathbf{x} = \{\mathbf{x}_i\}_{i=0}^{m}$, where each $\mathbf{x}_i$ can be a single value (FC layer) or a 2D feature map (conv layer).

Let $\mathbf{y} = \{\mathbf{y}_j\}_{j=0}^{n}$ be the n-channel output. We can define a linear layer as follows:

$$\mathbf{y}_j = \sum_{i=1}^{m} \mathbf{x}_i * \mathbf{W}_{ij} \tag{5.3}$$

where $\mathbf{W}_{ij}$ represents the weight(s) connecting $\mathbf{x}_i$ and $\mathbf{y}_j$ and $*$ represents multiplication or 2D convolution over a single channel. Without loss of generality, consider using OCS to split the last channel $\mathbf{x}_m$. This equates to rewriting Equation 5.3 as follows:

$$\mathbf{y}_j = \sum_{i=1}^{m-1} \mathbf{x}_i * \mathbf{W}_{ij} + (\mathbf{x}_m * \frac{\mathbf{W}_{mj}}{2}) + (\mathbf{x}_m * \frac{\mathbf{W}_{mj}}{2}) \tag{5.4}$$

$$\mathbf{y}_j = \sum_{i=1}^{m-1} \mathbf{x}_i * \mathbf{W}_{ij} + (\frac{\mathbf{x}_m}{2} * \mathbf{W}_{mj}) + (\frac{\mathbf{x}_m}{2} * \mathbf{W}_{mj}) \tag{5.5}$$

In both cases, we split channel m into 2 channels. To preserve equivalence, we can halve the weights (Equation 5.4) or halve the input activations (Equation 5.5). Figure 5.2(a) taken from the Net2Net paper illustrates weight OCS visually: by duplicating y_2 we can cut its outgoing weight v_2 in half.

OCS is an alternative to clipping for reducing the dynamic range of DNN values without retraining. Compared to clipping, OCS preserves the outliers but incurs additional network overhead. The outlier values are the largest values in a layer and contribute the most to the outputs. We expect OCS to outperform clipping in neural network accuracy — the question is whether it can do so with low overhead.

Figure 5.2(b) shows some additional caveats of OCS in a layer with 2 inputs and 2 outputs. The top equation describes the original layer; the next two equations illustrate OCS to split the activations and the weights, respectively. One caveat is that to split any weight value, an entire row must be added to the weight matrix. For a conv layer, OCS requires duplicating an entire 2D activation channel and all 2D weight filters connected to that channel. A second caveat is that not all values need to be split. At the bottom of Figure 5.2(b), w_4 is split in half while w_3 is not split.

5.3.2 Quantization-Aware Splitting

In this section, we show that duplicating and dividing a value by two following `Net2WiderNet` increases the total quantization error. We then propose an alternative split ratio which preserves quantization error. Without loss of generality, consider the deterministic rounding function $Q(x) = \lfloor x + \frac{1}{2} \rfloor$, which maps each real number to its closest integer, rounding halves toward $-\infty$. The maximum error introduced by $Q(x)$ is 0.5. Next define OCS as a function $f(w) : \mathbb{R} \to \mathbb{R}^2$ which maps a single value to two values. The naïve split used in `Net2WiderNet` is:

$$\text{OCS}_{\text{naïve}}(w) = \begin{pmatrix} w/2 \\ \\ w/2 \end{pmatrix} \tag{5.6}$$

It is clear that $Q(w) \neq Q(\frac{w}{2}) + Q(\frac{w}{2})$, i.e. naïve OCS does not preserve the quantized value. The maximum total quantization error is doubled as both halves may be rounded in the same direction (e.g., $w = 3$ and each half is 1.5).

To address this, we propose the following *quantization-aware* (QA) splitting function:

$$\text{OCS}_{\text{QA}}(w) = \begin{pmatrix} (w-0.5)/2 \\ \\ (w+0.5)/2 \end{pmatrix} \tag{5.7}$$

Intuitively, this forces $Q(x)$ to round in different directions when w is close to the midpoint between grid points. More formally, we can prove the following:

$$\begin{aligned} & Q(\frac{w-0.5}{2}) + Q(\frac{w+0.5}{2}) \\ &= \left\lfloor \frac{w-0.5}{2} + \frac{1}{2} \right\rfloor + \left\lfloor \frac{w+0.5}{2} + \frac{1}{2} \right\rfloor \\ &= \left\lfloor \frac{w+0.5}{2} \right\rfloor + \left\lfloor \frac{w+0.5}{2} + \frac{1}{2} \right\rfloor \\ &= \lfloor w + 0.5 \rfloor \end{aligned} \tag{5.8}$$

The last line is simply $Q(w)$, showing that QA OCS preserves the original quantization result. To derive the last line, we apply Hermite's Identity [SA03] with $n = 2$:

$$\sum_{k=0}^{n-1} \left\lfloor x + \frac{k}{n} \right\rfloor = \lfloor nx \rfloor \tag{5.9}$$

We can further show that QA splitting is optimal; there exists no way to split w which results in lower quantization error. This proof is omitted due to length.

5.3.3 Channel Selection

As stated earlier, OCS cannot target individual weights or activations and must duplicate entire channels. OCS performs splits one at a time, and always splits the channel containing the largest absolute value in the layer. By prioritizing channels containing the largest values, OCS seeks to minimize distortion caused by any subsequent clipping. We use a simple method to determine how many splits to perform in each layer (i.e., how many extra channels are created). For a layer containing C channels, OCS splits ceil$(r * C)$ channels, where r is the *expansion ratio*, a hyperparameter that determines approximately the level of tolerable overhead in the network. This method allocates extra channels without considering each layer's weight or activation distributions. We also tried a more intelligent approach which formulates extra channel allocation as a knapsack problem. The reward function is the percentage reduction in the dynamic range of the distribution, and the cost is the increase in memory size. We optimize the number of extra channels for all layers simultaneously subject to a constraint on the memory overhead. Unfortunately, the knapsack approach is experimentally not better than the simple method described above, and for space reasons we do not show results with knapsack.

Channel selection on DNN weights is straightforward to implement as the weights are known and fixed post-training. For the activations, we take an approach similar to TensorRT [Mig17]: we use a small number of training images to sample the activations in each layer. The sampled distributions in each layer are then used for OCS.

5.3.4 Implementation on Commodity Hardware

A key strength of OCS is simplicity, allowing it to be used in practical scenarios with either commodity hardware or emerging deep learning accelerators. Figure 5.2(b) shows the network

Table 5.1: Quantization-aware (QA) splitting in OCS — each table entry shows (**QA / non-QA**) where non-QA means simply dividing by two. The model is ResNet-20 for CIFAR-10.

Wt. Bits	OCS Expand Ratio			
	0.01	**0.05**	**0.1**	**0.2**
6	**92.0** / 91.9	91.9 / **92.0**	**92.1** / 91.9	92.0 / 92.0
5	**91.6** / 91.4	**91.7** / 91.6	**92.0** / 91.8	91.7 / **91.8**
4	88.0 / **88.3**	88.2 / **88.3**	**88.7** / 86.8	**89.1** / 86.8
3	**49.9** / 44.5	**58.3** / 44.8	**62.7** / 44.6	**76.5** / 52.8

modifications needed to implement OCS, which involves duplicating and scaling certain channels in the weights and activations. The weight modifications can be done off-line prior to serving the model. For the activations, a custom layer can be inserted which simply copies and scales the appropriate channels.

5.4 Experimental Evaluation on CNNs

This section reports experiments on CNN models for ImageNet classification [DDS+09] conducted using PyTorch [PGC+17] and Intel's open-source Distiller [3] quantization library. Post-training quantization was performed using Distiller's symmetric linear quantizer, which scales the quantization grid based on the maximum absolute value following Equation 5.1. For activation quantization, we first sampled the activation distributions using 512 *training* images (i.e., images not part of the test set) to determine the quantization grid points, then use this grid during testing. This profiling took between 40 and 200 seconds on our machine using an NVIDIA GTX 1080 Ti. Weight clipping and OCS does not require profiling and was performed without any data.

The chosen CNN benchmarks are four popular ImageNet models: VGG16 [SZ15] with batch normalization added, ResNet-50 [HZRS15], DenseNet-121 [HLWvdM17], and Inception-V3 [SLJ+15]. Pre-trained weights were obtained from the PyTorch model zoo and we ran inference only. The first layer was not quantized as it generally requires more bits than the others, and contains only 3 input channels meaning OCS would incur a large overhead.

[3] https://github.com/NervanaSystems/distiller

Table 5.2: [ImageNet Top-1 accuracy with OCS weight quantization] — the float accuracy is displayed under the model name. Results include different **Clip** methods, **OCS** with different expand ratios, and **OCS + the Best Clip method** at each bitwidth. For clipping, the best result is bolded and copied to the **Clip - Best** column. For OCS, the smallest expand ratio that outperforms all clipping methods is bolded, and the smallest expand ratio that achieves +1% accuracy over clipping is highlighted in blue. Best viewed in color.

Network	Wt Bits	Clip				Clip Best	OCS			OCS + Best Clip		
		None	MSE	ACIQ	KL		0.01	0.02	0.05	0.01	0.02	0.05
VGG-16 BN (73.4)	8	**73.0**	72.6	72.8	68.4	73.0	72.6	72.9	72.8	72.7	72.8	72.5
	7	**72.8**	72.5	72.5	60.7	72.8	72.1	**72.8**	72.5	72.4	72.1	72.6
	6	70.8	**71.3**	71.2	63.2	71.3	72.3	72.2	72.3	**71.8**	71.8	72.1
	5	63.1	**66.9**	61.2	62.7	66.9	69.3	70.2	71.0	68.8	69.5	70.0
	4	0.2	53.5	34.2	**59.4**	59.4	10.4	26.3	37.9	63.8	63.8	65.9
ResNet-50 (76.1)	8	75.4	**75.5**	75.4	73.5	75.5	**75.7**	75.7	75.7	**75.7**	75.7	75.4
	7	75.0	**75.2**	75.0	72.8	75.2	**75.5**	75.5	75.6	**75.5**	75.5	75.5
	6	72.9	73.5	**74.3**	71.6	74.3	**74.9**	74.7	75.0	**74.8**	74.8	75.2
	5	14.5	69.1	**69.9**	69.4	69.9	69.4	**71.9**	72.6	71.0	71.9	73.4
	4	0.1	45.0	33.2	**62.9**	62.9	12.1	36.1	55.2	66.2	67.1	69.3
DenseNet-121 (74.4)	8	**74.1**	73.8	73.7	71.0	74.1	**74.2**	74.2	74.2	**74.2**	74.2	74.2
	7	**73.8**	73.3	73.1	62.3	73.8	**73.9**	74.0	74.0	**74.1**	74.2	74.1
	6	71.0	**71.4**	71.1	60.7	71.4	72.9	73.0	73.2	73.2	73.1	73.1
	5	46.9	**65.4**	61.4	54.6	65.4	**65.5**	69.7	71.3	70.0	70.7	71.6
	4	0.4	33.3	25.2	**42.6**	42.6	13.1	37.5	53.0	52.7	56.5	63.0
Inception-V3 75.9)	8	**74.8**	74.6	74.0	72.6	74.8	**75.2**	75.4	75.3	74.8	**75.0**	74.9
	7	**73.2**	71.2	69.1	69.4	73.2	74.8	74.7	74.7	71.8	**73.8**	74.2
	6	58.3	**66.2**	62.3	63.0	66.2	71.3	71.8	72.1	70.5	71.7	72.5
	5	0.5	30.4	29.6	**40.5**	40.5	**45.2**	54.0	60.2	57.0	60.0	62.9
	4	0.1	0.2	0.1	**1.6**	1.6	0.1	0.2	0.6	2.1	2.3	4.8

5.4.1 Effect of Quantization-Aware Splitting

The first experiment compares our proposed quantization-aware (QA) splitting against simply dividing by two as per Net2Net. Table 5.1 displays results from ResNet-20 for CIFAR-10 [KH09]. Although the difference is negligible until 4 bits (at which point there is significant accuracy degradation), QA splitting is clearly better than the naïve method. This validates our mathematical ideas about QA splitting, and we use the technique in all ensuing experiments.

