Title
Co-design ofAlgorithms, Hardware, and Scheduling for Deep Learning Applications

Author
Huang, Qijing

Publication Date
2021

Peer reviewed|Thesis/dissertation

Co-design of Algorithms, Hardware, and Scheduling for Deep Learning Applications

by

Qijing Huang

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor John Wawrzynek, Chair
Professor Yakun Sophia Shao
Professor Joseph E. Gonzalez
Doctor Aravind Kalaiah

Summer 2021

Co-design of Algorithms, Hardware, and Scheduling for Deep Learning Applications

Abstract

Co-design of Algorithms, Hardware, and Scheduling for Deep Learning Applications

by

Qijing Huang

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor John Wawrzynek, Chair

For decades, ever-increasing computing power has been a driving force behind many technology revolutions, including the recent advances in artificial intelligence. However, due to the slowing of integrated circuit process scaling, for system architects to continue to satisfy the ever-growing compute appetite of today's applications, they must now resort to employing heterogeneous systems with specialized accelerators.

Building these accelerator systems, though, is extremely expensive and time-consuming. First, the development cycle for hardware is notoriously long, making it difficult to keep up with the rapid progress in algorithms. Meanwhile, existing compilers are incapable of navigating the intractable mapping space exposed by the novel accelerator architectures. Lastly, algorithms are often designed without hardware efficiency as a key metric, and therefore, pose extra challenges in designing efficient hardware.

This thesis tackles the significant challenges in jointly designing and optimizing algorithms, scheduling, and hardware designs for acceleration. We aim to advance the state-of-the-art through a three-pronged approach: the development of methodologies and tools that automatically generate accelerator systems from high-level abstractions, shortening the hardware development cycle; the adaptation of machine learning and other optimization techniques to improve accelerator design and compilation flows; and the co-design of algorithms and accelerators to exploit more optimization opportunities.

The target application domain of this thesis is deep learning which has achieved unprecedented success in a wide range of tasks such as computer vision, neural language processing, etc. As intelligent devices prevail, deep learning is foreseeably becoming a major computation demand in our everyday life. Therefore, by performing end-to-end system optimization with hardware acceleration, the dissertation aims to unleash the ubiquitous adoption of cutting-edge deep learning algorithms to transform various aspects of life.

To my parents Shaobin and Lianhua, who taught me about love and faith.
To my friends, who taught me that giving is receiving.

Contents

List of Figures

List of Tables

Acknowledgments

This thesis would not have been possible without the help, support, and guidance of many wonderful individuals.

First and foremost, I want to thank my advisor, John Wawrzynek. My journey at Berkeley would not have started without John, who recruited me to his research group. He has been very kind and supportive and has taught me many important lessons in research and life. He has frequently encouraged me to express my opinions and patiently listened to my immature research thoughts in the early years. His patience makes it possible for me to mature as an engineer and a researcher who can think critically and articulate ideas confidently. He allowed me to pursue my interests without questioning my ability to work on new topics. His trust and support have made my Ph.D. incredibly enjoyable and rewarding. His fundamental and insightful research advice has taught me how to view and approach novel problems in hardware design and computer architecture. Besides research, he truly cares about my well-being and makes active efforts to create a friendly environment for me to thrive. I learned to keep an easy mind in graduate school from the life wisdom he shared in various conversations. I'm grateful to have John as my advisor and wish I could keep him as a lifetime mentor and friend.

I would also like to express my gratitude to other faculty members at Berkeley, especially Yakun Sophia Shao, Joseph Gonzalez, Kurt Keutzer, Krste Asanovic, James Demmel, Ion Stoica, and Alan Mishchenko. Sophia has had numerous meetings with me to provide detailed feedback and guidance for my research and career. She is an exceptional academic model to me. Joey has always been very kind and resourceful. He has offered me a lot of helpful advice for academic career development. Kurt has given me insightful feedback from the machine learning perspective and set up a welcoming environment for collaboration with his group. Krste has generously shared his experience in computer architecture and circuit design. Jim has given me his detailed and helpful advice on optimization problems. Ion has helped us proofread papers and recommended me good career development strategies. Alan has discussed many of his high-impact projects with me and has helped broaden my horizon. I appreciate all their time and effort in sharing their expertise with junior researchers like me. Outside of school, I'm fortunate to have many amazing industrial mentors, who have introduced me to challenging research problems, and generously shared their advice in various areas of life and research: Aravind Kalaiah, Hamid Shojaei, Azade Nazi, Sat Chatterjee, Shobha Vasudevan, Azalia Mirhoseini, Yuandong Tian, and Randy Huang. I want to thank them for positively influencing me in various ways. I also want to give special thanks to Aravind, Sophia, and Joey, for agreeing to be on my qualify and dissertation committee.

I'm thankful to have opportunities to collaborate with many bright and dedicated researchers during graduate school time. I also would like to thank them for their contributions to the projects:

1. On Algorithm-Hardware Co-design: Zhen Dong, Yizhao Gao, Bichen Wu, Yifan Yang, Amir Golami, Dequan Wang, Zhewei Yao, Tianjun Zhang, Yaohui Cai, Tian Li. I

learned a great deal about the advancement in machine learning and computer vision from them.

2. On High-Level Synthesis and Centrifuge: Christopher Yarp, Sagar Karandikar, Nathan Pemberton, Benjamin Brock, Liang Ma, Guohao Dai, Robert Quitt, Shaoyi Cheng, Tan Nguyen, Arya Reais-Parsi. They showed their exceptional engineering skills and expertise in subjects including HLS, hardware, systems, and compilers.

3. On Machine Learning for Hardware and AutoPhase: Ameer Haj-Ali, William Moses, John Xiang, Keertana Settaluri, Hasan Genc. It was a great pleasure exploring new topics with these creative and productive individuals.

4. On integer linear programming and CoSA Scheduling: Grace Dinh, Josh Kang, Thomas Norell, Charles Hong. I learned many optimization techniques from them.

My growth as a graduate student has been dependent on many of my colleagues who generously share their compassion and technical expertise. I want to thank James Martin, Angie Wang, David Biancolin, Howard Mao, Albert Magyar, Lisa Wu, Seah Kim, Adam Izraelevitz, Dayeol Lee, Yang You, Marquita Ellis, Alon Amid, Albert Ou, Colin Schmidt, Jerry Zhao, Gilbert Bernstein, and many others, for creating an inclusive lab and department environment. I would like to express my gratitude to Kosta Ilov, who was always there to help me with various infrastructure issues.

Outside of research, I gained tremendous support from my friends at Berkeley. I want to thank Cecilia Zhang, Xin Wang, Tianjun Zhang, Zhewei Yao, Zixi Hu, Nathan Pemberton, Ameer Haj-Ali, Sagar Karandikar, Paras Jain, Biye Jiang, Isla Yang, Alice Ye, Cindy Chen, Jessica Zhou, Daniel Seita, Michael Chang, and Danyang Zhuo. I appreciate their company in various activities, including the numerous boba trips. I also gain lots of care and love from my IGSM friends: Sara Hong, Wenxia Lin, Jenny Chung, Grace Borja, and many others.

I'd also like to thank my friends from the University of Toronto: Angela Xu, Anqi Duan, Joshua Chou, Chain Zhang, Jack Luo, Suya Liu, Jenny Ren, Jacky Liu, Lucia Xu, Dean Xu, Xiang Chen. They showed extraordinary kindness to me, and I always think of them as family. In addition, I want to thank my undergraduate research group, led by Prof. Jason Anderson, for offering research opportunities for undergraduate students and being incredibly encouraging and supportive to us.

I am extremely thankful to have the opportunity to interact with many extraordinary individuals and form a community together. I might not have mentioned all of you here, but you have made my journey full of fun and inspired me in various ways. Through you, I learned that we are all connected people and that something we do will affect others. Moving forward, I hope I could be like you and do my little things to enlighten the days of others.

Lastly, I cannot thank my family enough for their unceasing support and inspiration from the beginning, in ways and depths that no words can express. They have taught me to work hard, persevere, and seek truth in all circumstances.

Chapter 1

Introduction

I started my graduate study aiming to answer the following question:

How do we design the most efficient hardware possible?

Such a question would not be meaningful without defining the proper *context*. To start the research, we first identified four necessary contexts for understanding efficient hardware development.

Degree of Specializaton. The hardware efficiency is limited by the inherent workload properties, i.e., whether general function support is required. The more functions to support, the more complex the logic needs to be. There is a inherent trade-off between generality and efficiency in hardware architecture design. General-purpose processors, such as CPU and GPU, are Turing-complete and flexible enough to accomplish arbitrary functions. Accelerators, contrarily, do not guarantee Turing completeness, and only support specific functions. The benefits of specialization include minimization of overhead in control logic, more aggressive domain-specific optimizations, etc. Given the same technology node and physical constraints, accelerators can achieve a significant performance improvement compared to general-purpose processors.

Development Cycle and Cost. The second context to consider is the expected time-to-market and costs. Generally speaking, the more human hours and resources spent in development, the more optimized the hardware is likely to be. However, too long of a development cycle can cause the products to be irrelevant by the time they go on the market because many applications are rapidly evolving. In addition, there is an associated non-recurring engineering (NRE) cost that restricts the resources we can dedicate to hardware design and optimization. As a result, we cannot assume unlimited time and resources for hardware development.

End-to-end Systems. Hardware does not work in isolation. Its compatibility with algorithms and software also significantly impacts the overall task execution performance and efficiency. The commonly used hardware specifications (e.g., FLOPs, TOPs) do not reflect the actual performance for running specific algorithms. Depending on the operational intensity of the algorithm design, the peak hardware performance can differ. Similarly, we

cannot evaluate the hardware performance properly without an optimized software stack. Therefore, in addition to improving the theoretical hardware peak performance, we should co-consider the algorithm design and software optimization to achieve more powerful system-level performance.

Target Application Domains. Different application domain presents different optimization opportunities for exploiting parallelism and data locality. The theoretically attainable speedup of different functions in an application is bounded by their inherent degrees of parallelism and operational intensity. According to Amdahl's law, the overall maximum speedup for the application is limited by its non-parallelizable portion. Furthermore, the popularity of the applications justifies whether it is cost-effective to develop specialized hardware support for the application. We need to closely examine the achievable speedup, cost, and demand for the application to make careful design tradeoffs.

1.1 Challenges and Opportunities

In this thesis, our target application domain is **deep learning** (DL). Deep learning has made a revolutionary impact on numerous real-world applications, making it a staple workload in modern-day computation. Since the major computation in deep learning comes from a fixed set of operations, developing accelerators for DL has become feasible and popular. DL accelerators have achieved remarkable advancement in performance in terms of latency, throughput, power efficiency, etc. According to a Google paper [98], the TOPs per watt of TPUv2 is 30 to 80 times better than the contemporary CPU or GPU. In this section, we identify three challenges from the modern-day development of the deep learning acceleration system.

Fast Algorithm Evolvement. The first challenge comes from the rapidly evolving algorithms. There has been an explosion in deep learning algorithm designs since 2012 to suit the needs of various tasks and deployment scenarios. The neural network structures have become deeper with more complex and dynamic connections among layers [78,110,198,200,209]. According to the OpenAI's AI Moore's Law, the computation required by the largest AI training doubles every 3.5 months, outpacing the actual hardware Moore's Law [144]. The time-to-market of deep learning accelerator is thus critical to its success. Hence, in addition to the original goal of improving hardware efficiency, we think optimizing the overall development cycle and design flow is as important.

Accuracy-driven Algorithm Design. Besides, algorithms are designed with task accuracy as the key metric, paying secondary attention to their efficiency and compatibility running on hardware. Meanwhile, the hardware designers often develop accelerators without modifications to the algorithms. Such practice leaves a large room for improvement on the table. Here, we notice an obvious optimization opportunity if we tailor the algorithm designs to hardware features and co-optimize the algorithms and accelerators.

Intractable Scheduling Space. Another challenge stems from the more complex and diverse deep learning algorithms and hardware accelerator designs. Scheduling, which maps

the algorithmic states to various hardware resources, has become both resource and time-consuming. The scheduling space is exponentially increasing with the ever-growing complexity of the algorithm and hardware designs. However, we must be able to navigate this intractable search space as the best schedule and worst schedule can result in orders of magnitudes difference in performance. Therefore, we should also pay attention to the software stack optimization for finding schedules that maximize the overall system performance.

To refine the original research question, here is the topic this thesis aims to discuss:

> *How to develop the most efficient accelerator systems for deep learning in a timely and cost-effective manner?*

1.2 Thesis Contributions

This thesis advances the state-of-the-art through a three-pronged approach: 1) the development of methodologies and tools that automatically generate accelerator systems from high-level abstractions, shortening the hardware development cycle, 2) the co-design of algorithms and accelerators to exploit more optimization opportunities, and 3) adaptation of optimization and machine learning techniques to improve accelerator design and compilation flows.