5.4.2 Weight Quantization

Table 5.2 compares different clipping methods and OCS on weight quantization. The weights were quantized to 8-4 bits, while the activations were quantized to 8 bits. Floating-point accuracy is displayed under the model name. Linear quantization without clipping or OCS is shown in the **Clip - None** column. For ease of comparison we copy the best clipping result to the **Clip - Best** column. A range of small expand ratios r was chosen for OCS.

Our results indicate that for large bitwidths, there is no advantage to doing weight clipping. This is in line with NVIDIA's report on TensorRT [Mig17], which reported the same at 8 bits. Clipping becomes beneficial at 6 bits or fewer, improving accuracy by up to 55% for Resnet-50 and 40% for Inception-V3. Interestingly, the best-performing clipping technique depends on bitwidth and follows a consistent pattern across network architectures. As we go from high to low bitwidth, the winning technique goes from no clipping, to MSE/ACIQ, to KL at 4 bits.

Weight OCS with an expansion ratio of only $r = 0.01$ outperforms our benchmarked clipping methods at 8-5 bits. At 8 and 7 bits the difference between OCS and clipping is small and there isn't a clear trend of improvement for higher expand ratios; in this regime OCS is not especially effective and the accuracy differences between expand ratios are mostly noise. At 6 and 5 bits, OCS with $r = 0.02$ outperforms clipping by 1% for all models except ResNet-50, and up to 13% for Inception-V3. This demonstrates that the basic idea of OCS works. By splitting the outliers to preserve their values instead of clipping them, OCS can improve the accuracy of post-training

quantization. Another trend is that OCS gets most of its gains from small expansion ratios. The gain from $r = 0$ (no OCS) to $r = 0.01$ is always larger than the gain from moving to higher r values. Note that our benchmark networks have channel widths in the tens to hundreds so $r = 0.01$ equates to a single channel split in many layers. This again makes intuitive sense: the first channel split will target the unique largest outlier, guaranteeing a narrower weight distribution. Further splits target smaller values which occur with higher frequency, making OCS less effective at reducing the distribution width.

Given this intuition, we expect that a combination of OCS (to remove the largest outliers) followed by clipping (to further shrink the quantization grid) might surpass either method alone. The rightmost columns of Table 5.2 show results for OCS + Best Clip (i.e., applying OCS followed by the best performing clip method at each bitwidth). At 5 and 4 bits, OCS plus clipping cleanly outperforms OCS alone. OCS and clipping both seek to shrink the dynamic range of the quantized values, and thus there is some level of overlap between them. At high precision, OCS with our chosen expand ratios eliminates enough outliers such that additional clipping is unnecessary. At low precision, we believe that OCS would require huge expand ratios to fully address the outlier problem. In this regime OCS can combine with clipping to produce the best quantization results.

5.4.3 Activation Quantization

The same benchmarks and setup were used for activation quantization, except weights were kept at 8 bits while the bitwidth was varied for activations. To select the channels to split, we sampled activation distributions and counted the number of extreme values (we used values greater than the 99'th percentile) in each channel. Channels with the highest counts were split.

Table 5.3 shows activation quantization results. Unlike the weights, clipping is effective at all bitwidths tested. This is again in agreement with TensorRT [Mig17], which applied clipping to 8-bit activations. MSE clipping outperforms the other clip threshold techniques in nearly all cases. The gap between MSE and KL divergence is very small for large bitwidths, but at fewer bits MSE is clearly better. ACIQ performs worse than the other two methods with the exception of

Table 5.3: ImageNet Top-1 accuracy with OCS activation quantization — formatting is identical to Table 5.2 except weight bits is kept at 8 while the activation bitwidth is changed. We did not combine OCS with clipping due to ineffectiveness of OCS on activations.

Network	Act. Bits	Clip				Clip Best	OCS		
		None	MSE	ACIQ	KL		0.01	0.02	0.05
VGG16-BN (73.4)	8	72.5	**73.2**	73.1	**73.2**	73.2	72.7	72.8	72.5
	7	70.8	**72.8**	**72.8**	72.7	72.8	70.5	70.7	70.2
	6	49.0	71.3	**71.4**	70.6	71.4	49.2	46.0	45.9
	5	0.7	**62.0**	58.1	51.6	62.0	1.6	1.0	1.4
	4	0.1	**11.5**	5.0	2.4	11.5	0.1	0.2	0.1
ResNet-50 (76.1)	8	75.5	**75.9**	75.8	75.8	75.9	75.6	75.5	75.7
	7	**75.4**	75.3	75.2	75.3	75.4	74.1	74.5	74.1
	6	62.6	**73.5**	**73.5**	72.8	73.5	63.3	63.3	63.6
	5	5.7	63.7	**65.4**	56.7	65.4	10.0	12.6	6.0
	4	0.1	9.0	**20.6**	7.2	20.6	0.1	0.1	0.1
DenseNet-121 (74.4)	8	74.0	**74.1**	73.8	**74.1**	74.1	74.1	***74.2***	74.1
	7	73.0	**73.7**	72.9	73.7	73.6	73.2	73.2	73.0
	6	67.2	**72.6**	70.9	72.1	72.6	67.9	65.8	66.6
	5	19.9	**66.9**	64.6	64.5	66.9	16.0	18.7	13.2
	4	0.2	**26.9**	20.1	16.5	26.9	0.1	0.1	0.1
Inception-V3 (75.9)	8	74.8	**75.1**	73.4	75.0	75.1	74.8	74.9	74.8
	7	72.6	**74.2**	71.3	73.8	74.2	72.6	72.4	72.6
	6	51.6	**69.6**	60.7	67.9	69.6	54.1	51.5	48.5
	5	1.3	**34.2**	5.8	25.3	34.2	1.0	0.9	1.0
	4	0.1	**0.3**	0.1	0.2	0.3	0.1	0.1	0.2

ResNet-50, where it showed good performance.

Activation OCS provides some improvement over simple linear quantization, but performs worse than clipping. This is likely because OCS relies on being able to identify the exact channel containing the largest outlier. With activations, profiling can only indicate which channels are *likely* to contain outliers, the best channel to split varies from input to input. To test our explanation, we experiment with **Oracle OCS**, which is simply OCS with exact knowledge of the activations generated by the network during testin. Oracle OCS chooses different channels to split in each input batch. Table 5.4 displays the results for Oracle OCS with different batch size on two models with 6 bit activations. Even at batch size 32, the oracle can already match or surpass the best

Table 5.4: ImageNet Top-1 accuracy with Oracle OCS on activations — Oracle OCS splits a different set of channels for each input batch. Results use 6 activation bits and $r = 0.02$.

Batch Size	ResNet 50	Inception V3
1	74.6	71.7
2	74.5	71.7
4	74.0	71.6
8	74.1	70.9
32	73.5	70.7
128	73.3	70.3
No OCS	62.6	51.6
Clip Best	73.5	69.6

clipping result. Further reducing the batch size (allowing channel selection at a finer granularity) leads to even better accuracy. These results show that OCS and our channel selection strategy can be effective for activations. However, channel selection must be done *dynamically*, requiring additional run-time analysis which is difficult to implement and likely inefficient in commodity systems.

5.4.4 OCS Memory Overhead

Because OCS increases the input channels by a factor of r, rounded up, the expand ratio r is a lower bound for the model size overhead. Table 5.5 shows both weight and activation overhead for ResNet-50 with different values of r, show that the true overhead matches r very closely. To measure run time, we build a different implementation of OCS which splits the weights and duplicates the channels ahead of time. This is different from the (simpler but slower) implementation from the previous section which uses `torch.index_select` to duplication activation during execution. Table 5.6 shows run time overhead for ResNet-50 inference, and while the run time tracks above the expand ratio it is still within a reasonable range. While these numbers were collected on GPU, they should also hold true for dedicated accelerators.

Table 5.5: OCS parameter size overhead (ResNet-50) — memory overhead is very close to the expand ratio.

ResNet-50	Expand Ratio			
	0.01	**0.02**	**0.05**	**0.1**
Rel. Wt Size	1.01	1.02	1.05	1.1
Rel. Act Size	1.02	1.03	1.06	1.11

Table 5.6: OCS run time overhead (ResNet-50) – run time is measured for inference on a single RTX 2080TI.

	Base	Expand Ratio		
		0.01	**0.02**	**0.05**
Time (s)	39.8	42.9	43.4	44.2
Ratio	1.00	1.08	1.09	1.11

5.5 Experimental Evaluation on RNNs

This section reports experiments on an RNN model with two stacked LSTM layers for language modeling [ZSV14]. The corpus used is the WikiText-2 dataset [MXBS16] with a vocabulary of 33,278 words. Each LSTM layer has a hidden size of 650, and the dimension of the word embedding in the input layer is 650. As the CNN results have shown that activation OCS is not effective, we focused on experimenting with OCS and clipping on the weights. Activations and the hidden state were kept in floating-point for this experiment.

Table 5.7 compares the effects of OCS combined with different clipping methods on weight quantization. Lower perplexity is better, and the baseline floating-point model achieves a perplexity of 95.1. The best result on each row (i.e., the best clipping method at each OCS expand ratio) is bolded. Clipping is not effective on this model. None of the clipping techniques achieve any perplexity improvement. OCS achieves a much better result. At 6 bits, OCS begins to outperform the baseline with $r = 0.05$. At 5 bits, OCS sees steady perplexity decrease with successively larger expand ratios, clearly outperforming the best clipping result past $r = 0.02$. This is strong evidence that OCS can effectively improve data-free quantization beyond clipping.

5.6 Conclusions and Future Work

We propose outlier channel splitting, a method to improve DNN quantization without retraining which can be applied on commodity hardware. OCS splits channels in a layer to reduce the magnitude of outliers. Unlike the existing clip-based methods, OCS introduces a new tradeoff by

Table 5.7: WikiText-2 perplexity with OCS weight quantization — lower is better. The floating-point baseline achieves a perplexity of 95.1. The best performing clip method along each row is bolded.

Wt. Bits	Expand Ratio	Clip Method			
		None	MSE	ACIQ	KL
6	0.00	**94.5**	98.1	99.0	97.7
	0.01	**95.0**	97.9	99.0	97.7
	0.02	**94.6**	97.8	96.1	96.3
	0.05	**93.9**	96.6	96.1	95.9
5	0.00	**98.2**	99.4	100.8	98.8
	0.01	**97.3**	99.7	99.9	97.7
	0.02	**95.7**	98.9	99.2	97.0
	0.05	**95.1**	98.2	98.5	96.3

reducing quantization error at the cost of network size overhead. Experimental results demonstrate that OCS on weights outperforms state-of-the-art clipping techniques with minimal overhead on deep CNN and RNN benchmarks. At very low precision, OCS in conjunction with clipping outperforms either method alone. Because clipping is used in NVIDIA TensorRT, a commercial DF quantization flow, we believe that OCS has potential applicability in real-life systems.

Future work includes a more in-depth study into different channel selection methods, as well as applying OCS quantization during training. Specifically, we believe that OCS can help shape weight distributions during training to obtain better results than training for quantization alone.

CHAPTER 6

OVERWRITE QUANTIZATION FOR DNN ACTIVATION QUANTIZATION

6.1 Preliminaries

Chapter 5 introduces the concept of data-free (DF) quantization and the challenge posed by outliers. It then proposes outlier channel splitting (OCS) to improve DF quantization for DNN weights. However, OCS is a static DNN graph transformation (akin to a compiler pass) that needs to identify the channel which contains the largest outlier(s) in a layer. This is fine for the weights which are statically known, but not for activations which are only determined at run time. Unfortunately, activation outliers may be even more detrimental to neural network accuracy than weight outliers. A presentation from NVIDIA's TensorRT team showed that for popular ImageNet CNNs, the weights can be quantized to 8-bit integer directly while activations must first be clipped to limit their dynamic range [Mig17]. As data in Table 5.3 of Chapter 5 shows, clipping can improve activation quantization; however accuracy degradation is still severe at lower precisions such as 5 or 4 bits.

6.1.1 Hardware Specialization for Activation Outliers

One promising line of work is using specially co-designed hardware to handle outliers at run time. Park et al. [PYV18, PKY18] proposed an outlier-aware DNN accelerator (OLAccel) which uses a conventional dense processing engine (PE) for central values, and a second sparse PE for outliers. The conventional PE uses one bitwidth (4 or 8 bits in their experiments) while the outlier PE uses double that bitwidth to gain more dynamic range necessary to represent the outliers. The execution flow of OLAccel is shown in Figure 6.1. An input vector is separated into a dense vector containing most values, and a sparse format for outliers. Similar to other sparse vector formats, each outlier is accompanied by indices (width, height, channels) to track its location in the original vector. The dense (outlier-free) vector and the outliers are then processed in two separate hardware

units. Note that the figure does not show some additional complexities, such as how to fetch the corresponding value to multiply with the outlier, and how to combine the outputs of the two units into a single dense vector.

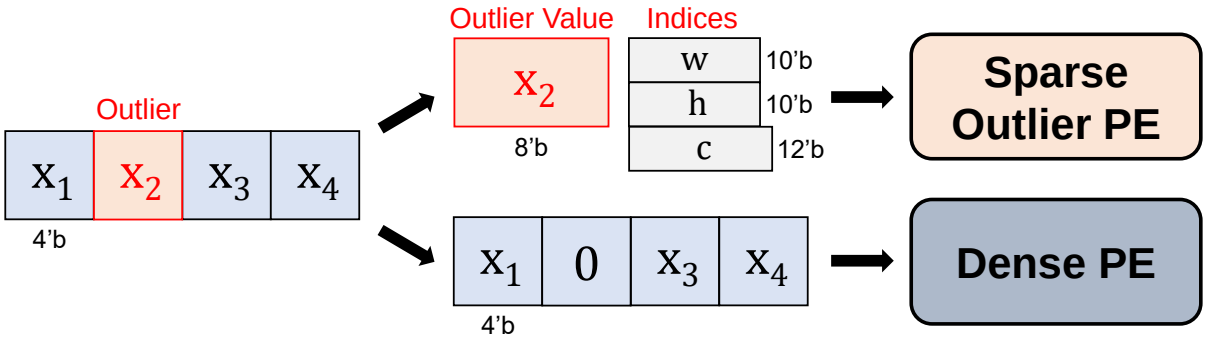


Figure 6.1: Execution flow of outlier-aware accelerator (OLAccel) [PKY18] — the input vector is separated into a dense vector containing most values, and sparse outliers. The vector is processed by a dense PE while the outliers are processed by a different sparse PE. Here w, h, and c are width, height, and channel indices needed to track the outlier.

Outlier-aware computation in OLAccel can greatly improve DNN accuracy. For AlexNet with 4-bit weights and 16-bit outliers, accuracy within 1% of the floating-point baseline can be achieved even when the outlier PE processes only 0.01 of all values. OLAccel exploits the fact that outliers are rare to perform the vast majority of computation in very low-precision, with a small fraction of computation done using higher bitwidth. It is also important to note that OLAccel does not require any form of training; it is a DF quantization technique.

The weakness of OLAccel is that the hardware resource overhead of the outlier processing engine is non-trivial. Park et al. performed an iso-area comparison with Eyeriss [CES16] and ZeNA [KAY17] but did not specify the exact area taken up by the outlier PE. However, we can make the following observations: (1) the outlier PE requires additional multiply-accumulate (MAC) units since it operates at a different bitwidth; (2) the sparse representation of outliers incurs extra storage and adds complexity to the PE. The storage overhead can be quantified by looking at Figure 6.1: each outlier contains 8 or 16 bits for data, and 32 additional bits for indices.

In this chapter of the thesis, we present overwrite quantization (OverQ), a hardware-efficient

technique which improves activation quantization by addressing outliers. OverQ's primary goal is to reduce the area overhead of OLAccel while still achieving significant accuracy gains over a naïve baseline. To achieve this, OverQ is non-deterministic. Instead of instead of handling all outliers like OLAccel, it only handles a fraction of outliers depending on the exact activation values. However, an exploratory study on ImageNet ResNet-18 shows that OverQ handles 80-95% of outliers and achieves 1.0-2.5% Top-1 accuracy improvement in realistic scenarios. An FPGA prototype built using HLS shows that OverQ requires no additional MACs, and its logic overhead is negligible compared to the MAC array size. Although the scope of the evaluation is currently limited, it serves as a proof of concept to demonstrate the potential of overwrite quantization in larger systems.

6.2 Overwrite Quantization

Consider the problem of quantizing an N-element activation vector $\{x\}_{i=1}^{N}$. Let outliers in the vector be defined as values larger than some fixed threshold. The basic idea of overwrite quantization is illustrated in Figure 6.2. An outlier x_i can *overwrite* its adjacent value x_{i+1} when x_{i+1} is smaller than a fixed threshold [1]. When this overwrite occurs, x_i is represented using twice the normal bitwidth while x_{i+1} is dropped. OverQ essentially dynamically re-allocates bitwidth from an insignificant value to a critically important outlier. OverQ exploits two properties of DNN activations: (1) outliers are rare, but contribute disproportionately to a layer's output; (2) activation distributions peak near zero, and contain many ReLU-induced zeros. The first property guarantees that overwrite occurs infrequently, so not too many values are dropped out. The second property means that an outlier will lie beside a small value with significant probability, enabling overwrite. From a hardware standpoint, OverQ is resource efficient. Instead of using a separate, larger MAC unit for an outlier as in OLAccel, OverQ reuses the MAC unit from an adjacent value and performs two low-precision MACs to compute the outlier result. Furthermore, OverQ uses a single flag bit

[1] The boundary value x_N has no adjacent value and cannot benefit from OverQ.

for each value to indicate the presence of overwrite, which should be much smaller than building a dedicated sparse PE.

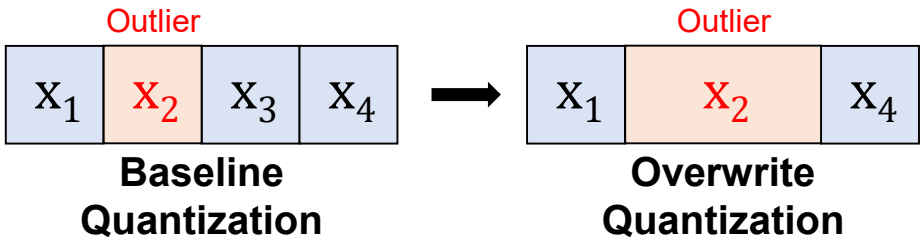


Figure 6.2: Basic idea of Overwrite Quantization — an outlier x_i is allowed to overwrite its adjacent value x_{i+1} when x_{i+1}. When this happens, twice the normal bitwidth can be used to represent x_i and x_{i+1} is dropped.

In DNNs, OverQ would be applied along a single dimension of the activation tensor (width, height, or channels for convolutional networks). The chosen dimension affects the hardware architecture as it determines which dimensions must be spatially unrolled. Some brief experiments showed that performing OverQ along the channels is much more effective than along either spatial dimension. The reason is that spatially adjacent activations are highly correlated in CNNs, while different channels exhibit less correlation. Discussion from this point onward assumes OverQ along the channels of a CNN.