1.2.1 Hardware and Software Co-Design

In chapter 2, we introduce an automatic accelerator system generation flow to assist hardware and software co-design in exploiting optimization opportunities. Design automation eliminates mundane tasks and allows users to focus on tasks that are more challenging and creative. The first step towards an intelligent design flow for hardware acceleration is to automatically realize repetitious implementation details starting from a high-level abstraction, which can be achieved via High-Level Synthesis (HLS). Taking a systems perspective, we have developed an agile hardware development flow to automatically generate full-stack acceleration solutions leveraging HLS.

Centrifuge: Auto RISC-V Accelerator-SoC Generation via HLS. Today's systems-on-chips (SoCs) contain a multitude of accelerators to optimize for common workloads. However, this heterogeneity comes at the expense of increased hardware development time, not only including the design of individual accelerators, but also the selection and evaluation of the set of accelerators that should be included in a particular system. While analytical modeling can provide early insights into a new system, they generally do not account for effects that only manifest when an accelerator is integrated into a complex system. Therefore, we developed a flow called *Centrifuge* [92] that rapidly produces complete SoC systems with many integrated HLS-generated accelerators as specified by the user. It simulates the design quickly and cycle-accurately on FPGAs and generates complete software stacks on top, including Linux and full application frameworks. The generated accelerator system can be emulated and run interactively on cloud FPGAs through a simulation infrastructure

we built called *FireSim* [102]. Both works allow for agile design-space exploration of novel accelerator-based systems by enabling the users to easily explore and evaluate a variety of accelerators with different integration techniques. To further relieve the design burden for the user, we implemented Golang HLS, aiming to facilitate the use of FPGA accelerators for parallel workloads by domain experts without any background in hardware.

1.2.2 Algorithm and Hardware Co-Design

Chapter 3 introduces the hardware and algorithm co-design we performed and the corresponding observations. Deep Neural Networks (DNNs) have achieved unmatched accuracy in computer vision tasks at the expense of increased computational requirements. To deploy these computationally demanding DNNs in resource-constrained edge systems while satisfying real-time requirements, we perform algorithm-hardware co-design to improve the efficiency without compromising the accuracy. We had two works in co-designing the algorithms and hardware for embedded FPGA accelerators. Both aim to answer the same question: what are the most effective operations the hardware should support to achieve the highest efficiency given a specific computer vision task?

Synetgy: Image Classification with Shift Operation. In this work [218], we studied the existing algorithms and hardware design for image classification and found that the shift operation can be as competitive as the spatial aggregation in convolution. We thus co-designed a shift-only accelerator pipeline without any spatial convolution and achieved the state-of-art framerate on an embedded FPGA.

CoDeNet: Object Detection with Input-Adaptive Deformable Convolution. Unlike image classification, object detection problems are more sensitive to the spatial variance of objects, and therefore, require specialized convolutions to aggregate spatial information. To address this need, recent work introduces dynamic deformable convolution to augment regular convolutions with learned input-dependent access patterns. In *CoDeNet* [157], we developed a novel object detection pipeline with deformable convolutions. We performed a set of algorithm modifications, including irregular-access versus limited-range and fixed-shape, to allow for corresponding hardware improvements on a flexible hardware accelerator.

In both works, our high-efficiency accelerators solution reaches real-time inference speed with a tiny high-accuracy model.

1.2.3 Scheduling and Hardware Co-Design

With the emergence of numerous DNN accelerators with diverse architectures, there is a need for a fast, performant, and explainable approach to scheduling. As system design problems come with constraints, explicitly expressing these constraints as part of an optimization problem and solving the optimization allow us to decisively prune the search space. In chapter 4, we introduce a new optimization framework called *CoSA* [88] to schedule DNN accelerators and co-optimize the accelerator design with scheduling.

CoSA: Constrained Optimization for Scheduling Accelerators. The motivation for *CoSA* was to prune the invalid scheduling space that is commonly present for feedback-driven or ML-based scheduling. *CoSA* leverages the regularities in DNN operators and hardware to formulate the DNN scheduling space as a mixed-integer programming (MIP) problem with algorithmic and architectural constraints, where it can automatically generate a highly efficient schedule in a single pass. We demonstrated that our framework generates schedules that significantly outperform the state-of-the-art approaches by 2.5× across a wide range of DNN networks while improving the time-to-solution by 90× and the energy efficiency by 22%. In addition, we extended *CoSA* to optimize the hardware design and scheduling in unison and have seen promising results. The *CoSA* work presented a brand new optimization framework to address the limitations of traditional polyhedral transformation and showed significant improvement in the compilation time and performance over existing ML or exhaustive search approaches.

1.2.4 Machine Learning for Hardware

In addition to developing acceleration systems for deep learning, we are also interested in studying how to apply different machine learning techniques to challenging problems in hardware design and compiler optimizations. In Chapter 5, we discuss *AutoPhase* [73, 156], one of our works in applying deep RL to compilers and exploring the possibility of transfer learning with program features.

AutoPhase: Reinforcement Learning for Compiler Phase-Ordering. Prior solutions to phase-ordering have relied on heuristics or hand engineering to tackle this NP-hard problem. To achieve human-level performance, the compilers and design tools need to be able to capture the optimization heuristics and make sequential decisions by learning the important features of the programs and the underlying platforms. This behavior is attainable with the recent advancements in deep reinforcement learning (RL). In *AutoPhase* [73, 156], we formulate the problem as a Markov Decision Process (MDP) and solve it with RL. In our approach, the state is the current program intermediate representation. The action is the next pass to apply, and the reward is the improvement in the number of cycles. Augmented with an HLS compiler, *AutoPhase* improves generated circuit performance by 28% compared to using the -O3 compiler flag and shows promising results generalizing to thousands of different randomly generated programs.

Chapter 2

Hardware and Software Co-Design

Due to the end of Moore's law and classical scaling, architects today must resort to building heterogeneous and specialized systems to continue to satisfy the ever-growing appetite for compute of today's applications. Today's systems-on-chips (SoCs) contain a multitude of accelerators to optimize for common workloads. Building and evaluating these systems are extremely expensive and time-consuming, even in the early stages of development. In addition, software integration and optimization for the accelerator SoC often come after the chip tape-out, making it infeasible to modify the current hardware for critical performance improvement. We start this work by asking: What is a good methodology for designing accelerator SoCs that leads to 1) high end-to-end system performance, 2) low NRE costs, and 3) short time-to-market?

In this chapter, we describe a methodology and implement an open-source flow ("Centrifuge") that can rapidly generate and evaluate heterogeneous SoCs by combining an HLS toolchain with the open-source FireSim FPGA-accelerated simulation.

Our system can quickly produce complete SoC systems with many integrated HLS-generated accelerators, simulate them quickly and cycle-accurately on FPGAs, and run complete software stacks on top, including booting Linux and running full application frameworks. By integrating these tools, our methodology allows users to rapidly generate an entire hardware/software stack for a customized SoC and evaluate its end-to-end performance using cycle-exact FPGA simulation, allowing for agile design-space exploration of novel accelerator-based systems.

2.1 Accelerator Design Methodology

Modern SoC systems consist of general-purpose compute augmented with large numbers of specialized accelerators. However, this heterogeneity comes at the expense of increased hardware development time, not only including the design of individual accelerators, but also the selection and evaluation of the set of accelerators that should be included in a particular system. Early in the process of selecting a set of accelerators to include in a system, architects

frequently rely on high-level/abstract software modeling. While this kind of modeling is sufficient for early design space exploration and saves the time of traditional cycle-accurate modeling or RTL design entry, it generally does not account for effects that only manifest when an accelerator is integrated into a complex system.

To achieve a greater level of detail in evaluation, architects frequently use cycle-accurate full-system simulation platforms. These simulators span a wide range of design points, making tradeoffs in simulation accuracy, simulation performance, and ease of use. Broadly speaking, these simulators can be broken into software-based simulators and hardware-accelerated simulators. In comparison with hardware-accelerated simulation, software-based simulation is simpler to use, but requires significant modeling expertise and validation. Furthermore, due to low simulation performance, software-based cycle-accurate simulation is unable to run long-running workloads, which makes it difficult to determine if an accelerator is actually beneficial when deployed in a system. In comparison, FPGA-accelerated simulators are able to simulate systems at much higher simulation rates, but require specifying an accelerator design by writing RTL, which drastically slows down early design-space exploration. With either of these simulation techniques, a key hurdle is the fact that a design must be developed and converted into RTL or a software model, which requires a significant time investment.

To bypass this issue, High-Level Synthesis (HLS) tools have been developed, which allow users to specify designs in a more software-centric manner but produce an RTL design that can later be used in hardware-accelerated simulation. While HLS tools have traditionally been restricted to producing accelerators that run on FPGAs, recently there has been an explosion in the use of HLS tools to generate and refine accelerator designs that are ultimately integrated into a complex system and taped-out, including those at Google, NVIDIA, Bosch, Qualcomm, etc [132]. A key advantage of using HLS to generate accelerators is that the accelerators can be verified at the C-code level, and the HLS tool can be trusted to produce correct output. Verification in this form is considerably faster and more productive than traditional hardware verification [104].

2.2 Background and Motivation

With the increasing complexity of workload and systems in the datacenter, hardware-software co-design is becoming critical to truly optimize full-systems. Several projects have explored high-level modeling for accelerator design. Aladdin [176] is a software simulator that takes C code as input and estimates performance, power, and area of a target accelerator design. The program behavior is modeled with dynamic data dependence graphs (DDDG), which can be generated from C code directly. This model assumes that all data can be preloaded into the local scratchpads, which falls short for real designs with limited on-chip memory budget and complex memory access patterns. [177] addresses this issue by extending Aladdin with the gem5 full-system simulator [18] to provide support for simulating complex accelerator-system interactions. This work shows that the pareto-optimal Energy-Delay Product (EDP) points for accelerators evaluated in isolation differ from ones explored through full-system co-design.

While this approach is fast and easy to deploy, detailed accelerator design insights are difficult to gain as no true hardware is generated. Besides, its simulator speed (∼50KIPS) limits the deployment of full-stack software, whereas in our system that runs at tens of MIPS [17, 105], the real impact of accelerators can be manifested at the application level. PARADE [40] is another extension to gem5 that leverages HLS to generate accurate accelerator models for accelerator-rich architecture(ARA) on complex network-on-chips (NoCs). It provides a global accelerator manager to manage the accelerator runtime. In all of these cases however, the prior work does not move the designer towards obtaining an actual implementation—once these tools generate an accelerator design and a designer selects a particular set of accelerators, the designer must then write RTL or HLS for the accelerators or the glue logic. [151, 152] proposed a approach to design accelerator SoCs using HLS. Differing from prior works, we aim to provide a fast simulation environment to evaluate an accelerator in a full-stack setting. Our framework quickly provides a baseline set of interfaces and an easy-to-use simulation environment that software developers can program against and use for performance optimization of the software stack, even before real silicon is available.

2.2.1 Related Work

With the increasing complexity of workload and systems in the datacenter, hardware-software co-design is becoming critical to truly optimize full-systems. Several projects have explored high-level modeling for accelerator design. Aladdin [176] is a software simulator that takes C code as input and estimates performance, power, and area of a target accelerator design. The program behavior is modeled with dynamic data dependence graphs (DDDG), which can be generated from C code directly. This model assumes that all data can be preloaded into the local scratchpads, which falls short for real designs with limited on-chip memory budget and complex memory access patterns. [177] addresses this issue by extending Aladdin with the gem5 full-system simulator [18] to provide support for simulating complex accelerator-system interactions. This work shows that the Pareto-optimal Energy-Delay Product (EDP) points for accelerators evaluated in isolation differ from ones explored through full-system co-design. While this approach is fast and easy to deploy, detailed accelerator design insights are difficult to gain as no true hardware is generated. Besides, its simulator speed (∼50KIPS) limits the deployment of full-stack software, whereas in our system that runs at tens of MIPS [17, 105], the real impact of accelerators can be manifested at the application level. PARADE [40] is another extension to gem5 that leverages HLS to generate accurate accelerator models for accelerator-rich architecture(ARA) on complex network-on-chips (NoCs). It provides a global accelerator manager to manage the accelerator runtime. In all of these cases, however, the prior work does not move the designer towards obtaining an actual implementation—once these tools generate an accelerator design and a designer selects a particular set of accelerators, the designer must then write RTL or HLS for the accelerators or the glue logic. [151, 152] proposed an approach to design accelerator SoCs using HLS. Differing from prior works, we aim to provide a fast simulation environment to evaluate an accelerator in a full-stack setting. Our framework quickly provides a baseline set of interfaces and an

easy-to-use simulation environment that software developers can program against and use for performance optimization of the software stack, even before real silicon is available.