Figure 6.3 illustrates how overwrite quantization works in greater detail and describes its two variants. Figure 6.3(a) shows a dot product computation between activations x_i and weights w_i. Each activation is associated with a flag bit indicating whether that value is being overwritten. Figure 6.3(b) shows OverQ-Half, where half the outlier value ($x_i/2$) is stored in each of the two hardware vector slots. During computation, the hardware module detects the flag bit at index $i+1$ and uses w_i instead of w_{i+1}. This enables the two products at indices i and $i+1$ thus sum to $x_i \times w_i$. As long as the weight vector is unrolled properly, the weights w_i and w_{i+1} will be spatially adjacent and accessing the adjacent weight is relatively cheap in hardware (see Section 6.2.2). We can also analyze OverQ-Half's effect on dynamic range. Let $x = \sum_0^{N-1} x^i 2^i$ be an N-bit binary value with x^i

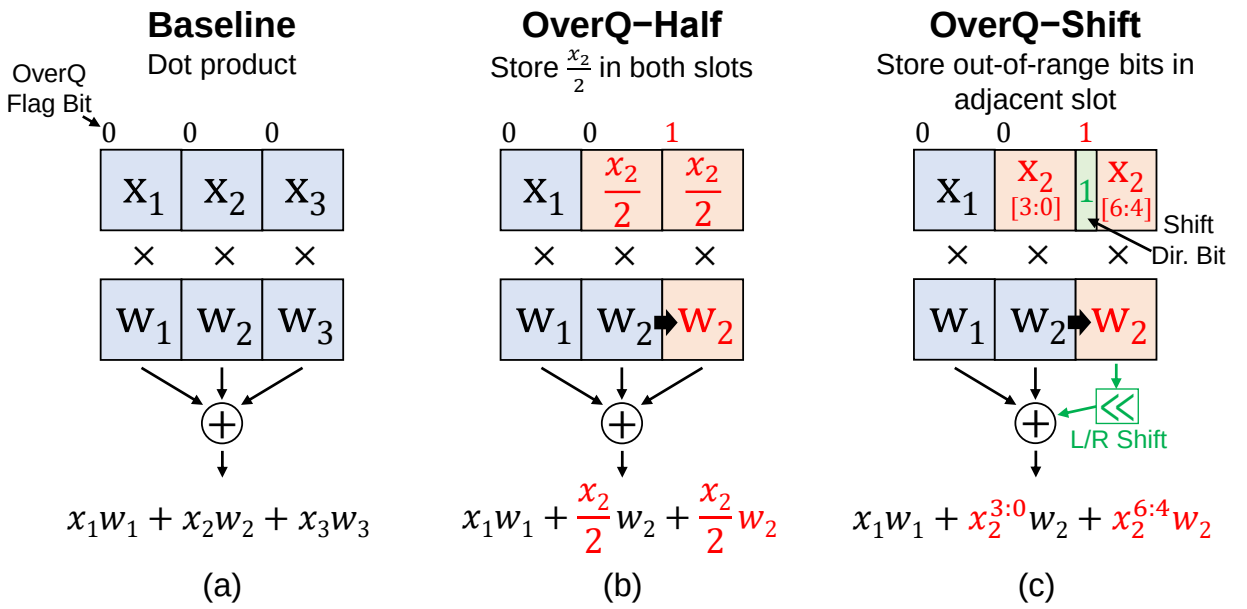


Figure 6.3: Dot product computation with OverQ — the two variants, OverQ-Half and OverQ-Shift, differ in how they use the additional bits to represent outliers. For both variants the weight w_i is copied to the adjacent MAC. Superscripts in the rightmost figure indicate bit slices.

being the i'th bit, and let $x^{a:b}$ indicate the sequence of bits $x^a,\ldots,x^b$. We have the following:

$$2\times(\frac{x}{2})^{(N-1):0} = 2\times(x >> 1)^{(N-1):0} = 2\times(x^{N:1} >> 1) = x^{N:1} \tag{6.1}$$

which shows that OverQ-Half increases the dynamic range of representation by one bit, from $x^{(N-1):0}$ to $x^{N:1}$. This means it doubles the original dynamic range.

Figure 6.3(b) shows OverQ-Shift, a variant of OverQ which further increases the dynamic range at the cost of greater hardware complexity. OverQ-Shift uses the adjacent slot's lower bits to hold out-of-range bits of the outlier x_i (either additional MSBs or additional LSBs). The first bit of the adjacent slot is used to hold a shift direction (right for MSB or left for LSB). In the figure, slot i stores bits $x^{3:0}$ and slot $i+1$ stores MSBs $x^{6:4}$. The shift direction bit indicates whether the product at index $i+1$ needs to be shifted left or right before being added to the accumulator. MSBs must be shifted right and LSBs must be shifted left. If the baseline quantization uses N bits, then OverQ-Shift increases the effective bitwidth by $N-1$.

Figure 6.4 gives a numerical example for overwrite quantization and how the two variants affect

precision (the total number of bits of representation) and dynamic range (the most significant bit that can be represented). With the same bits of storage, OverQ-Shift has a much greater effect on quantization than OverQ-Half. Critically, OverQ-Half does not increase the precision: this means that it only provides a benefit when the value is an outlier (i.e., the value exceeds the max quantization threshold and would normally be clipped). However, OverQ-Half has the advantage of simplicity: it can be implemented in hardware with only basic muxing logic. OverQ-Shift, on the other hand, can use its MSB mode for outliers, or use its LSB mode to improve the quantization precision for non-outlier values. The drawback of OverQ-Shift is that it is more hardware intensive and requires two constant shifters as well as more complex muxing logic.

Given the above, it is useful to separate the two ways in which OverQ can benefit a DNN accelerator. First is *outlier handling (OL)*, where an outlier overwrites its much-smaller neighbor. Second is *zero-reuse (ZR)*, where any non-zero value can overwrite an adjacent ReLU-induced zero. Put differently, OL refers to the ability to represent outliers beyond the quantization threshold, while ZR refers to the increase in precision for some non-outlier values. Critically, OverQ-Half can only support OL, while OverQ-Shift can support both OL and ZR. Our experiments specifically investigate the importance of both effects.

Figure 6.4: Numerical example of overwrite quantization — each slot stores three bits. OverQ-Half increases dynamic range by one bit, but does not affect precision. OverQ-Shift right simultaneously increases dynamic range and precision by two bits, while OverQ-Left increases only increases precision by two bits. The green bit in the adjacent slot is the shift direction bit.

Outlier value = 10110.11, baseline uses 3 bits from the decimal point

	Represented Value (Underlined)	Bits in Own Slot	Bits in Adj. Slot
Baseline	10110.11	110	-
OverQ-Half	10110.11	011	011
OverQ-Shift (shift right)	10110.11	110	110
OverQ-Shift (shift left)	10110.11	110	011

6.2.1 Channel Reordering

Although OverQ is intended to be a dynamic hardware-centric technique, its effectiveness can potentially be improved by ahead-of-time static transformations. As described above, OverQ is predicated on the fact that outliers will be adjacent to a small value with high probability. We can increase this probability by reordering the channels in a layer such that channels likely to contain outliers are adjacent to channels likely to contain zeros. Our proposed channel reordering procedure is as follows. First, activation distributions are sampled using a small profiling dataset (this is already fairly standard practice for data-free activation quantization [Mig17]). Then, the number of outliers in each channel are counted (outliers can be defined as the largest 1% of activations). Finally, the channels are reordered by outlier count following a high, low, high, low, high, etc sequence. Figure 6.5 shows the outlier count in each channel of a CNN layer before and after reordering. Reordering can be implemented as an ahead-of-time graph transformation which swaps weight filters in a layer such that the output channels follow the desired order. Importantly, this means there is no overhead at run time.

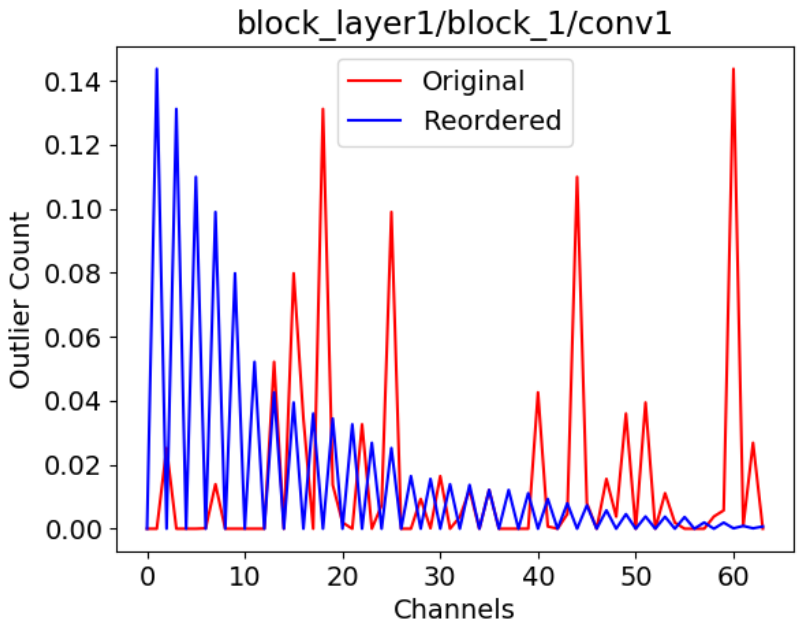


Figure 6.5: Channel reordering for OverQ — reordering places channels with high outlier count next to channels with low outlier count. Network use is ImageNet ResNet-18.

6.2.2 Mapping Overwrite Quantization to a Spatial Architecture

OverQ is intended from the ground up to be a lightweight, hardware-centric quantization technique. Specifically, OverQ can be implemented efficiently and with minimal hardware changes in a *weight-stationary* spatial array (a common architecture for DNN accelerators). Here, weight-stationary (WS) is a type of DNN dataflow [CES16] in which weights are held stationary in PEs while inputs and partial sums move through the accelerator. A recent study on dataflow choice in DNN hardware literature [YGP+18] showed that WS dataflow was the most popular. Experiments in the study also demonstrated that WS to be the most hardware efficient (albeit only by a small margin). Figure 6.6(a) shows the organization of a 2D WS spatial accelerator for matrix-vector computation; the left side shows the movement of input activations and output partial sums (psums) while the right side depicts the architecture of a PE.

A weight-stationary spatial architecture is well-suited for OverQ for two reasons. First, the array spatially unrolls the input channels; in Figure 6.6(a) the input channels are mapped to rows along the vertical axis. Adjacent channels are therefore mapped to spatially adjacent PEs during processing. Second, the weights are held stationary in each PE. These two factors make it relatively easy for a PE to access its adjacent weight. Figure 6.6(b) illustrates the implementation of OverQ-Half. A 1-bit wire is added to each PE to propagate the OverQ flag bit, which is used to multiplex between the PE's own weight and its adjacent weight.

Figure 6.6(c) shows the modifications needed for OverQ-Shift, which is more complex. In addition to the hardware for OverQ-Half, a second mux is needed after the multiplier to implement a possible shift on its output. For simplicity the figure only depicts selecting between no shift and a right shift, which is sufficient for outlier overwrite. To support zero-reuse, a larger mux is needed to choose between no shift, right shift, or left shift. A second select bit would be routed from input activation register.

To decide which activations to overwrite, we take advantage of the fact that output accumulation typically occurs at a larger bitwidth than the input weights or activations (e.g., this is done in Google's TPU [JYP+17]). When outputs exit the WS array at the bottom, they must be rescaled

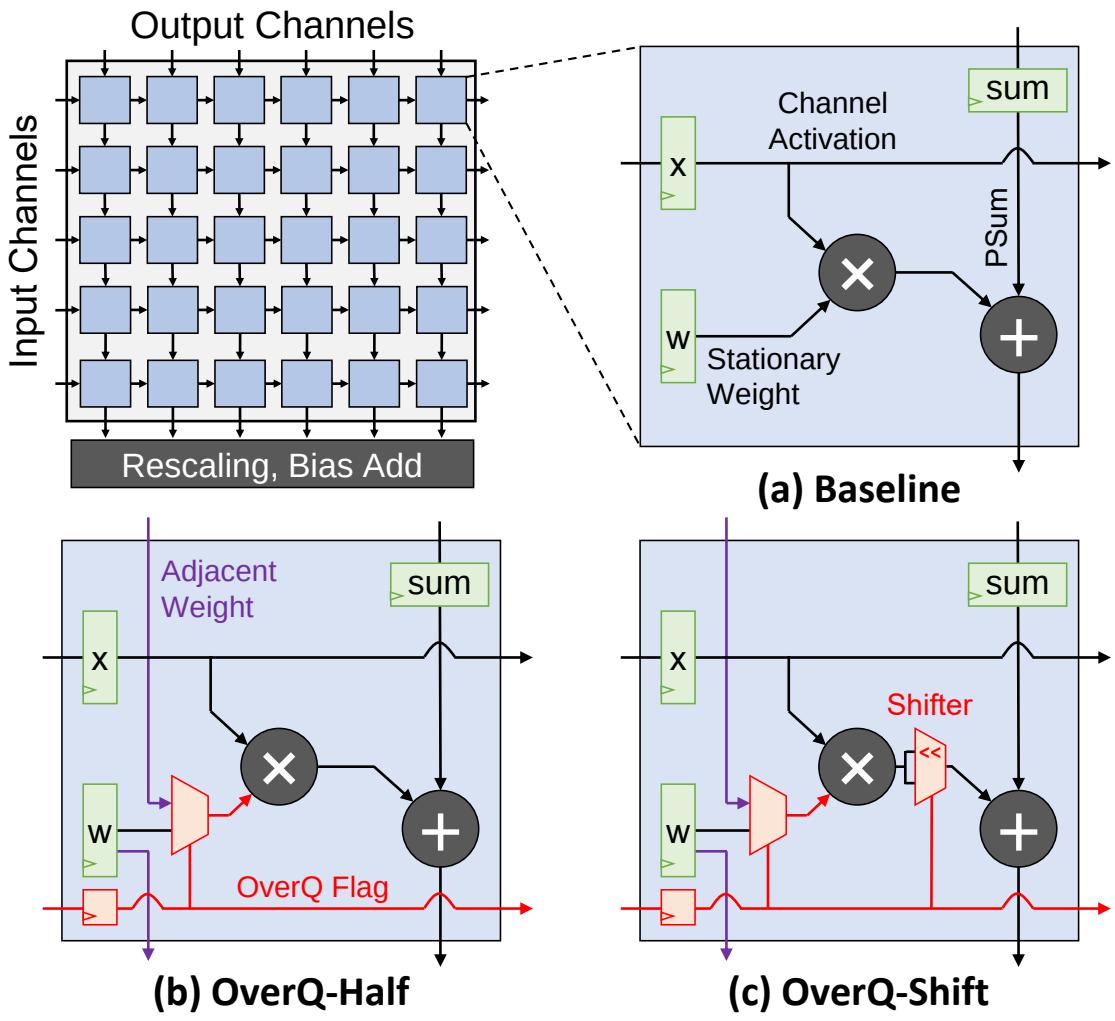


Figure 6.6: OverQ hardware architecture sketch — OverQ can be supported with lightweight changes in a weight-stationary 2D spatial array, a common DNN accelerator design. (a) Baseline systolic array and PE showing weight, activation, and partial sum (PSum) along with MAC unit; (b) OverQ-Half PE requires only a flag bit and a mux between own and adjacent weights; (c) OverQ-Shift PE requires additional muxing and shifters.

and re-quantized down to activation bitwdith for the next layer. The rescaling unit can be modified to make decisions on where to perform OverQ. The size of the rescaling unit scales with the width of the array only, whereas the PEs scale with both width and height of the array. As a result we believe that the dominant resource overhead of OverQ will be from the modifications to each PE.

6.3 Experimental Proof of Concept

To demonstrate the potential of overwrite quantization, two sets of experimental results are provided below. The first is an evaluation of the impact of OverQ on the accuracy of ResNet-18 [HZRS15] for ImageNet classification [DDS+09]. ResNet-18 was the most modern network used in the experimental section of OLAccel (the others being AlexNet and VGG-16). The second is measurements of resource usage on a small-scale prototype FPGA accelerator for CNN inference. Many accuracy experiments define the threshold for what constitutes an outlier using MMSE clipping [SSH15, SHS16], which was shown to be the best performing method from literature in Chapter 5. For the accuracy experiment, baseline and OverQ quantization was implemented in TensorFlow [ea15] and applied to ResNet-18 for ImageNet classification. The basic flow of quantization is to scale and clip values into a range $[-2^B - 1, 2^B - 1]$, rounding to integer, and scaling back to the original range. B here is the unsigned bitwidth in a sign-magnitude representation. OverQ was implemented on top of this leveraging `tf.select`, TensorFlow's ternary if-else operator. For the FPGA area experiments, we take the weights and activations generated by Tensorflow and use it to validate the correctness of the hardware.

6.3.1 Effectiveness of Channel Reordering

Reordering was implemented as a TensorFlow graph edit pass which visits each layer and performs static weight shuffling to generate the desired output channel order. This pass can be applied to a TensorFlow model checkpoint to generate an alternative checkpoint with reordered channels. To evaluate channel reordering, let us first define *outlier coverage* as the fraction of outliers which can benefit from overwrite. Ideally, coverage would be 100% such as in OLAccel, but the stochastic nature of OverQ prevents this. Nevertheless it is preferred that coverage be as high as possible. Print statements (`tf.print`) were used to log outlier coverage in ResNet-18 during inference for a single batch of 250 images. The outlier threshold for this experiment was determined using MMSE clipping. Table 6.1 compares the coverage original and reordered models

Table 6.1: Outlier fraction and coverage on ResNet-18 — data for the first six layers are shown. Channel reordering non-trivially improves coverage in most layers.

Layer Name	Outlier Fraction	Coverage		
		Baseline	Reorder	Delta
block_layer1/block_1/conv1	0.22%	76.5%	85.3%	8.8%
block_layer1/block_1/conv2	0.32%	99.5%	95.6%	-3.8%
block_layer1/block_2/conv1	0.15%	82.4%	85.6%	3.2%
block_layer1/block_2/conv2	0.09%	83.0%	85.4%	2.4%
block_layer2/block_1/conv1	0.15%	73.0%	80.1%	7.1%
block_layer2/block_1/conv2	0.09%	88.7%	90.3%	1.6%

in the first eight layers. The data shows that reordering non-trivially improves outlier coverage, up to 8.8% in some layers. Reordering is most effective in early layers where the number of channels is relatively small. The remaining experiments in this section use the reordered ResNet-18 model.

6.3.2 Accuracy Impact on ImageNet Classification

As Chapter 5 showed, the accuracy of a quantized CNN depends greatly on the clipping threshold (i.e., the scaling value mentioned above). A data-free quantization flow following NVIDIA TensorRT [Mig17] and OCS [ZHD+19a] was used. A profiling dataset of 500 training images was used to sample the activation distribution, and from this the clip threshold S for each layer was determined. S is the maximum value that can be represented by the fixed-point format. The threshold below which values may be overwritten in OverQ is set at $S/4$; comparing a fixed-point value against $S/4$ can be done efficiently using bitwise logic. Two experiments were conducted: (1) sweeping the clip threshold from $0.2 - 0.9\times$ of the maximum sampled value; (2) using the MMSE clip threshold in each layer. Weights were quantized to eight bits in all experiments. The floating-point baseline accuracy for ResNet-18 is 69.7%. We also investigate the relative importance of outlier handling (OL) and zero-reuse (ZR) by experimenting with each effect in isolation, then together.

Figure 6.7 shows the sweep of the clip threshold S with five and four bit activation quantization. The plots show Baseline, OL, ZR, and simultaneous OL+ZR. In both plots we see the same pattern:

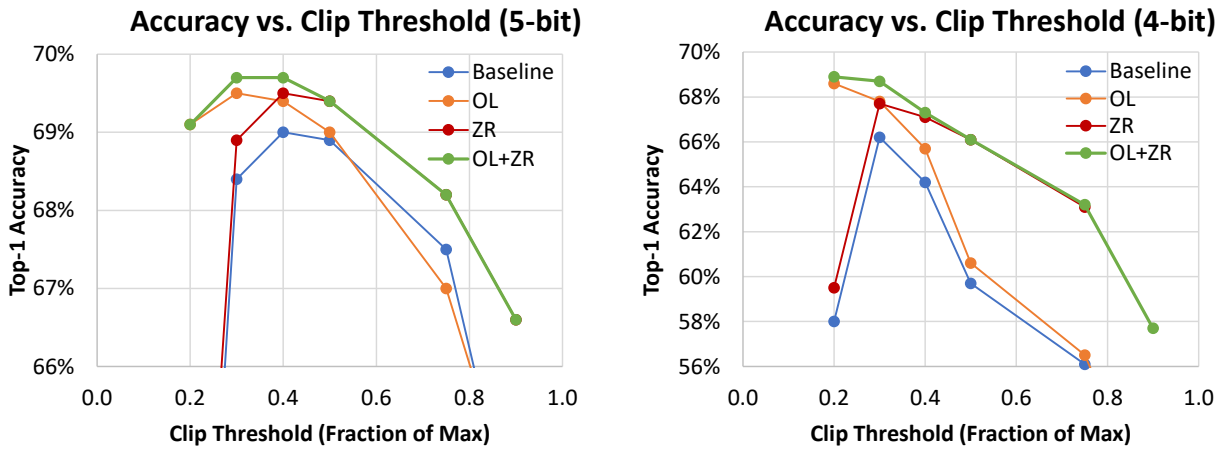


Figure 6.7: OverQ Top-1 accuracy vs. clip threshold on ResNet-18 — the plot shows Baseline, outlier handling (OL), zero-reuse (ZR), and simultaneous OL+ZR. The floating-point network achieves 67.9%.

at small clip thresholds OL performs above baseline and ZR near baseline, while at large clip thresholds OL performs near baseline and ZR above. Conceptually, a small clip threshold produces more outliers (cases where the activation exceeds S) and so OL occurs more often, achieve greater accuracy gains. Meanwhile, ZR alone performs poorly since it cannot address clipped outliers which bring down the accuracy. In the large S regime there are very few outliers for OL, meaning ZR is more effective. Fortunately, it is not necessary to choose. Simultaneous OL+ZR strictly outperforms either technique alone and the curve behaves as the sum of the accuracy gains from OL and ZR. At the accuracy peak, OL+ZR improves Top-1 by 0.7% at 5 bits and 2.5% at four bits.

Table 6.2 shows the accuracy under MMSE clipping. OverQ obtains 0.3% accuracy improvement at five bits and 1.7% at four bits. With 5-bit activations, MMSE clipping with OverQ achieves accuracy equal to the floating-point baseline. With 4-bit activations, OverQ achieves a 0.9% accuracy loss from baseline. With 3-bit activations, OverQ achieves a significant 8.4% accuracy improvement compared to just clipping. OLAccel [PKY18] reports that for deep models (ResNet-101 and DenseNet-121), OLAccel with 4-bit quantization and 3% outlier threshold results in $< 1\%$ accuracy loss. While OLAccel's benchmarks are larger and deeper, we believe the accuracy results of OverQ are comparable.