2.3 Centrifuge Overview

In this chapter, we introduce our methodology and flow for agile generation and evaluation of multi-accelerator SoCs, named *Centrifuge*[1] In Centrifuge, we proposed a methodology and developed an open-source toolchain to rapidly generate and evaluate heterogeneous SoCs by combining an HLS toolchain with the open-source FireSim [102] FPGA-accelerated simulation platform:

1. We provide a flow that generates full SoC systems containing user-defined accelerators written in HLS. This flow integrates the Rocket Chip SoC generator with custom accelerators generated with Vivado HLS. Accelerators in the generated system can be attached to the system in three ways: ① coprocessor-style RoCC accelerators, ② accelerators that connect to the SoC's on-chip network, and ③ disaggregated accelerators that attach directly to Ethernet.

2. We provide a flow that automatically generates software infrastructure to interact with the accelerators on the generated SoC systems from within accelerators.

3. We add a Verilog FAME-1 [193] pass to the open-source FireSim [102] simulator to support simulating designs that contain Chisel blackboxes of Verilog designs, in our case, the accelerator designs produced by Vivado HLS.

4. We generate SoCs with several integrated accelerators and evaluate accelerators with different coupling and software stacks. In addition, we conduct three case studies to demonstrate the capability of the toolchain.

With this methodology, we can rapidly generate an entire hardware/software stack for a customized SoC that can be fabricated as an ASIC and evaluate its end-to-end performance using FPGA simulation, allowing for rapid design-space exploration of novel accelerator-based systems, while providing cycle-exact performance measurements with little user effort. We call our implementation of this approach Centrifuge. Once a user is satisfied with the baseline accelerated system produced by Centrifuge, they can then continue to hand-optimize the design, as if they had written RTL from scratch.

2.4 Centrifuge Design Flow

In this section, we detail the key components of Centrifuge. We first describe how we build an SoC with integrated HLS-generated accelerators, then outline our extensions to the FireSim

[1]Our tool is named Centrifuge because it lets us *rapidly iterate* on novel many-accelerator SoCs and *separates* the good accelerators from the bad.

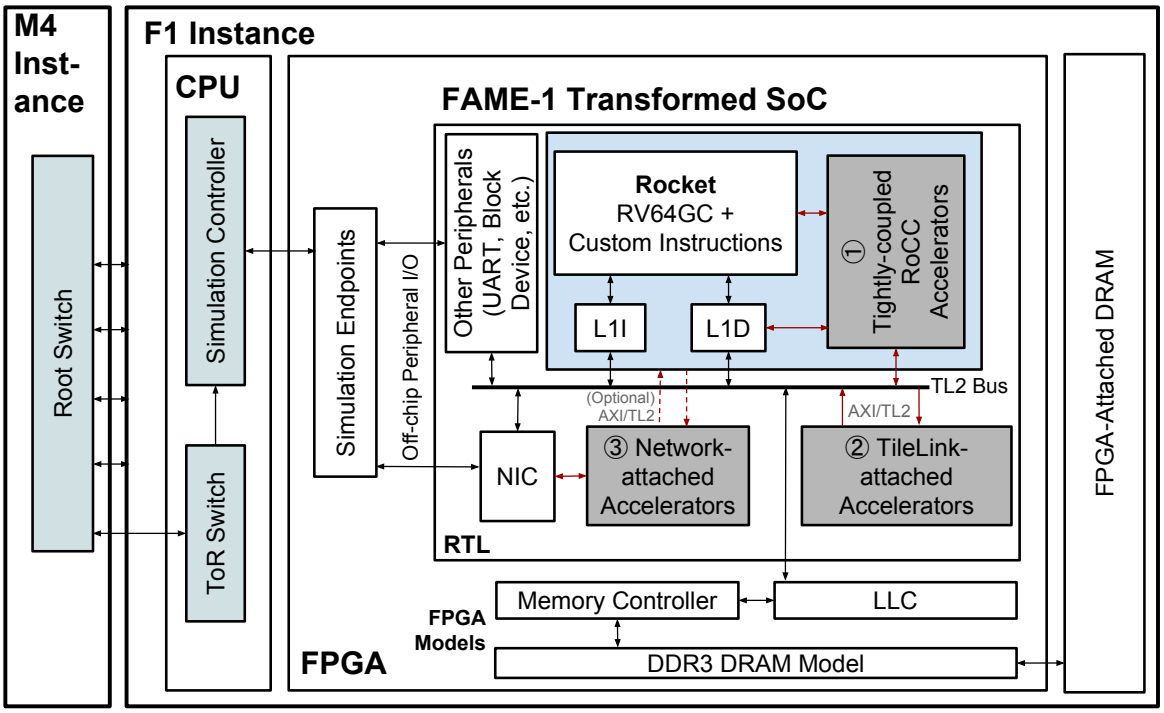


Figure 2.1: Block Diagram of FireSim Simulating Centrifuge-generated SoC with Accelerators

FPGA-accelerated simulation platform [102] to enable fast cycle-exact simulation of our generated SoCs.

2.4.1 Generating a Base SoC with Rocket Chip

As the basis for our SoC system, we use the Rocket Chip generator [8], an open-source SoC generator written in Chisel that provides standard SoC components, including the RISC-V Rocket Core (replaceable with the BOOM Out-of-Order core) and uncore components. FireSim provides several standard peripherals, including a UART, Block Device, and NIC [102]. Altogether, this produces a Linux-capable RISC-V SoC that can interface with a standard Ethernet network. The base SoC components are shown in the "RTL" box in Figure 2.1, excluding the gray accelerator boxes. We configure the system to have 16 KB L1 I/D Caches, a 4 MiB LLC, 16GB of DDR memory, and a 200 Gbit/s Ethernet NIC. The gray boxes in Figure 2.1 show three methods for integrating accelerators into the SoC. We detail these in the following section.

2.4.2 Integrating Accelerators into the SoC

To enable exploration of various accelerator designs, we supply shim infrastructure to incorporate HLS-generated accelerators into the aforementioned SoC in three distinct ways:

1. RoCC accelerators (coprocessor sharing L1 and LLC with the processor, invoked by RoCC instruction)
2. TileLink accelerators (closely-coupled accelerator sharing LLC with the processor, invoked by either RoCC instruction or memory-mapped I/Os(MMIO))
3. Network-attached accelerators (TileLink accelerators with direct connection to the Ethernet)

These three models are representative of recent academic and commercial designs.

2.4.2.1 RoCC Accelerators

The RoCC accelerator interface provided by Rocket Chip allows a user to integrate an accelerator closely with the processor in the SoC. The accelerator can receive commands directly from the general-purpose processor through a dedicated command queue. Programs can issue these commands using custom RoCC instructions that fit within the RISC-V ISA. RoCC accelerators also have ports directly into the private L1 data cache of the general-purpose core and a port into the next-level cache in the system.

2.4.2.2 TileLink-attached accelerators

Looser coupling of accelerators to a local application core's LLC is achieved by attaching accelerators to the on-chip TileLink interconnect [181] in Rocket Chip. TileLink is an open and free chip-scale cache-coherent interconnect standard used by the Rocket Chip SoC-generator to connect devices on low-latency SoC buses. It supports a MOESI-equivalent protocol to provide coherent access for an arbitrary mix of caching or non-caching masters. There are three levels of conformance protocols and five channels implemented as five physically distinct unidirectional parallel buses with one sender and one receiver on each. The completion of data transactions is out-of-order to improve throughput.

The Rocket Chip generator also takes advantage of a library called Diplomacy [127], which supports automatic parameter negotiation and checking between SoC components. In HLS C programs, pointer-type arguments to a C function are synthesized into AXI4 master ports when the m_axi interface pragma is specified. Each memory access in the C code is turned into an AXI4 request in the generated hardware. To attach accelerators with these AXI-4 memory systems generated by Vivado HLS, we use an open TileLink-to-AXI4 bridge adapter [181] to connect accelerators to the Rocket Chip SoC.

2.4.2.3 Network-attached accelerators

The last accelerator-integration option we provide is to allow accelerators to directly interface with an external Ethernet network by directly communicating with the in-SoC NIC to send/receive packets, without the intervention of the general-purpose processor. There are two ways for the accelerator to directly send and receive data to and from the network.

Accelerator MMIO to NIC. The accelerator can directly post send and receive commands to the NIC in the SoC by accessing the NIC's MMIO control registers in the HLS code. The send and receive buffers are pre-allocated as local buffers and set to the *s_axilite* interface in the HLS wrapper, so both buffers appear to be memory-mapped slaves on the TileLink bus. The accelerator can issue a send command by telling the NIC the base address and the length of the data it intends to send and a destination MAC address. If the accelerator is expecting incoming data, it issues a *post_recv* command to the NIC with the address of the receive buffer. Currently, we require that the general-purpose processor not simultaneously access the network interface while the accelerator is using the NIC.

Dedicated Send/Receive Queues. Accelerators can also directly communicate with the NIC and thus the external Ethernet network through dedicated send and receive queue pairs. To implement this functionality, we extend the NIC design with routing/tagging based on the Ethernet Ethertype field. The NIC automatically directs network packets to the accelerator's queues if the Ethertype value is "`ACCEL_ONLY`".

In this mode, rather than issuing commands and polling the NIC, the accelerator can directly send and receive the Ethernet packets through decoupled FIFO queues. We split each queue into two sub-queues, one for passing the Ethernet header and one for passing the payload. We can then treat the payload queue as a data stream in a dataflow programming model with blocking read and non-blocking write. This allows us to take a design written in the Vivado HLS dataflow model and split it into multiple network-attached accelerators. Currently, the flow requires that you have sufficient buffering and that the send and receive rate are matched. In future work, we plan to add a flow-control mechanism to designs.

2.4.3 Generating Accelerators with Vivado HLS

We generate accelerator RTL by taking advantage of Vivado High-Level Synthesis (HLS) rather than requiring users to manually write RTL for accelerator designs. While HLS tools have previously been relegated to FPGA-based deployment [211], ASIC CAD tool vendors have begun to ship HLS tools geared towards ASIC designers. The process of converting a high-level (C) description of an application to a hardware accelerator in Centrifuge is detailed in Figure 2.2.

At a high-level, the flow to integrate HLS-generated accelerators into the SoC is as follows:

1. The programmer develops a standard C program and identifies a function to accelerate (HLS-compatible).

2. Our LLVM pass replaces all calls to the accelerated function with calls to a new wrapper function. This wrapper function calls the accelerator.

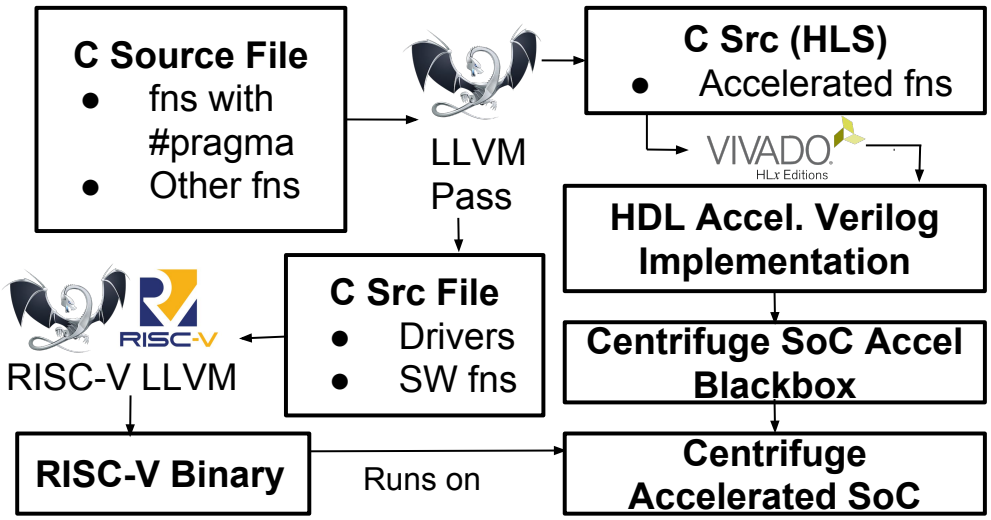


Figure 2.2: Centrifuge HLS Flow, C to Acclerator RTL + Software Driver/Application

3. LLVM writes a RISC-V assembly file which is assembled and linked by the standard RISC-V GCC toolchain.

4. The function name to be accelerated is placed into a tcl script that is used to drive HLS.

5. Vivado HLS produces a Verilog implementation of the accelerated function.

6. Our custom FAME-1 transformation is applied to the generated Verilog.

7. A pair of controllers are attached to the accelerator and act as bridges between the accelerator and the RoCC/TileLink2/Network interfaces. They handle the command/response messages between the processor and the accelerator and memory request/response messages.