Table 6.2: OverQ ResNet-18 Top-1 accuracy — the table comparse no clip, MMSE clip, and OverQ with MMSE clip thresholds. The floating-point network achieves 67.9%.

Weight Bits	Act. Bits	No Clip	MMSE Clip	MMSE Clip + OverQ
8	5	69.1	69.4	**69.7**
8	4	64.3	67.1	**68.8**
8	3	36.8	56.3	**64.7**

6.3.3 Hardware Resource Impact on FPGA Prototype

For our hardware evaluation, we implement OverQ on a small matrix-matrix multiply accelerator, and validate the design using data extracted from the previous experiments in TensorFlow. The accelerator was built using C++ source and synthesized to Verilog using Xilinx Vivado HLS version 2016.3, then implemented to bitstream for a Xilinx `xc7z020` FPGA device. The baseline design is an $M \times N$ array pipelined in HLS, where M and N are the unroll factors for the input and output channels. Evaluation was done on inputs, weights, and outputs taken from a single layer in ResNet-18; OverQ has no impact on latency. Table 6.3 shows resource usage of the baseline, OverQ-Half (OL only) and OverQ-Shift (OL+ZR) designs. The key observation is that although the percentage increase in LUTs and FFs are considerable (LUT usage grow by over $3\times$ from baseline to OverQ-Shift), the LUT and FF utilization on the device remains very low in comparison to DSPs (which are used to implement MAC units). Based on our data, scaling up the design will result in DSPs running out well before the logic overhead of OverQ becomes an issue. Commercial FPGAs have an abundance of LUTs and FFs available, but DSPs are at a premium because the latter is much larger in area. DNN accelerators are almost always bottlenecked by DSP and/or BRAM usage [ZLS+15, QWY+16, ZL17]. Fortunately, OverQ was specifically designed to avoid MAC and DSP overhead, and this is reflected in the results. Another important finding is that the addition of OverQ does not hurt timing.

It is important to note that the accelerator is only for matrix-matrix products, and lacks other components typically found in a DNN accelerator such as memory system, rescaling unit, bias unit, etc. The hardware which decides which activations to overwrite was not prototyped. Section 6.2.2

Table 6.3: OverQ resource usage in an FPGA prototype — OverQ incurs non-trivial LUT and FF overhead, but total utilization of these resources remains very low. OverQ has no impact on DSP, BRAM, and timing.

Design	4x4			8x8			Device
	Base	OverQ Half	OverQ Shift	Base	OverQ Half	OverQ Shift	Total
CP	4.79	4.56	4.81	5.01	4.91	4.98	
Slice	215	208	230	677	828	965	
LUT	116	220	414	671	923	1660	53,200
FF	752	752	860	2290	2619	2781	106,400
DSP	16	16	16	64	64	64	220
BRAM	0	0	0	0	0	0	140

describes how the rescaling unit can be modified to make the OverQ decisions and how its resource overhead is expected to be small compared to the overhead in the PE array.

6.4 Conclusions and Future Work

Our preliminary data demonstrates that OverQ has potential as a lightweight hardware-centric technique for improving activation quantization. However, we have only performed tests on a single, relatively simple network (ResNet-18) and our hardware prototype consisted of just the core MAC array, leaving out other important components required for a full-fledged DNN accelerator (e.g., memory system, batch normalization, and bias add). Nevertheless, the data gives us confidence that OverQ is indeed feasible in real hardware at fairly low hardware overhead. The most pertinent future work is expanding the scope of the evaluation to an accelerator which can execute an entire DNN on its own, and to collect data on larger networks.

BIBLIOGRAPHY

[A+18] T. Ajayi et al. Experiences Using the RISC-V Ecosystem to Design an Accelerator-Centric SoC in TSMC 16nm. *CarryV*, 2018.

[ACRB16] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. YodaNN: An Ultra-Low Power Convolutional Neural Network Accelerator based on Binary Weights. *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 236–241, Jul 2016.

[AH18] D. Amodei and D. Hernandez. AI and Compute, May 2018.
URL `https://openai.com/blog/ai-and-compute/`

[ANA10] A. Agarwal, M. C. Ng, and Arvind. A Comparative Evaluation of High-Level Hardware Synthesis Using Reed–Solomon Decoder. *IEEE Embedded Systems Letters*, 2(3):72–76, 2010.

[BAMR10] Y. Ben-Asher, D. Meisler, and N. Rotem. Reducing Memory Constraints in Modulo Scheduling Synthesis for FPGAs. *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*, 3(3):15, 2010.

[BCB14] D. Bahdanau, K. Cho, and Y. Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv preprint*, arXiv:1409.0473, Sep 2014.

[BDTD+16] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to End Learning for Self-Driving Cars. *arXiv preprint*, arXiv:1604.07316, Apr 2016.

[BNHS18] R. Banner, Y. Nahshan, E. Hoffer, and D. Soudry. ACIQ: Analytical Clipping for Integer Quantization of Neural Networks. *arXiv preprint*, arXiv:1810.05723, Oct 2018.

[CAD+12] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From OpenCL to High-Performance Hardware on FPGAs. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, pages 531–534, Aug 2012.

[CAS16] P. Covington, J. Adams, and E. Sargin. Deep Neural Networks for YouTube Recommendations. *Conference on Recommender Systems (RecSys)*, pages 191–198, Sep 2016.

[CBD15] M. Courbariaux, Y. Bengio, and J.-P. David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. *Advances in Neural Information Processing Systems (NeurIPS)*, pages 3123–3131, 2015.

[CC04] D. Chen and J. Cong. DAOmap: A Depth-Optimal Area Optimization Mapping Algorithm for FPGA Designs. *Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 752–759, 2004.

[CCA+11] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2011.

[CCA+13] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems. *ACM Trans. on Embedded Computing Systems (TECS)*, 13(2):24, 2013.

[CCP06] D. Chen, J. Cong, and P. Pan. FPGA Design Automation: A Survey. *Foundations and Trends in Electronic Design Automation*, 1(3):139–169, 2006.

[CD94] J. Cong and Y. Ding. FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 13(1):1–12, 1994.

[CDS+14] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-earning. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar 2014.

[CES16] Y.-H. Chen, J. Emer, and V. Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2016.

[CFO+18] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. E. Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. G. Rapsang, S. K. Reinhardt, B. D. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Y. Xiao, D. Zhang, R. Zhao, , and D. Burger. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro*, 38(2):8–20, 2018.

[CGM+15] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini. Origami: A Convolutional Network Accelerator. *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, May 2015.

[CGS16] T. Chen, I. Goodfellow, and J. Shlens. Net2net: Accelerating Learning via Knowledge Transfer. *Int'l Conf. on Learning Representations (ICLR)*, May 2016.

[CH95] J. Cong and Y.-Y. Hwang. Simultaneous Depth and Area Minimization in LUT-based FPGA Mapping. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 68–74, 1995.

[CH05] A. Chowdhary and J. P. Hayes. Area-Optimal Technology Mapping for Field-Programmable Gate Arrays Based on Lookup Tables. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 24(7):999–1013, 2005.

[Cho16] F. Chollet. Xception: Deep learning with Depthwise Separable Convolutions. *Conf. on Computer Vision and Pattern Recognition (CVPR)*, Jun 2016.

[CHS+16] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv preprint*, arXiv:1602.02830, Feb 2016.

[CHW+13] A. Coates, B. Huval, T. Wang, D. J. Wu, A. Y. Ng, and B. Catanzaro. Deep Learning with COTS HPC Systems. *Int'l Conf. on Machine Learning (ICML)*, pages 1337–1345, Jun 2013.

[CKES17] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *Journal of Solid-State Circuits (JSSC)*, 52(1):127–138, 2017.

[CKH+16] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, et al. Wide & Deep Learning for Recommender Systems. *Workshop on Deep Learning for Recommender Systems (DLRS)*, pages 7–10, Sep 2016.

[CL14] X.-W. Chen and X. Lin. Big Data Deep Learning: Challenges and Perspectives. *IEEE Access*, 2:514–525, 2014.

[CLK+19] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi. Deep Positron: A Deep Neural Network Using the Posit Number System. *Design, Automation, and Test in Europe (DATE)*, pages 1421–1426, 2019.

[CLL+14] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, et al. DaDianNao: A Machine-Learning Supercomputer. *Int'l Symp. on Computer Architecture (ISCA)*, pages 609–622, 2014.

[CLL+15] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang,

and Z. Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint*, arXiv:1512.01274, Dec 2015.

[CLN+11] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, Apr 2011.

[COR+16] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The Cityscapes Dataset for Semantic Urban Scene Understanding. *Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 3213–3223, Jun 2016.

[CWD99] J. Cong, C. Wu, and Y. Ding. Cut Ranking and Pruning: Enabling a General and Efficient FPGA Mapping Solution. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 29–35, 1999.

[CWV+14] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *CoRR*, abs/1410.0759, 2014.
URL `http://arxiv.org/abs/1410.0759`

[CWV+18] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan. PACT: Parameterized Clipping Activation for Quantized Neural Networks. *arXiv preprint*, arXiv:1805.0608, May 2018.

[CYF+15] Y. Cheng, F. X. Yu, R. S. Feris, S. Kumar, A. Choudhary, and S.-F. Chang. An Exploration of Parameter Redundancy in Deep Networks with Circulant Projections. *Int'l Conf. on Computer Vision (ICCV)*, pages 2857–2865, 2015.

[D+18] S. Davidson et al. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro*, 38(2):30–41, 2018.

[DCLT18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint*, arXiv:1810.04805, Oct 2018.

[DCM+12] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, et al. Large Scale Distributed Deep Networks. *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1223–1231, Dec 2012.

[DDS+09] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-

Scale Hierarchical Image Database. *Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.

[DGR+74] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *Journal of Solid-State Circuits (JSSC)*, 9(5):256–268, Oct 1974.

[DLW+17] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, and B. Yuan. CirCNN: Accelerating and Compressing Deep Neural Networks using Block-Circulant Weight Matrices. *Int'l Symp. on Microarchitecture (MICRO)*, pages 395–408, 2017.

[DLX+19] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan. PermDNN: Efficient Compressed Deep Neural Network Architecture with Permuted Diagonal Matrices . *Int'l Symp. on Microarchitecture (MICRO)*, Oct 2019.

[Doe16] C. Doersch. Tutorial on Variational Autoencoders. *arXiv preprint*, arXiv:1606.05908, Jun 2016.

[DPY18] J. Dean, D. Patterson, and C. Young. A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution. *IEEE Micro*, 38(2):21–29, Jan 2018.

[DSSMS16] E. Del Sozzo, A. Solazzo, A. Miele, and M. D. Santambrogio. On the Automation of High Level Synthesis of Convolutional Neural Networks. *Int'l Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 217–224, May 2016.

[DTHZ14] S. Dai, M. Tan, K. Hao, and Z. Zhang. Flushing-Enabled Loop Pipelining for High-Level Synthesis. *Design Automation Conf. (DAC)*, Jun 2014.

[ea15] M. A. et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org.
URL `http://tensorflow.org/`

[EVGW+10] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman. The Pascal Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision (IJCV)*, 88(2):303–338, Sep 2010.

[FS94] A. H. Farrahi and M. Sarrafzadeh. Complexity of the Lookup-Table Minimization Problem for FPGA Technology Mapping. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 13(11):1319–1332, 1994.

[GEB16] L. A. Gatys, A. S. Ecker, and M. Bethge. Image Style Transfer Using Convolutional Neural Networks. *Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 2414–2423, Jun 2016.

[GPAM+14] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Nets. *Advances in Neural Information Processing Systems (NeurIPS)*, pages 2672–2680, 2014.

[HCS+17] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *Journal of Machine Learning Research (JMLR)*, 18(187):1–30, 2017.

[HKM+17] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, et al. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017.

[HLM+16] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. *Int'l Symp. on Computer Architecture (ISCA)*, pages 243–254, 2016.

[HLWvdM17] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten. Densely connected convolutional networks. *Conf. on Computer Vision and Pattern Recognition (CVPR)*, 1(2):3, 2017.

[HMD16] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *Int'l Conf. on Learning Representations (ICLR)*, Feb 2016.

[Hor12] K. J. Horadam. *Hadamard Matrices and Their Applications*. Princeton university press, 2012.

[HPN+17] S. Han, J. Pool, S. Narang, H. Mao, E. Gong, S. Tang, E. Elsen, P. Vajda, M. Paluri, J. Tran, B. Catanzaro, and W. J. Dally. DSD: Dense-Sparse-Dense Training for Deep Neural Networks. *Int'l Conf. on Learning Representations (ICLR)*, Feb 2017.

[HPTD15] S. Han, J. Pool, J. Tran, and W. Dally. Learning Both Weights and Connections for Efficient Neural Network. *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1135–1143, 2015.

[HSL+16] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger. Deep Networks with Stochastic Depth. *European Conference on Computer Vision (ECCV)*, pages 646–661, 2016.

[HZC+17] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient Convolutional Neural Networks for Nobile Vision Applications. *arXiv preprint*, arXiv:1704.04861, 2017.

[HZRS15] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *arXiv preprint*, arXiv:1512.0338, Dec 2015.

[IRCC17] Y. Ioannou, D. Robertson, R. Cipolla, and A. Criminisi. Deep Roots: Improving CNN Efficiency with Hierarchical Filter Groups. *Conf. on Computer Vision and Pattern Recognition (CVPR)*, Jun 2017.

[IS15] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv preprint*, arXiv:1502.03167, Mar 2015.

[JAFF16] J. Johnson, A. Alahi, and L. Fei-Fei. Perceptual Losses for Real-Time Style Transfer and Super-Resolution. *European Conference on Computer Vision (ECCV)*, pages 694–711, Oct 2016.

[JKC+18] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 2704–2713, Jun 2018.

[JSD+14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint*, arXiv:1408.5093, 2014.

[JSH+18] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, et al. Highly Scalable Deep Learning Training System with Mixed-Precision: Training Imagenet in Four Minutes. *arXiv preprint*, arXiv:1807.11205, Jul 2018.

[JYP+17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. *Int'l Symp. on Computer Architecture (ISCA)*, pages 1–12, 2017.

[KAY17] D. Kim, J. Ahn, and S. Yoo. ZeNA: Zero-Aware Neural Network Accelerator. *IEEE Design & Test*, 35(1):39–46, Aug 2017.

[KC15] D. K. Kim and T. Chen. Deep Neural Network for Real-Time Autonomous Indoor Navigation. *arXiv preprint*, arXiv:1511.04668, Nov 2015.

[KH09] A. Krizhevsky and G. Hinton. Learning Multiple Layers of Features from Tiny Images. *Tech report*, 2009.

[KMM+19] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, et al. A Study of BFLOAT16 for Deep Learning Training. *arXiv preprint*, arXiv:1905.12322, May 2019.

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1097–1105, 2012.

[LCY13] M. Lin, Q. Chen, and S. Yan. Network in Network. *arXiv preprint*, arXiv:1312.4400, 2013.

[LFDA16] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-End Training of Deep Visuomotor Policies. *Journal of Machine Learning Research (JMLR)*, 17(1):1334–1373, Apr 2016.

[LHP+15] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous Control with Deep Reinforcement Learning. *arXiv preprint*, arXiv:1509.02971, Sep 2015.

[LL16] F. Li and B. Liu. Ternary Weight Networks. *arXiv preprint*, arXiv:1605.04711, May 2016.

[LLW+19] J. Lu, S. Lu, Z. Wang, C. Fang, J. Lin, Z. Wang, and L. Du. Training Deep Neural Networks Using Posit Number System. *arXiv preprint*, arXiv:1909.03831, Sep 2019.

[LMB+14] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: Common Objects in Context. *European Conference on Computer Vision (ECCV)*, pages 740–755, 2014.

[LPM15] M.-T. Luong, H. Pham, and C. D. Manning. Effective Approaches to Attention-Based Neural Machine Translation. *arXiv preprint*, arXiv:1508.04025, Aug 2015.

[LTH+17] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, et al. Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. *Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 4681–4690, Jun 2017.

[LWL+18] Z. Liu, B. Wu, W. Luo, X. Yang, W. Liu, and K.-T. Cheng. Bi-Real Net: Enhancing the Performance of 1-bit CNNs with Improved Representational Capability and

Advanced Training Algorithm. *European Conference on Computer Vision (ECCV)*, September 2018.

[LYBP16] J. Lu, J. Yang, D. Batra, and D. Parikh. Hierarchical Question-Image Co-Attention for Visual Question Answering. *Advances in Neural Information Processing Systems (NeurIPS)*, pages 289–297, Dec 2016.

[LZP17] X. Lin, C. Zhao, and W. Pan. Towards Accurate Binary Convolutional Neural Network. *Advances in Neural Information Processing Systems (NeurIPS)*, pages 344–352, 2017.

[MBY+18] B. Moons, D. Bankman, L. Yang, B. Murmann, and M. Verhelst. BinarEye: An Always-On Energy-Accuracy-Scalable Binary CNN Processor with All Memory on Chip in 28nm CMOS. *Custom Integrated Circuits Conf. (CICC)*, pages 1–4, May 2018.

[MDAdF16] M. Moczulski, M. Denil, J. Appleyard, and N. de Freitas. ACDC: A Structured Efficient Linear Layer. *Int'l Conf. on Learning Representations (ICLR)*, 2016.

[MDQ13] A. Morvan, S. Derrien, and P. Quinton. Polyhedral Bubble Insertion: a Method to Improve Nested Loop Pipelining for High-Level Synthesis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 32(3):339–352, 2013.

[MFAG19] E. Meller, A. Finkelstein, U. Almog, and M. Grobman. Same, Same but Different - Recovering Neural Network Quantization Error through Weight Factorization. *Int'l Conf. on Machine Learning (ICML)*, Jun 2019.

[Mig17] S. Migacz. 8-bit Inference with TensorRT. *NVIDIA GPU Technology Conference*, May 2017.

[MKS+15] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-Level Control through Deep Reinforcement Learning. *Nature*, 518(7540):529, Feb 2015.

[MNA+17] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, et al. Mixed Precision Training. *arXiv preprint*, arXiv:1710.03740, Oct 2017.

[MP18] P. G. Mousouliotis and L. P. Petrou. SqueezeJet: High-Level Synthesis Accelerator Design for Deep Convolutional Neural Networks. *Int'l Symp. on Applied Reconfigurable Computing (ARC)*, pages 55–66, Apr 2018.

[MXBS16] S. Merity, C. Xiong, J. Bradbury, and R. Socher. Pointer Sentinel Mixture Models. *arXiv preprint*, arXiv:1609.07843, Sep 2016.

[NSW18] D. H. Noronha, B. Salehpour, and S. J. Wilton. LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks. *Int'l Workshop on FPGAs for Software Programmers (FSP)*, pages 1–8, Aug 2018.

[NvBBW19] M. Nagel, M. van Baalen, T. Blankevoort, and M. Welling. Data-Free Quantization Through Weight Equalization and Bias Correction. *Int'l Conf. on Computer Vision (ICCV)*, Oct 2019.

[Đok08] D. Ž. Đoković. Hadamard Matrices of Order 764 Exist. *Combinatorica*, 28(4):487–489, 2008.

[OOS17] A. Odena, C. Olah, and J. Shlens. Conditional Image Synthesis with Auxiliary Classifier GANs. *Int'l Conf. on Machine Learning (ICML)*, pages 2642–2651, Jun 2017.

[ORK+15] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung. Accelerating Deep Convolutional Neural Networks using Specialized Hardware. *Microsoft Research Whitepaper*, 2(11), 2015.

[Pan01] V. Y. Pan. *Structured Matrices and Polynomials: Unified Superfast Algorithms*. Springer, 2001.

[PGC+17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic Differentiation in PyTorch. *Advances in Neural Information Processing Systems Workshops (NIPS-W)*, 2017.

[PKA69] W. K. Pratt, J. Kane, and H. C. Andrews. Hadamard Transform Image Coding. *Proceedings of the IEEE*, 57(1):58–68, 1969.

[PKL98] P. Pan, A. K. Karandikar, and C. Liu. Optimal Clock Period Clustering for Sequential Circuits with Retiming. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 17(6):489–498, 1998.

[PKY18] E. Park, D. Kim, and S. Yoo. Energy-Efficient Neural Network Accelerator Based on Outlier-Aware Low-Precision Computation. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2018.

[PL98] P. Pan and C.-C. Lin. A New Retiming-Based Technology Mapping Algorithm for LUT-based FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 35–42, 1998.

[PYV18] E. Park, S. Yoo, and P. Vajda. Value-aware Quantization for Training and Inference of Neural Networks. *arXiv preprint*, arXiv:1804.07802, Apr 2018.

[QWY+16] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, et al. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 26–35, Feb 2016.

[Rau94] B. R. Rau. Iterative Modulo Scheduling: an Algorithm for Software Pipelining Loops. *Int'l Symp. on Microarchitecture (MICRO)*, pages 63–74, Nov 1994.

[RMC15] A. Radford, L. Metz, and S. Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *arXiv preprint*, arXiv:1511.06434, Nov 2015.

[RORF16] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *European Conference on Computer Vision (ECCV)*, Oct 2016. ArXiv:1603.05279.

[RRWN11] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A Lock-free Approach to Parallelizing Stochastic Gradient Descent. *Advances in Neural Information Processing Systems (NeurIPS)*, pages 693–701, Dec 2011.

[RWC+19] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are Unsupervised Multitask Learners. *unpublished manuscript*, 2019. URL {https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf}

[RZLL16] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. SQuAD: 100,000+ Questions for Machine Comprehension of Text. *arXiv preprint*, arXiv:1606.05250, 2016.

[SA03] S. Savchev and T. Andreescu. *Mathematical Miniatures*, chapter 12. Hermite's Identity, pages 41–44. Mathematical Association of America, 2003.