8. The accelerator is added to the SoC and the design is elaborated by Chisel.

2.4.4 Generating the software stack for a complete SoC

To complete the generation of our SoC system, we need to produce software shims that provide access to the generated accelerators from various levels of the software stack. In the previous section, we discussed how our LLVM pass will generate workload binaries with calls to the accelerator, either as bare-metal programs or programs that expect to run on Linux. Below, we outline the system-level software shims to provide access to accelerators.

2.4.4.1 Running Bare-metal

In a bare-metal environment, the interaction between software and accelerators is straightforward. In order to invoke a RoCC accelerator, the custom instruction assigned to the target accelerator needs to be called with arguments stored in processor registers. For TileLink

accelerators, we need to perform store operations to the memory-mapped registers to pass function arguments and control commands. In both cases, similar to a software function call, we pass in scalar arguments directly and pointer arguments as physical memory addresses. The accelerator can directly access memory through the caches (L1 for RoCC, L2 for Tilelink). All requests are serviced by the memory controller without directly involving the processor.

2.4.4.2 Running on Linux

In Linux, RoCC-based accelerators are invoked using custom instructions and can use the processor TLB to perform translations; they do not require special operating system drivers. Tilelink accelerators, however, become more complex with an operating system due to virtual memory handling. Specifically, the drivers and software wrappers for an accelerator/application must support the following three features:

Accessing MMIO from user space. TileLink accelerators are controlled through memory-mapped physical addresses. In order for the application to access these addresses, they must first be mapped into the application's virtual address space. On Linux, physical memory can be accessed through the "/dev/mem/" special file. By calling `mmap` with the offset set to the desired physical address, we can map any physical address into our virtual memory. This procedure is handled automatically in our generated wrapper code.

Translating from virtual to physical addresses. To handle address translation, the software wrappers for TileLink accelerators call an automatically-generated RoCC accelerator that interacts with the hardware page-table walker to translate from virtual to physical addresses.

Ensuring physically contiguous data-layout. For pointer arguments that fit within a single page, translation alone is sufficient. However, if the argument spans multiple pages, the allocated memory may not be physically contiguous. In lieu of maintaining a TLB in each accelerator, Centrifuge requires that all arguments be made physically contiguous before invoking an accelerator. To allow this, we provide a Linux driver that allocates a large, physically-contiguous, region of memory at boot time and exposes it to users through a modified mmap system call. Users can allocate space for their arguments using this system call (minimizing overheads). If the user would prefer to not modify their source, the generated function wrappers can copy arguments into contiguous memory automatically when the accelerator is invoked.

2.5 Centrifuge Case Studies

In this section, we evaluate several microbenchmarks and perform three case studies to demonstrate the capability of our methodology. We ran our experiments on FireSim on Amazon F1 instances and used Vivado HLS to synthesize application C code into hardware as it is stable and free to the community. Our microbenchmarks are adapted from CHStone [77]

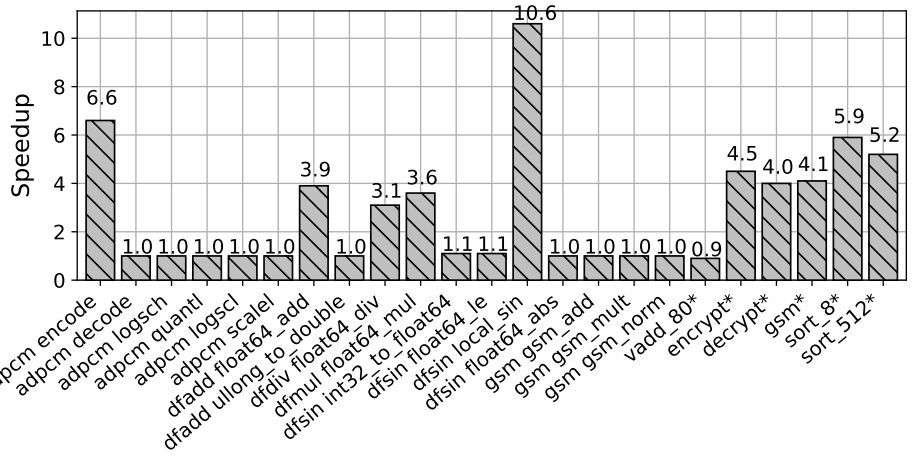


Figure 2.3: RoCC Accelerators SpeedUp Compared to Software(* indicates accelerator with pointer type inputs)

and HLSpolito on GitHub [167]. With the microbenchmarks, we demonstrate that Centrifuge can be used to evaluate the following design tradeoffs:

Acceleration Region. We first conducted a sweep to extract functions from the microbenchmarks and compiled them to the RoCC accelerators with Centrifuge. Results in Figure 2.3 showing the accelerator speedup can be used to direct decision on which code region to accelerate. For example, the *adpcm* example should be accelerated at the *encode* level instead of at the basic operation level (e.g. *logsch*, *quantl*, *logscl*) .

Software Stack. We then generated five TileLink accelerators and ran them under Linux. Figure 2.4 shows the runtime breakdown of the accelerators normalized to the software performance. The slowdown of Tilelink accelerators on Linux is primarily due to performing address translation on each argument. Note that RoCC accelerators do not experience any slowdown on Linux because they are virtually addressed. With Centrifuge, we can evaluate and optimize the physically-addressed TileLink accelerator and its Linux driver together.

Accelerator Coupling. Lastly, we show how different coupling affects the accelerator performance with Centrifuge by accelerating a communication-bound kernel *vadd* in different sizes. From Figure 2.5, we see that the RoCC accelerator outperforms software when the vector size is small. As we increase the vector size, the TileLink accelerator gets a higher speedup compared to the software. Three main factors affect the accelerator speedup: the interface bandwidth, the cache hit latency, and the cache size. The TileLink system bus is 512-bit wide, whereas the RoCC memory interface is 64-bit wide. The L2 hit latency is $20\times$ longer than L1 hit latency. For the RoCC accelerator, once it starts to miss in L1 cache, it will suffer from a similar cache access latency as the TileLink accelerator. However, since TileLink accelerator has wider memory accesses, it performs better in a more bandwidth-bound scenario.

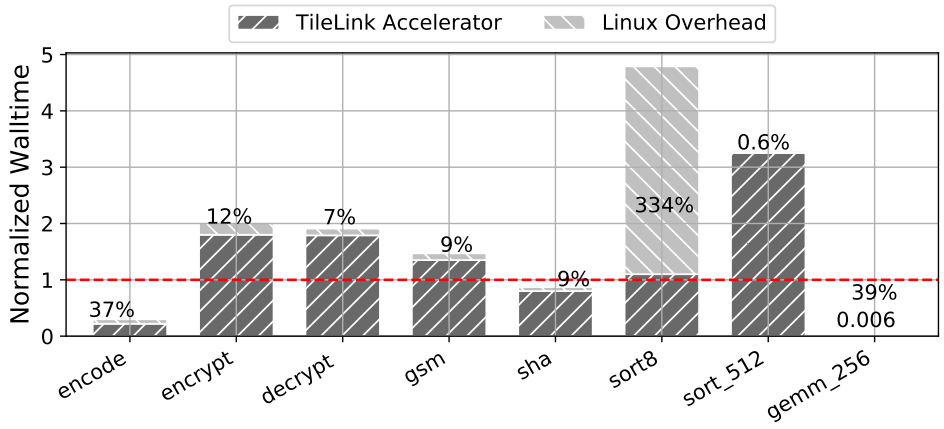


Figure 2.4: Tilelink Accelerators with Linux Driver

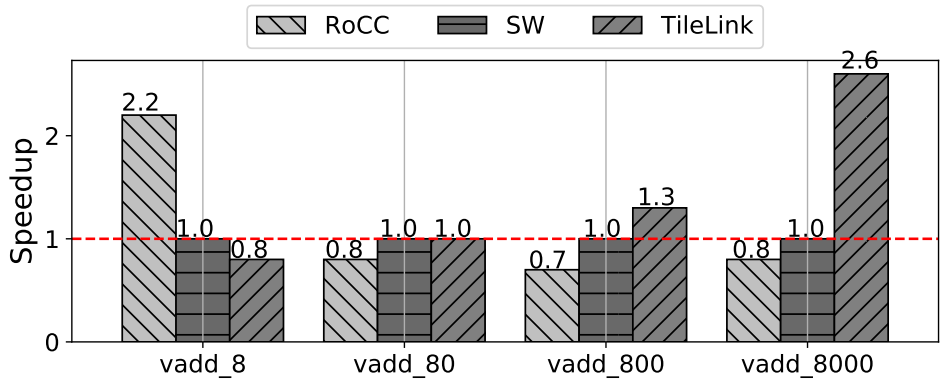


Figure 2.5: Different Coupling for *vadd* Accelerator

2.5.1 Smart-House Hub

In this case study, we demonstrate Centrifuge using a hypothetical SoC intended for a smart-house assistant (e.g., Alexa, Google Home, etc). Our device will need to listen for user audio commands, encode them into an appropriate format, perform machine-learning inference to detect commands, and finally encrypt the command for transfer to the cloud over a wireless network.

2.5.1.1 Evaluating the Baseline Application

We begin by measuring runtimes for each of these kernels without accelerators. Figure 2.6 shows how the runtimes of these steps might compare in a typical deployment ("Software Only"). Notice that the lion's share of time is spent in audio preprocessing (*adpcm_ encode*) and the matrix-multiply underlying command classification (*gemm_ 256*). The remaining time

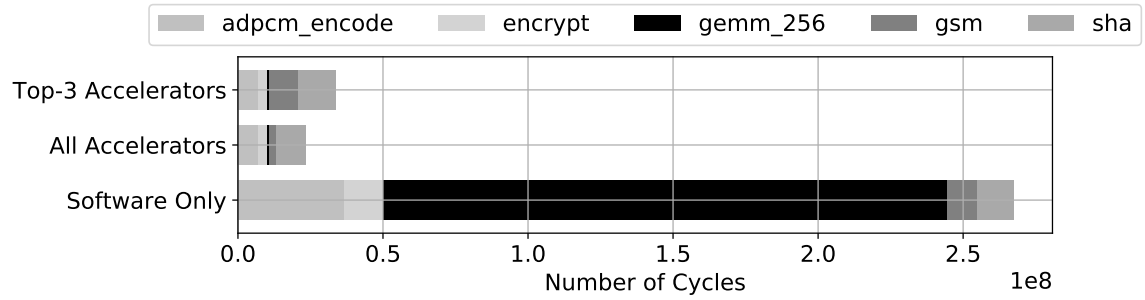


Figure 2.6: Breakdown of key computational kernels in a hypothetical smart-house assistant SoC. The top-3 accelerators for end-to-end performance are *adpcm_ encode*, *gemm_ 256*, and *encrypt*

.

is split roughly evenly between hashing (*sha*), encryption (*encrypt*), and wireless encoding (*gsm*).

2.5.1.2 Generating Accelerators

Having identified the key kernels in our application, we begin by adding HLS annotations to each function. This mostly involves identifying inputs and outputs, and ensuring the function prototype has the correct number of arguments. By modifying the source code and annotations, the design can be further specialized for hardware deployment if appropriate. We then run Centrifuge on our annotated application, specifying each kernel. The result is RTL for a new SoC with the specified accelerators and a new application binary with each accelerated function replaced with a call to an accelerator.

2.5.1.3 Evaluating Accelerators

The next step is to evaluate our accelerators in an end-to-end system using FireSim. With FireSim, we can run our code as if it were on a real machine, including any timing measurements. Figure 2.6 shows the runtime breakdown using our new accelerators ("All Accelerators"). We notice that *gemm_ 256* shows the greatest improvement, both local (250×) and end-to-end (11.5×), and should likely be included. The adpcm encoder shows a more modest local improvement (about 5×) but has the second largest impact on total runtime (6% end-to-end improvement). Both encryption and gsm-encoding show 4-5x improvements in local runtime, but have a modest 1% impact on end-to-end performance; we may choose to include these if power and area permit. Finally, the sha accelerator sees little improvement locally, and has a very small impact on end-to-end runtime; we would likely choose not to accelerate that function.

Algorithm 1 DGEMM Algorithm: PREFETCH(X, M) will begin loading M into X for the next iteration. OWN(X) determines if this node owns X.

1: $k'(k) \leftarrow (k + j) \bmod K_{\text{blocks}}$
2: **for all** i in M_{block} **do**
3: **for all** j in N_{block} **do**
4: **if** OWN($C_{i,j}$) **then**
5: PREFETCH($l_A, A_{i,k'(0)}$)
6: PREFETCH($l_B, B_{k'(0),j}$)
7: **for all** k in K_{block} **do**
8: PREFETCH($l_A, A_{i,k'(k+1)}$)
9: PREFETCH($l_B, B_{k'(k+1),j}$)
10: $C_{i,j}$ += $l_A \times l_B$
11: **end for**
12: **end if**
13: **end for**
14: **end for**

2.5.1.4 Continue Hardware and Software Development

Putting it all together, we decide to include the *gemm_256*, *adpcm_encode*, and *encrypt* accelerators and leave the remaining kernels to the CPU. This results in an 8x improvement in end-to-end runtime (including all the accelerators would result in an 11.5% improvement). In addition, we now have a consistent hardware/software interface and a high-performance simulator for use by our software team, while hardware engineers can continue to optimize the kernels, using either HLS or hand-written RTL.

2.5.2 Distributed Matrix Multiplication Accelerator

We applied Centrifuge to a MPI-based distributed matrix multiplication implementation [24]. This algorithm employs MPI's one-sided communication protocol, along with extensive tiling and prefetching. We used two separate implementations for the core matrix-multiplication algorithm; an HLS-optimized kernel (adapted from [210]), and a CPU-optimized tiling algorithm. The accelerator is integrated as a TileLink accelerator. It runs 441 × faster than the CPU for performing 8-bit integer matrix multiplication.

In this workload, the MPI processes can read and write directly into each others memory using one-sided `MPI_Get` and `MPI_Put` operations. We divide up our distributed matrices into tiles, with one tile per CPU/accelerator pair. Algorithm 1 describes our algorithm, which is typical of one-sided matrix multiplication, each process is responsible for computing the output block of the resultant C matrix that it owns. In order to compute this output block, we iterate through the corresponding row of A tiles and column of B tiles, pull them into memory, multiply the matrices together, accumulating the result into the local tiles of the C

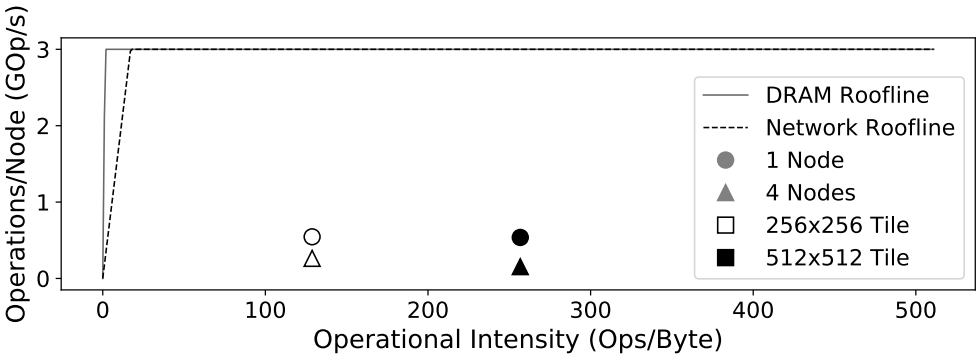


Figure 2.7: CPU Roofline Model

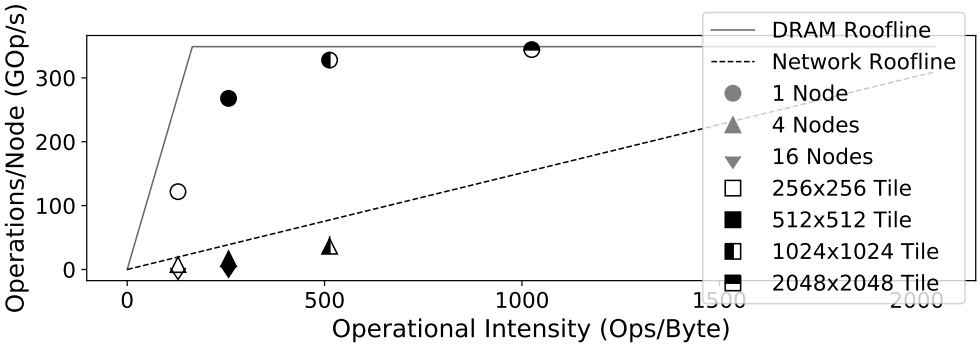


Figure 2.8: Accelerator Roofline Model

matrix. We use one level of prefetching to allow overlap of communication and computation, and we also offset the order of iteration in order to provide load balance. Tiles are stored in an MPI window so that they can be accessed using MPI's one-sided primitives. The MPI window is created inside a pinned memory region that is visible to the accelerator so that the accelerator can directly read local tiles stored in distributed memory without copying them. When we retrieve local copies of matrix blocks l_A and l_B, we also store them inside pinned accelerator-accessible memory that we allocate using a C++ allocator.

We first validated the HLS design by running the C simulation and comparing the output against a golden reference. This takes less than a minute for each debugging iteration. We then used Centrifuge to generate the SoC and simulated the accelerator on FireSim. It took ∼3 hours to generate the FPGA images and seconds to return the test results for multiplying matrices of size 256×256. In comparison, it would take around half a day to run the bare-metal program on a commercial software-based RTL simulator. Using a software-based RTL simulator, it would be infeasible to validate the design in a more realistic setting, for example, using a larger input size, running under Linux, and in a network environment.

We can further evaluate the complex interaction between the accelerator, CPU, and the

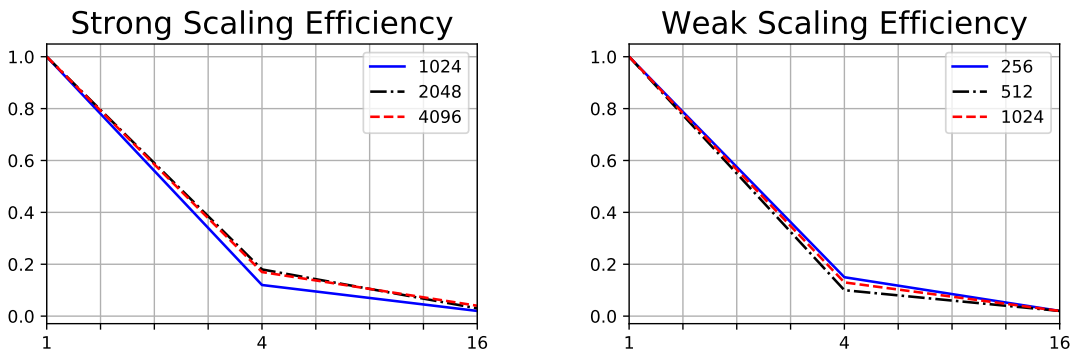


Figure 2.9: Scaling Efficiency

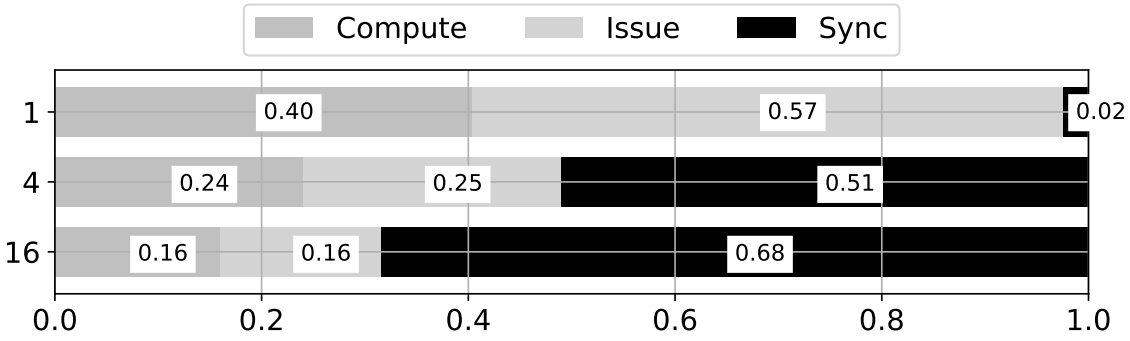


Figure 2.10: DGEMM Runtime Breakdown for 1024×1024 Tiles

network by employing Centrifuge. To evaluate the design, we first calculate peak performance using a roofline model [204]. Because we are running a full-system cycle-level simulator, we were able to use standard evaluation tools like STREAM [131] and iperf [65] to find the actual bounds for the roofline model. The distributed dgemm framework described above was taken from an existing high-performance library and run unmodified (except for calls to the accelerator and special memory allocation of arguments as described in section 2.4.4.1). We ran the experiments on 1, 4, and 16-node configurations with 2.0 GB/s measured DRAM bandwidth and 1.2 Gbit/s measured network bandwidth.

Figure 2.7 shows that the performance of the workload on CPU is dominated by a lower bound than its peak compute bound. As shown in Figure 2.8, by running the accelerator, the workload becomes communication bound, and our accelerator performance matches the measured roofline. The accelerator achieves a peak throughput of 344 GOp/s for 16-bit integer multiply-accumulates (a $\sim 600\times$ improvement compared to our processor). On one node, the accelerator's performance follows the DRAM roofline (memory communication bound), while on four or sixteen nodes, its performance tracks the network roofline instead (network communication bound). Therefore, for the distributed workload in this example,

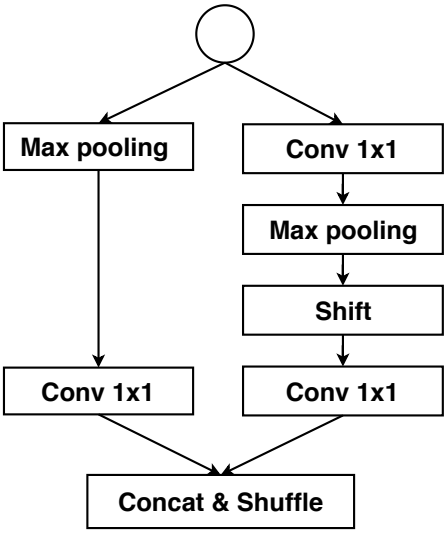


Figure 2.11: DiracDeltaNet

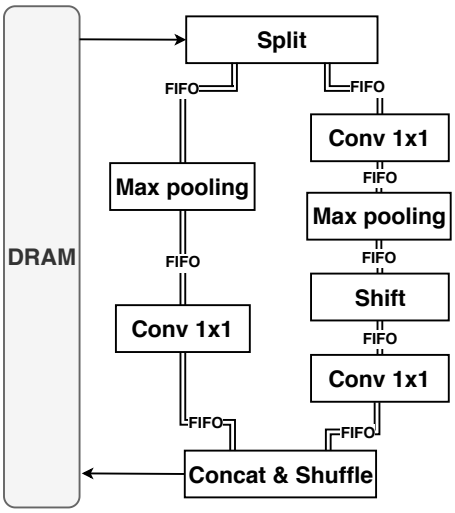


Figure 2.12: Hardware Design

major improvements should be made to the network bandwidth instead of the accelerator itself. Figure 2.9 shows the strong and weak scaling efficiency of the distributed workload running on the accelerators. A detailed runtime breakdown for the distributed workload with tile size 1024×1024 is shown Figure 2.10.

2.5.3 Deep Learning Accelerators

With fast design feedback, our flow is particularly suited for developing accelerators for rapidly changing deep learning algorithms. In this section, we will describe several deep learning accelerators developed with Centrifuge. Note that all the accelerators in this section took less than one month to implement.

2.5.3.1 Design for New Algorithms

Figure 2.11 shows the basic building block for a new efficient network design called DiracDeltaNet [219].

In this design, all 3x3 convolutions are replaced with a 1x1 convolution and shift operation, while the addition-skip connection is replaced with concatenation and shuffle operations. Figure 2.12 shows our hardware dataflow design for the building block. In the design, all layers are spatially mapped to corresponding hardware units. There are three 8×8 Multiply-Accumulate(MAC) units to support the three 1x1 convolution layers. The weights are pre-fetched into the on-chip buffers. The input activations are loaded to the FIFOs from DRAM. Each hardware unit starts its execution based on the arrival of data. Since we preload all the weights, each input activation can be reused output_channel_size times after it is fetched from DRAM. Table 2.1 shows the Ops/cycle for different DiracDeltaNet subgraphs on our TileLink accelerator. As the compute-to-communication ratio varies among

Table 2.1: Accelerator Performance
(The workload size is represented as image_width × channel_depth)

Workload Size	Total Ops	Ops/cycle
32×16	196608	4.55
32×32	786432	15.12
32×64	3145728	20.59
16×128	3145728	21.35
8×64	196608	17.09

different subgraphs, the empirical performance of the accelerator varies drastically (∼4×). The algorithm designer can then leverage the information from current hardware architecture to optimize the DNN design to include more hardware-efficient layers.

2.5.3.2 Distributed Accelerators

The dataflow architecture with large weight buffers mentioned in the previous example is also known to have low latency for inference with a small batch size. However, it might not be economical to have such a large design with all the layers hard wired together.