[SCC+19] X. Sun, J. Choi, C.-Y. Chen, N. Wang, S. Venkataramani, V. V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan. Hybrid 8-bit Floating Point (HFP8) Training and Inference for Deep Neural Networks. *Advances in Neural Information Processing Systems (NeurIPS)*, pages 4901–4910, Dec 2019.

[SCD+16] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao. Throughput-Optimal OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 16–25, Feb 2016.

[SFRO17] C. D. Sa, M. Feldman, C. Ré, and K. Olukotun. Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[SGM19] E. Strubell, A. Ganesh, and A. McCallum. Energy and Policy Considerations for Deep Learning in NLP. *arXiv preprint*, arXiv:1906.02243, Jun 2019.

[SHG+14] Y. Shen, X. He, J. Gao, L. Deng, and G. Mesnil. Learning Semantic Representations using Convolutional Neural Networks for Web Search. *Int'l Conf. on World Wide Web*, pages 373–374, Apr 2014.

[SHM+16] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the Game of Go with Deep Neural Networks and Tree Search. *nature*, 529(7587):484, Jan 2016.

[SHS16] S. Shin, K. Hwang, and W. Sung. Fixed-Point Performance Analysis of Recurrent Neural Networks. *Int'l Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, pages 976–980, 2016.

[Sif14] L. Sifre. Rigid-Motion Scattering for Image Classification. *Ph.D. thesis*, 2014.

[SLJ+15] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going Deeper with Convolutions. *Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[SLLW18] K. Sun, M. Li, D. Liu, and J. Wang. IGCV3: Interleaved Low-Rank Group Convolutions for Efficient Deep Neural Networks. *arXiv preprint*, arXiv:1806.00178, Jun 2018.

[SSH15] W. Sung, S. Shin, and K. Hwang. Resiliency of Deep Neural Networks Under Quantization. *arXiv preprint arXiv:1511.06488*, 2015.

[SSK15] V. Sindhwani, T. Sainath, and S. Kumar. Structured Transforms for Small-Footprint Deep Learning. *Advances in Neural Information Processing Systems (NeurIPS)*, pages 3088–3096, 2015.

[SSS+17] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the Game of Go Without Human Knowledge. *Nature*, 550(7676):354, Oct 2017.

[SVL14] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to Sequence Learning with Neural

Networks. *Advances in Neural Information Processing Systems (NeurIPS)*, pages 3104–3112, Dec 2014.

[SZ15] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint*, arXiv:1409.15568, Apr 2015.

[TDGZ15] M. Tan, S. Dai, U. Gupta, and Z. Zhang. Mapping-Aware Constrained Scheduling for LUT-Based FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2015.

[The16] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint*, arXiv:1605.02688, May 2016. URL `http://arxiv.org/abs/1605.02688`

[TLDZ14] M. Tan, B. Liu, S. Dai, and Z. Zhang. Multithreaded Pipeline Synthesis for Data-Parallel Kernels. *Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 718–725, Nov 2014.

[TYW+19] T. Tambe, E.-Y. Yang, Z. Wan, Y. Deng, V. J. Reddi, A. Rush, D. Brooks, and G.-Y. Wei. AdaptivFloat: A Floating-point based Data Type for Resilient Deep Learning Inference. *arXiv preprint*, arXiv:1909.13271, Sep 2019.

[UFG+17] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 65–74, Feb 2017.

[VB16] S. I. Venieris and C.-S. Bouganis. fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, May 2016.

[VBC+19] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. M. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, et al. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. *DeepMind Blog*, Jan 2019.

[VL15] O. Vinyals and Q. Le. A neural Conversational Model. *arXiv preprint*, arXiv:1506.05869, Jun 2015.

[Wal76] J. S. Wallis. On the Existence of Hadamard Matrices. *Journal of Combinatorial Theory, Series A*, 21(2):188–195, 1976.

[WDL+18] Y. Wang, C. Ding, Z. Li, G. Yuan, S. Liao, X. Ma, B. Yuan, X. Qian, J. Tang, Q. Qiu, and X. Lin. Towards Ultra-High Performance and Energy Efficiency of

Deep Learning Systems: An Algorithm-Hardware Co-Optimization Framework. *AAAI Conf' on Artificial Intelligence (AAAI)*, Feb 2018.

[WLCS18] S. Wu, G. Li, F. Chen, and L. Shi. Training and Inference with Integers in Deep Neural Networks. *Int'l Conf. on Learning Representations (ICLR)*, May 2018.

[WLD+18] S. Wang, Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang, and Y. Liang. C-LSTM: Enabling Efficient LSTM using Structured Compression Techniques on FPGAs. *to appear in International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2018.

[WSC+16] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv preprint*, arXiv:1609.08144, Sep 2016.

[WWY15] H. Wang, N. Wang, and D.-Y. Yeung. Collaborative Deep Learning for Recommender Systems. *Int'l Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 1235–1244, Aug 2015.

[WXH+16] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li. DeepBurning: Automatic Generation of FPGA-based Learning Accelerators for the Neural Network Family. *Design Automation Conf. (DAC)*, page 110, Jun 2016.

[XGD+17] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated Residual Transformations for Deep Neural Networks. *Conf. on Computer Vision and Pattern Recognition (CVPR)*, Jun 2017.

[XWZ+18] G. Xie, J. Wang, T. Zhang, J. Lai, R. Hong, and G.-J. Qi. IGCV2: Interleaved Structured Sparse Convolutional Neural Networks. *arXiv preprint*, arXiv:1804.06202, Apr 2018.

[XXC12] J. Xie, L. Xu, and E. Chen. Image Denoising and Inpainting with Deep Neural Networks. *Advances in Neural Information Processing Systems (NeurIPS)*, pages 341–349, Dec 2012.

[XZY+18] X. Xu, X. Zhang, B. Yu, X. S. Hu, C. Rowen, J. Hu, and Y. Shi. DAC-SDC Low Power Object Detection Challenge for UAV Applications. *arXiv preprint*, arXiv:1809.00110, 2018.

[YCS16] T. Yang, Y. Chen, and V. Sze. Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning. *Int'l Conf. on Learning Representations (ICLR)*, arXiv:1611.05128, 2016.

[YCYL+17] R. A. Yeh, C. Chen, T. Yian Lim, A. G. Schwing, M. Hasegawa-Johnson, and M. N. Do. Semantic Image Inpainting with Deep Generative Models. *Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 5485–5493, Jun 2017.

[YGP+18] X. Yang, M. Gao, J. Pu, A. Nayak, Q. Liu, S. E. Bell, J. O. Setter, K. Cao, H. Ha, C. Kozyrakis, and M. Horowitz. DNN Dataflow Choice Is Overrated. *arXiv preprint*, arXiv:1809.04070, Sep 2018.

[YLC+18] R. Yu, A. Li, C.-F. Chen, J.-H. Lai, V. I. Morariu, X. Han, M. Gao, C.-Y. Lin, and L. S. Davis. NISP: Pruning Networks using Neuron Importance Score Propagation. *arXiv preprint*, arXiv:1711.05908, 2018.

[ZDSZ19] R. Zhao, C. De Sa, and Z. Zhang. Overwrite Quantization: Opportunistic Outlier Handling for Neural Network Accelerators. *arXiv preprint*, arXiv:1910.06909, Oct 2019.

[ZDZ+16] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. Cambricon-X: An Accelerator for Sparse Neural Networks. *Int'l Symp. on Microarchitecture (MICRO)*, pages 1–12, 2016.

[ZGRC14] H. Zheng, S. T. Gurumani, K. Rupnow, and D. Chen. Fast and Effective Placement and Routing Directed High-Level Synthesis for FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 1–10, 2014.

[ZHD+19a] R. Zhao, Y. Hu, J. Dotzel, C. De Sa, and Z. Zhang. Improving Neural Network Quantization without Retraining using Outlier Channel Splitting. *Int'l Conf. on Machine Learning (ICML)*, pages 7543–7552, Jun 2019.

[ZHD+19b] R. Zhao, Y. Hu, J. Dotzel, C. D. Sa, and Z. Zhang. Building Efficient Deep Neural Networks with Unitary Group Convolutions. *Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 11303–11312, Jun 2019.

[ZL13] Z. Zhang and B. Liu. SDC-Based Modulo Scheduling for Pipeline Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, pages 211–218, Nov 2013.

[ZL17] J. Zhang and J. Li. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 25–34, Feb 2017.

[ZLH+19] X. Zhang, Y. Li, C. Hao, K. Rupnow, J. Xiong, W.-m. Hwu, and D. Chen. SkyNet: A Champion Model for DAC-SDC on Low Power Object Detection. *arXiv preprint*, arXiv:1906.10327, Jun 2019.

[ZLS+15] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 161–170, Feb 2015.

[ZMK+17] Y. Zhu, R. Mottaghi, E. Kolve, J. J. Lim, A. Gupta, L. Fei-Fei, and A. Farhadi. Target-Driven Visual Navigation in Indoor Scenes using Deep Reinforcement Learning. *Int'l Conf. on Robotics and Automation (ICRA)*, pages 3357–3364, May 2017.

[ZQXW17] T. Zhang, G.-J. Qi, B. Xiao, and J. Wang. Interleaved Group Convolutions. *Conf. on Computer Vision and Pattern Recognition (CVPR)*, Jun 2017.

[ZST+18] B. Zhuang, C. Shen, M. Tan, L. Liu, and I. Reid. Towards Effective Low-Bitwidth Convolutional Neural Networks. *Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 7920–7928, Jun 2018.

[ZSV14] W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent Neural Network Regularization. *arXiv preprint*, arXiv:1409.2329, 2014.

[ZSZ+17] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang. Accelerating Binarized Convolutional Neural Networks with Software-programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017.

[ZTDZ15] R. Zhao, M. Tan, S. Dai, and Z. Zhang. Area-Efficient Pipelining for FPGA-Targeted High-Level Synthesis. *Design Automation Conf. (DAC)*, Jun 2015.

[ZWLS10] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized Stochastic Gradient Descent. *Advances in Neural Information Processing Systems (NeurIPS)*, Dec 2010.

[ZWN+16] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. DoReFar-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv preprint*, arXiv:1606.06160, Jul 2016.

[ZXL+17] H. Zhang, T. Xu, H. Li, S. Zhang, X. Wang, X. Huang, and D. N. Metaxas. Stack-GAN: Text to Photo-Realistic Image Synthesis with Stacked Generative Adversarial Networks. *Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 5907–5915, Jun 2017.

[ZYG+17] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen. Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights. *arXiv preprint*, arXiv:1702.03044, 2017.

[ZZLS17] X. Zhang, X. Zhou, M. Lin, and J. Sun. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. *arXiv preprint*, arXiv:1707.01083, Aug 2017.

[ZZZ+10] J. Zhang, Z. Zhang, S. Zhou, M. Tan, X. Liu, X. Cheng, and J. Cong. Bit-level Optimization for High-Level Synthesis and FPGA-Based Acceleration. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 59–68, 2010.