Instead, we can have many composable deep neural network accelerators with different dataflow modules, and have them directly communicate with each other through a high-performance network as shown in Figure 2.13. We implement this design based on VGG16 [183]. We first tested the idea with a small 2-layer neural network with two 16×16 Conv3×3. By replacing the data stream with the Ethernet connection, we reduced the total latency by 1.5%. This indicates that the overhead from the direct network connection is tolerable. We then prototyped two accelerators with our framework: one with only the convolution clusters, and one with both convolution and fully connected layers for reducing the results. Both designs can directly send and receive Ethernet packets to the network through the NIC. The weights are 2-bits, and the activations are 4-bits in the hardware. Assuming the chip is running at 3.2 GHz, for a 64 × 64 large images, it takes 13191136 cycles (4.1ms) to classify one frame on a single node accelerator, and 11151953 cycles (3.5ms) to finish the same task on a two-node system that has direct accelerator-to-accelerator network connections. While both designs have the same number of compute units, the two-node design benefits from increased aggregate memory bandwidth. In this case, the benefits of increased memory bandwidth outweigh any overheads from the network.

2.5.4 Graph Accelerator

For data-dependent workloads or applications that have intricate synchronization patterns with other system components, it is challenging to derive an analytical model for deciding

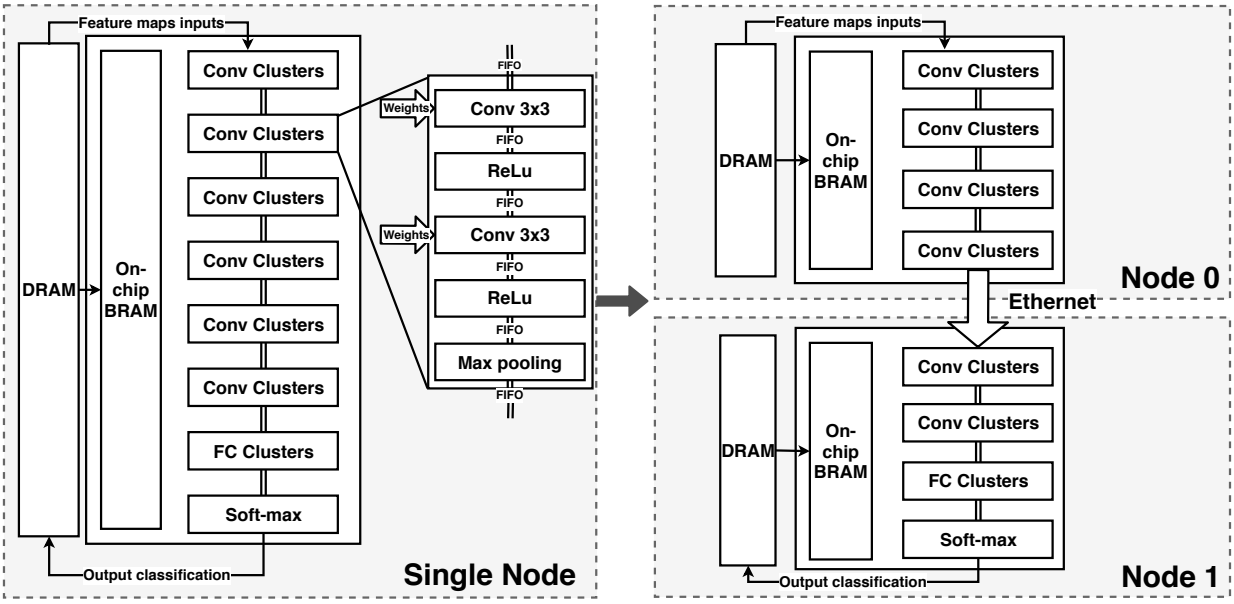


Figure 2.13: Multi-node accelerators, connected via Ethernet

whether accelerators are needed. One advantage of our framework is that the user can quickly generate the accelerator and evaluate it with the rest of the system.

This section demonstrates this capability by adding a graph accelerator to run the connected components (CC) algorithm concurrently with the CPU. Connected Components [83] is a widely used graph algorithm that detects all connected regions in a graph.

As there are two compute nodes (CPU and accelerator) in the heterogeneous system, the two-node algorithm listed in Algorithm 2 is used.

In the algorithm, all edges should be directed. The vertices are divided into two parts, V_0 and V_1, and each compute node is responsible for updating one part of them. The update steps are as follows:

1. All vertices in V_0 and V_1 are initialized to a unique label.
2. Each compute node scans the edges it owns. For each edge, if the label of the destination vertex is smaller than the label of the source vertex, it should be updated to the label of the source vertex. If this destination vertex belongs to the remote node, the vertex and its corresponding label should be stored into a send buffer.
3. The send buffer will be shared to the remote node once it is full.
4. After sharing the data with the remote node, each node will be blocked until they receive data from the remote node.
5. Both nodes start to process the receive buffer and repeat steps 1 to 5 until it converges.

Algorithm 2 Connected Components

Require: $G = (V, E)$
Ensure: label for each vertex, $label(v_i)$
1: **for** each $v_i \in V$ **do**
2: $label(v_i) = i$
3: **end for**
4: **while** not finished **do**
5: **for** each $e \in E$ **do**
6: // e is from v_i to v_j
7: **if** $(label(v_j) > label(v_i))$ **then**
8: $label(v_j) = label(v_i)$
9: **end if**
10: **end for**
11: **end while**

The local node has a receive buffer to buffer the incoming updates. Before updating, the label of each vertex is set to a unique value (e.g., index), representing each vertex as an individual component. Then, updating is executed in iterations. During an iteration, the value of V_1 is transferred to the node. Then, edges in E_0 are scanned one by one. If the label of the destination vertex is smaller than that of the source vertex, it will be updated to the label of the source vertex. We allocate a local buffer for remote vertices V_1. When a remote vertex is updated, it will be stored in the local buffer. When the buffer is full, the updating function is blocked until all updated vertices are transferred to another computation node. There is also another local buffer for receiving an updated value of V_0 from other computation nodes. We can easily extend this design to multiple computation nodes by allocating more buffers for sending and receiving updated vertices.

In the hardware design, we store all edges in the off-chip DRAM. The graph accelerator is a Tilelink accelerator that shares the L2 cache with the processor. The accelerator has three large local buffers to store the labels of vertices it owns, the send data, and the receive data. The communication between the accelerator and the processor is through shared memory. Based on the design above, we ran the igo-Facebook graph from the Stanford Network Analysis Project (SNAP) [117].

In our evaluation, the workload runs 5× slower on the CPU-accelerator implementation compared to the CPU-only implementation. We found that the overhead of synchronizing with the accelerator outweighs the benefits of having more compute resources. This demonstrates the case in which adding more loosely coupled accelerator nodes can be a bad idea, even on a system with only one single-issue in-order processor.

2.6 Parallel Abstraction for HLS

In Centrifuge, the sequential nature of the high-level C/C++ language has posed a challenge in adequately describing the concurrency of the workloads and supporting various forms of parallelism in hardware. In existing HLS frameworks like Vivado HLS [211], Altera OpenCL [94], and LegUp [27], the support of sequential C is maturing but the high-level abstraction to support concurrent models of computation is still limited. Therefore, in this exploratory work, we performed a study to investigate the feasibility of supporting arbitrary forms of concurrency in HLS. We map concurrency described in Communicating Sequential Processes (CSP) to hardware through HLS. We select a subset of features from the CSP-inspired language Go [52] to our high-level HLS abstraction.

2.6.1 Communicating Sequential Processes

CSP is an established formalism for concurrency in computer programs. Since its introduction by Hoare in 1977 [84], it has been an active area of research. CSP is similar in many respects to the Actor model. Traditionally, concurrently executing programs must negotiate access to shared memory in the presence of arbitrary interleaving, usually requiring careful management of locks around critical sections. Under CSP and the Actor model, processes within programs execute as if independently, concerned only with their local state. In CSP these processes are anonymous, while under the Actor model, they are themselves named actors. To communicate with each other or share information, CSP uses synchronized unbuffered channels. Under the Actor model, the channel is implicit (Actors send messages to each other by name) and unbuffered. Under CSP, writes to a channel cannot complete until another process is ready to read the value, whereas Actors do not need to synchronous message writes to others read.

Modern programming languages like Go and Rust [106] embed ideas from CSP and Actor systems as higher-level interfaces to concurrency. Using channels as a first-class concurrency primitive avoids the difficulty of (and the human errors introduced by) managing lower-level primitives like mutexes, conditions variables, and so on. These channels can be buffered or unbuffered, with multiple producers and consumers. Crucially, this general idea seems to map neatly to hardware logic implementation. Independent actors/processes can be implemented as separate hardware modules, and the channels interconnecting them as fixed FIFO channels. Since the abstraction has forced the user to restrict the processes' states and boundaries more cleanly than lower-level concurrency models, asking programmers to write in a CSP model can make it easier to synthesize hardware for concurrent processes.

2.6.2 Go-to-Verilog HLS

Leveraging existing HLS tools, an LLVM compiler for Go (llgo), and an LLVM C-backend (llvm-cbe), we implement a Go-to-Verilog HLS compiler. Given a program written in Go, our tool first compiles it with llgo and emits LLVM IR. The C-Backend of LLVM then takes the generated LLVM IR and decompiles it to C code. The C code is later fed into the HLS

tools for Verilog generation. As a GC language, Go has a runtime system, whereas in C, everything is statically scheduled during compile time. When a Go program is compiled into LLVM IR, particular operations are transformed into function calls to interface with the Go runtime system. The difference in the language features of Go and C imposes the first set of requirements our tool needs to handle:

1. Replace the function calls to the runtime system with C functions that perform the same task.
2. Change the Go channel interface to equivalent constructs in C.
3. Remove the initialization of the runtime system from the generated C code.

In addition, current HLS tools only support a subset of features in the C/C++ language. This limitation imposes a second set of requirements on the form of C code we generate as input to HLS tools. The common constructs that are not supported by HLS in C include:

1. Dynamic allocation of variables/arrays
2. Dereferencing pointers that are not resolvable during compile time
3. Recursion
4. Indirect function calls

Current HLS tools do not have mature support for any features that would involve the use of dynamic memory. This is mainly due to the lack of a central, shared stack or heap memory system in the statically scheduled HLS accelerators for holding intermediate values. With the addition of a similar memory system, those features can be potentially supported. In this project, we confine our goal to the source-to-source transformation from Go to the HLS-compatible C code. There are two commercial HLS tools that we target: LegUp HLS [27] and Vivado HLS [211]. LegUp HLS provides APIs for a multi-sender and multi-receiver FIFOs abstraction. We can map the Go channel interface to the FIFO APIs of LegUp in a straightforward manner. A major difference is that the Go Runtime takes a pointer to the variable of random size as the input, whereas LegUp only takes a 64-bit value as the input to their send interface. For the intrinsic variables smaller than 64-bit in Go, we can dereference its pointer and send its value over LegUp FIFOs. For a struct variable in a larger size, a wrapper is created to serialize it and instantiate several FIFO write transactions. On the receiver end, another wrapper needs to be created to receive the data until a done token is seen. Similar to LegUp, Vivado HLS also supports the streaming interface by using the hls::stream data type. However, the Vivado HLS dataflow cannot support multiple producers or consumers. We expand this capability in our tool flow by installing arbiters to arbitrate concurrent requests during system integration. Lastly, we verify that our flow produces correct RTL and works on an FPGA by system integration. Our target device is Zedboard System-on-Chip. This platform has a ZC7020 FPGA device with an ARM A9 CPU.

2.6.3 Discussion

This study demonstrated a working end-to-end toolchain for synthesizing a subset of Go to hardware in LegUp and Vivado HLS environments. In particular, we show that CSP-style algorithms built with Go's channels and goroutines can be straightforwardly mapped to hardware. First, we modified our Go compiler to generate LLVM IR to use types and functions available in the LegUp environment. Second, we changed the LLVM C backend to generate satisfactory C from that IR for the HLS toolchain. We enabled the feature to synthesize generated program to a Vivado FPGA/SoC development board. We verified the toolchain by synthesizing a few examples; our preliminary benchmarks confirm that there is a significant performance advantage when the independent processes perform sufficiently complex operations.

The most difficult part of this project was understanding the requirements at each of the component boundaries. What subset of valid C does the HLS suite support? How different are Vivado's expectations? What form of the LLVM IR does the C backend understand? What are the assumptions behind the design decisions made in the Go compiler, and how far can we stretch them to mutate types and the IR?

Although our support for Go features is incomplete, it is sufficiently cohesive to demonstrate how programs written with Go's CSP-style concurrency model naturally fit hardware acceleration. The semantics of our Go programs from the view of the programmer are carried through to the hardware unchanged; the intermediate forms simply have to get out of the way.

2.7 Conclusion

In this chapter, we described a methodology and flow, *Centrifuge*, that can rapidly generate and evaluate heterogeneous SoCs by combining an HLS toolchain with the open-source FireSim FPGA-accelerated simulation platform. Our system can quickly produce complete SoC systems with many integrated HLS-generated accelerators as specified by the user, simulate them quickly and cycle-accurately on FPGAs, and run complete software stacks on top, including booting Linux and running full application frameworks. Our system allows users to easily explore a variety of accelerator integration techniques, by automatically integrating accelerators in several ways—as tightly coupled RoCC accelerators, as accelerators that communicate over the standard on-chip network, and lastly as "disaggregated" accelerators that are directly attached to an Ethernet network between SoCs. We extended the FireSim simulation platform with a new FAME-1 transformation that operates on the Verilog designs emitted by Vivado HLS rather than Chisel RTL. By integrating these tools, our methodology allows users to rapidly generate an entire hardware/software stack for a customized SoC that can be fabricated as an ASIC and evaluate its end-to-end performance using cycle-exact FPGA simulation, allowing for agile design-space exploration of novel accelerator-based systems.

Chapter 3

Algorithm and Hardware Co-Design

Deep convolutional neural networks (CNNs) power state-of-the-art solutions on a wide range of computer vision tasks. However, deploying them on edge devices where computational resources are limited has been challenging due to their high computational demand. Using FPGAs to accelerate CNNs has attracted significant research attention in recent years. FPGAs excel at low-precision computation, and their adaptability to new algorithms lends themselves to supporting rapidly changing CNN models. This chapter summarizes two of our works on co-designing algorithms and hardware to achieve high-accuracy real-time low batch size inference for two different commonly used computer vision tasks on embedded FPGAs.

Despite recent efforts to use FPGAs to accelerate CNNs, as [113] points out, there still exists a wide gap between accelerator architecture design and CNN model design. The computer vision community has been primarily focusing on improving the accuracy of CNNs on target benchmarks with only secondary attention to the computational cost of CNNs. As a consequence, recent CNNs have been trending toward more layers [79], more complex structures [87, 242], and more complicated operations [220].

On the other hand, FPGA accelerator design has not leveraged the latest progress of CNNs. Many FPGA designs still focus on networks trained on CIFAR10 [109], a small dataset consisting of 32x32 thumbnail images. This dataset is usually used for experimental purposes and is too small to have practical value. More recent designs aim to accelerate inefficient CNNs such as AlexNet [110] or VGG16 [183], both of which have fallen out of use in state-of-the-art computer vision applications. In addition, we observe that in many previous designs, key application characteristics such as frames-per-second (FPS) are ignored in favor of simply counting GOPs, and accuracy, which is critical to applications, is often not even reported.

Specifically, we see gaps between CNN architectures and accelerator design in the following areas:

Inefficient CNN models: Many FPGA accelerators still target older, inefficient models such as AlexNet and VGG16, which require orders-of-magnitude greater storage and computational resources than newer, efficient models that achieve the same accuracy. With an inefficient model, an accelerator with high throughput in terms of GOPs can actually have

low inference speed in terms of FPS, where FPS is the more essential metric of efficiency. To achieve AlexNet-level accuracy, SqueezeNet [93] is 50x smaller than AlexNet; SqueezeNext [62] is 112x smaller; ShiftNet-C [206], with 1.6% higher accuracy, is 77x smaller. However, not many designs target those efficient models. Additionally, techniques for accelerating older models may not generalize to newer CNNs.

CNN structures: Most CNNs are structured solely for better accuracy. Some CNNs are structured for optimal GPU efficiency, but few, if any, are designed for optimal FPGA efficiency. For example, the commonly used additive skip connection [78] alleviates the difficulty of training deep CNNs and significantly boosts accuracy. Despite its mathematical simplicity, the additive skip connection is difficult to implement on FPGAs efficiently. Additive skip connections involve adding the output data from a previous layer to the current layer, which requires either using on-chip memory to buffer the previous layer's output or fetching the output from off-chip memory. Both options are inefficient on FPGAs.

CNN operators: CNN models contain many different types of operators. Commonly used operators include 1×1, 3×3, 5×5 convolutions, 3×3 max-pooling, etc. More recent models also contain the depth-wise, group, dilated, and factorized convolutions. Not all of these operators can be efficiently implemented on FPGAs. If a CNN contains many different types of operators, one must either allocate more dedicated compute units or make the compute unit more general. Either solution can potentially lead to high resource requirements, limited parallelism, and more complicated control flow. Also, hardware development will require more engineering effort.

Quantization: CNN quantization has been widely used to convert weights and activations from floating-point to low-precision numbers to reduce the computational cost. However, many of the previous methods are not practically useful for FPGAs due to the following problems: 1) Quantization can lead to serious accuracy loss, especially if the network is quantized to low precision numbers (less than 4 bits). Accuracy is vital for many computer vision applications. Unfortunately, carefully reporting accuracy has not been the norm in the FPGA community. 2) Many of the previously presented quantization methods are only effective on large CNN models such as VGG16, AlexNet, ResNet, etc. Since those models are known to be redundant, quantizing those to low-precision is much easier. We are not aware of any previous work tested on efficient models such as MobileNet or ShuffleNet. 3) Many methods do not quantize weights and activations directly to fixed-point numbers. Usually, quantized weights and activations are represented by fixed-point numbers multiplied by some shared floating-point coefficients. Such representation requires more complicated computation than purely fixed-point operations and is more expensive.

To bridge these gaps, we investigated various opportunities to co-design algorithms and hardware targeting two major computer vision tasks: image classification (*Synetgy*) and objective detection (*CoDeNet*).

3.1 Co-design for Image Classification

Image classification is the process of assigning labels to images. It is one of the most fundamental tasks in computer vision and is widely adopted in various applications, such as autonomous driving, medical imaging, robotics, etc. Supervised learning is an effective way to approach image classification tasks. It trains the classification algorithm with sets of images and their corresponding labels. Deep convolutional neural networks (CNN) [78, 110, 184, 192] are the most popular supervised learning algorithms for image classification since their groundbreaking advancement in 2012. However, the success comes at the cost of the ever-increasing computation requirements. In this work, we adopt an algorithm-hardware co-design approach to develop a CNN accelerator called Synetgy and a novel CNN model called DiracDeltaNet. Both the accelerator and the CNN are tailored to FPGAs and are optimized for ImageNet classification accuracy and inference speed (in terms of FPS).

Our co-design approach produces a novel CNN architecture DiracDeltaNet that is based on ShuffleNetV2 [128], one of the state-of-the-art efficient models with small model size, low FLOP counts, hardware friendly skip connections, and competitive accuracy. We optimize the network by replacing all 3×3 convolutions with shift operations [206] and 1×1 convolution, enabling us to implement a compute unit customized for 1×1 convolutions for better efficiency. The name "DiracDeltaNet" comes from the fact that the network only convolves input feature maps with 1×1 kernels. Such kernel functions can be seen as discrete 2D Dirac Delta functions. We further quantize the network to 4-bit weights and 4-bit activations, exploiting the strengths of FPGAs, with only a less than 1% accuracy drop. In short, DiracDeltaNet's small model size, low operation count, low precision and simplified operators allow us to co-design a highly customized and efficient FPGA accelerator. Furthermore, the implementation only took two people working for one month using High-Level Synthesis (HLS).

We trained DiracDeltaNet on ImageNet, implemented it on our accelerator architecture, Synetgy, and deployed it on a low-cost FPGA board (Ultra96). Our inference speed reaches 66.3 FPS, surpassing previous works with similar accuracy by at least 11.6x. The DiracDeltaNet on our accelerator architecture also achieves 88.1% top-5 classification accuracy—the highest among all the previously reported embedded FPGA accelerators.

3.2 Synetgy Background and Motivation

3.2.1 Efficient CNN Models

For the task of image classification, improving accuracy on the ImageNet [47] dataset has been the primary focus of the computer vision community. For accuracy-sensitive applications, even a 1% improvement in accuracy on ImageNet is worth doubling or tripling model complexity. As a concrete example, ResNet152 [78] achieves 1.36% higher ImageNet accuracy than ResNet50 at the cost of 3x more layers. In recent years, efficient CNN models have begun to receive more research attention. SqueezeNet [93] is one of the early models focusing on

reducing the parameter size. While SqueezeNet is designed for image classification, later models, including SqueezeDet [205] and SqueezeSeg [207, 208], extend the scope to object detection and point-cloud segmentation. More recent models such as MobileNet [86, 171] and ShuffleNet [128, 228] further reduce model complexity. However, without a target computing platform in mind, most models designed for "efficiency" can only target intermediate proxies to efficiency, such as parameter size or FLOP count, instead of focusing on more salient efficiency metrics, such as speed and energy. Recent works also try to bring in hardware insight to improve the actual efficiency. SqueezeNext [62] uses a hardware simulator to adjust the macro-architecture of the network for better efficiency. ShiftNet [206] proposes a hardware-friendly shift operator to replace expensive spatial convolutions. AddressNet [232] designed three shift-based primitives to accelerate GPU inference.

3.2.2 CNN Quantization

CNN quantization aims to convert full-precision weights and activations of a network to low-precision representations to reduce the computation and storage cost. Early works [75, 238] mainly focus on quantizing weights while still using full-precision activations. Later works [37, 162, 235, 241] quantize both weights and activations. Many previous works [162, 235, 238] see serious accuracy loss if the network is quantized to low precisions. Normally, an accuracy loss of more than 1% is already considered significant. Also, in many works [37, 238], quantized weights or activations are represented by low-precision numbers multiplied with some floating-point coefficients. This can bring several challenges to hardware implementation. Last but not least, most of the previous works report quantization results on inefficient models such as VGG, AlexNet, and ResNet. Given that those models are redundant, quantizing them to lower precisions is much easier. We have not yet seen any work which successfully applies quantization to efficient models.

3.2.3 Hardware Designs

Most existing CNN hardware research has focused on improving the performance of either standalone 3×3 convolution layers or a full-fledged, large CNN on large FPGA devices. [222] quantitatively studies the computation throughput and memory bandwidth requirement for CNNs. [129, 226] present their own optimizations for CNNs based on analytical performance models. They achieve high throughput on VGG16 using their proposed design methodology with OpenCL. [223] designs convolution in the frequency domain to reduce the compute intensity of the CNN. They demonstrate good power performance results on VGG16, AlexNet, and GoogLeNet. [142] implements a ternary neural network on high-end Intel FPGAs and achieves higher performance/Watt than Titan X GPU. Most of the works mentioned above and others [10, 118, 202], target inefficient CNNs on the middle to high-end FPGA devices. For compact CNNs, [196] demonstrates a binary neural network(BNN) FPGA design that performs CIFAR10 classification at 21906 frames per second(FPS) with 283 μs latency on Xilinx ZC706 device. The BNN reports an accuracy of 80.1%. [137, 231] run the BNN on a

smaller device ZC7020. Although all three works achieve promising frame rates, they have not implemented more extensive neural networks for the ImageNet classification. It should be noted that classification on CIFAR10 dataset is orders of magnitude simpler than ImageNet, since CIFAR10 contains 100x fewer classes, 26x fewer images, and 49x fewer pixels in each image. Networks trained on CIFAR10 dataset also have way smaller complexity compared to those trained on ImageNet. In comparison, networks for ImageNet classification are closer to real-world applicability. [159] first attempted to deploy VGG16 for ImageNet classification on embedded device zc7020 and achieved a frame rate of 4.45 fps. Later [66] improved the frame rate to 5.7 fps. However, their frame rate was relatively low for real-time image classification tasks. [19, 97, 159] have achieved a high frame rate on smaller devices, however, the accuracy of their network is not on par with [66] for ImageNet classification.

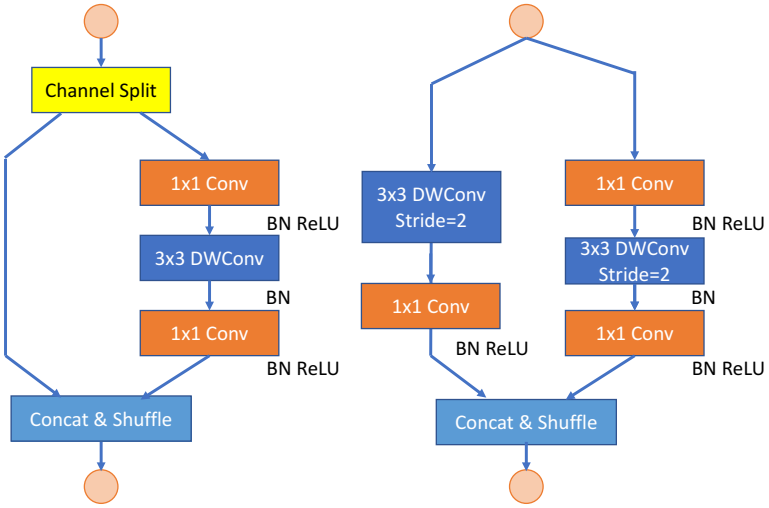
3.3 Synetgy CNN Design

We discuss the CNN design in this section. The design of our CNN incorporates the feedback from both the computer vision applications and hardware accelerator design. Specifically, an ideal CNN model for embedded FPGA acceleration should satisfy the following aspects: 1) The network should not contain too many parameters or FLOPs but should maintain a competitive accuracy. 2) The network structure should be hardware friendly to allow efficient scheduling. 3) The network's operation set should be simplified for efficient FPGA implementation. 4) The network's weights and activations should be quantized to low-precision fixed-point numbers without much accuracy loss.

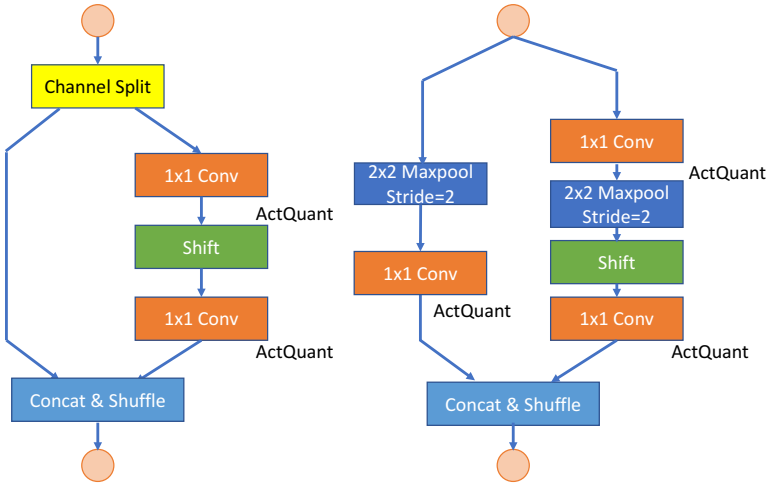
3.3.1 ShuffleNetV2

We select ShuffleNetV2-1.0x [128] as our starting point. ShuffleNetV2 is one of the state-of-the-art efficient models. It has a top-1 accuracy of 69.4% on ImageNet (2% lower than VGG16), but contains only 2.3M parameters (60x smaller than VGG16) and 146M FLOPs (109x smaller than VGG16).

The block-level structure of ShuffleNetV2 is illustrated in Fig. 3.1a. The input feature map of the block is first split into two parts along the channel dimension. The first branch of the network does nothing to the input data and directly feeds the input to the output. The second branch performs a series of 1×1 convolutions, 3×3 depth-wise convolutions, and another 1×1 convolution operations on the input. Outputs of two branches are then concatenated along the channel dimension. Channel shuffle [228] is then applied to exchange information between branches. In down-sampling blocks, depth-wise 3×3 convolutions with a stride of 2 are applied to both branches of the block to reduce the spatial resolution. 1×1 convolutions are used to double the channel size of input feature maps. These blocks are cascaded to build a deep CNN. We refer readers to [128] for the macro-structure description of the ShuffleNetV2.



(a) ShuffleNetV2 blocks [128].



(b) Our modified DiracDeltaNet blocks. We replace depth-wise convolutions with shift operations. In the downsampling blocks, we use stride-2 max-pooling and shift operations to replace stride-2 depth-wise convolutions. We also double the filter number of the 1st 1×1 convolution on the non-skip branch in each module.

Figure 3.1: ShuffleNetV2 blocks vs. DiracDeltaNet blocks

We select ShuffleNetV2-1.0x not only because of its small model size and low FLOP count but also because it uses concatenative skip connections instead of additive skip connections. Additive skip connections, as illustrated in Fig. 3.2(a), were first proposed in [78]. It effectively alleviates the difficulty of training deep neural networks and therefore improves accuracy. It is widely used in many CNN designs. However, additive skip connections are not efficient on FPGAs. As illustrated in Fig. 3.2(a), both the skip and the residual branches' data need to be fetched on-chip to conduct the addition. Though addition does not cost too much

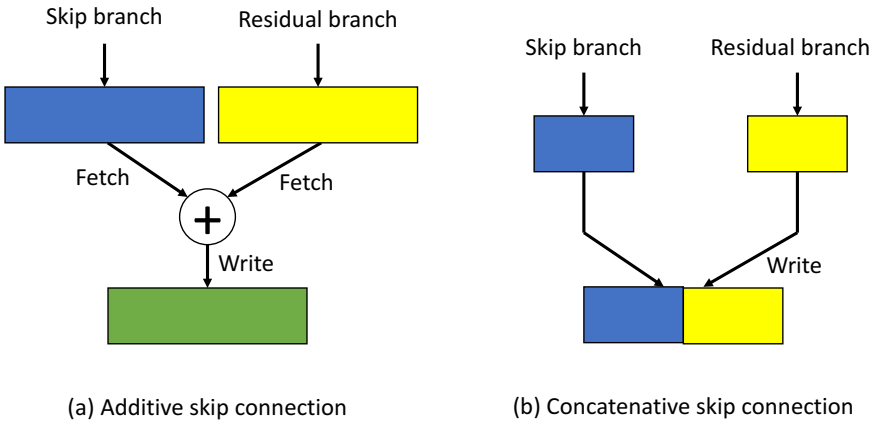


Figure 3.2: Additive Skip Connections vs. Concatenative Skip Connections. Rectangles represent data tensors.

computation, the data movement is expensive. Concatenative skip connections, as illustrated in Fig. 3.2(b), were first proposed in [87]. It has a similar positive impact on CNN training. With concatenative skip connections, data from the skip branch is already in off-chip DRAMs. So we can concatenate the two branches simply by writing the residual branch data next to the skip branch data. This avoids the extra memory access in additive skip connections and alleviates the memory bandwidth pressure.

3.3.2 DiracDeltaNet

Based on ShuffleNetV2, we build DiracDeltaNet through the following modifications: 1) we replace all the 3×3 convolutions with shift and 1×1 convolutions; 2) we reduce the kernel size of max-pooling from 3×3 to 2×2; 3) we modify the order of channel shuffle.

We replace all 3×3 convolutions and 3×3 depth-wise convolutions with shift operations and 1×1 convolutions. The motivation is that smaller convolution kernel sizes require less reuse of the feature map, resulting in a simpler data movement schedule, control flow, and timing constraint. As pointed out by [206], CNNs rely on spatial convolutions (3×3 convolutions and 3×3 depth-wise convolutions) to aggregate spatial information from neighboring pixels to the center position. However, spatial convolutions can be replaced by a more efficient operator called shift. The shift operator aggregates spatial information by copying nearby pixels directly to the center position. This is equivalent to shifting one channel of feature map towards a certain direction. When we shift different channels in different directions, the output feature map's channel will encode all the spatial information. A comparison between 3×3 convolution and shift is illustrated in Fig. 3.3. A module containing a shift and 1×1 convolution is illustrated in Fig. 3.4.

For 3×3 depth-wise convolutions, we directly replace them with shift operations, as shown

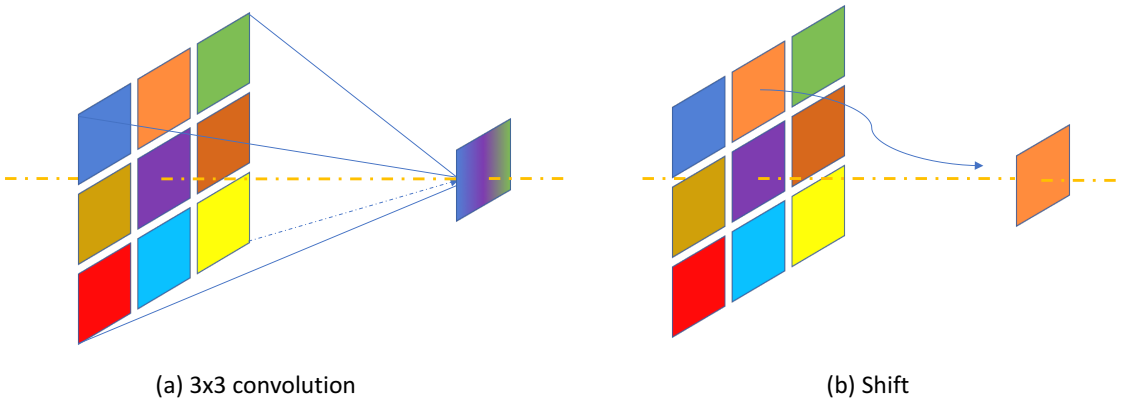


Figure 3.3: 3×3 Convolution vs. Shift. In 3×3 convolutions, pixels in a 3×3 region are aggregated to compute one output pixel at the center position. In the shift operation, a neighboring pixel is directly copied to the center position.

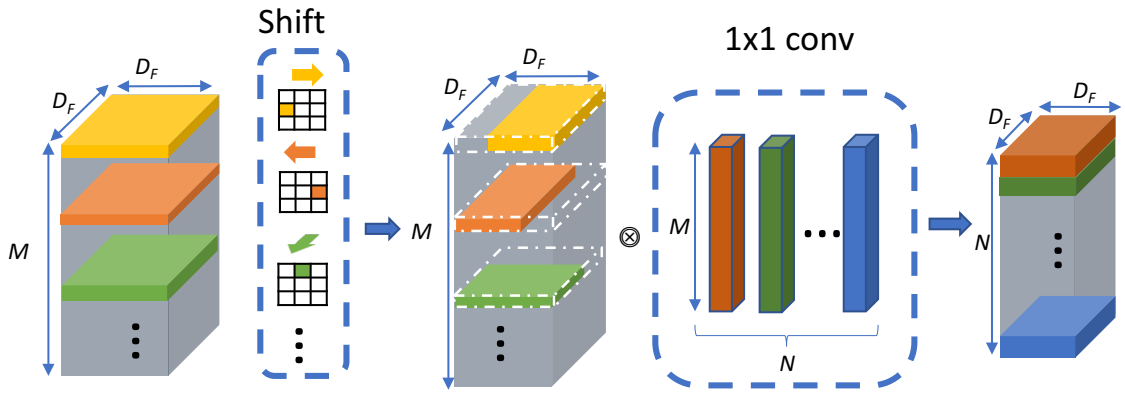


Figure 3.4: Using shift and 1×1 convolutions to replace 3×3 convolutions. This figure is from [206].

in Fig. 3.1b. This direct replacement can lead to some accuracy loss. To mitigate this, we double the output filter number of the first 1×1 convolution on the non-skip branch from Fig. 3.1b. Nominally, doubling the output channel size increases both FLOP count and parameter size by a factor of 2. However, getting rid of 3×3 convolutions allows us to design a computing unit customized for 1×1 convolutions with higher execution efficiency than a comparable unit for 3×3 depth-wise convolutions. In the downsample block, we directly replace the stridden 3×3 depth-wise convolutions with a stride-2 2×2 max-pooling. Unlike [206], our shift operation only uses four cardinal directions (up, down, left, right) in addition to the identity mapping (no-shift). This simplifies our hardware implementation of

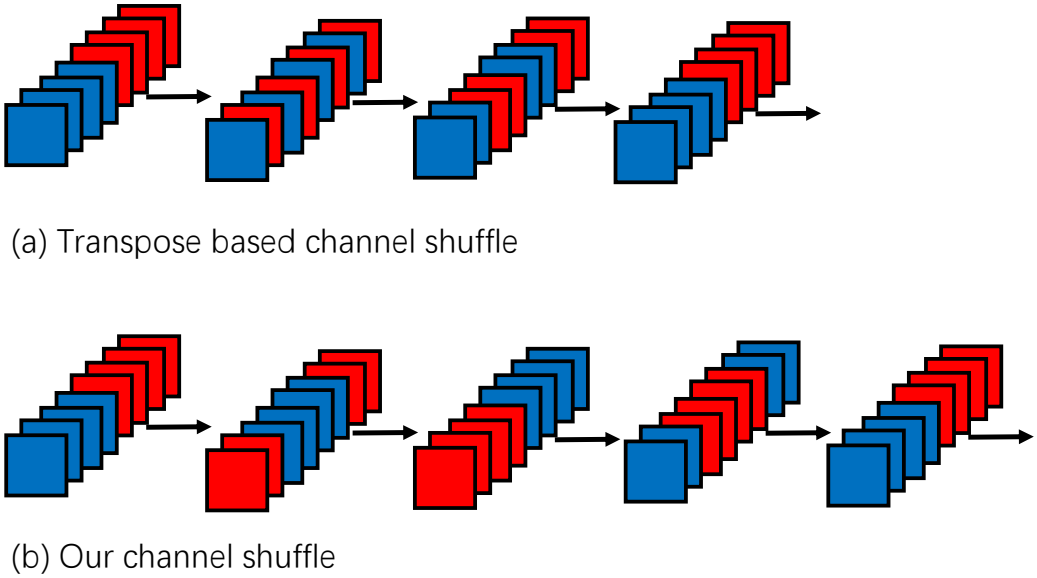


Figure 3.5: Transpose Based Shuffle (ShuffleNetV2) vs. Our HW Efficient Shuffle (DiracDeltaNet)

the shift operation without hurting accuracy.

The first stage of ShuffleNetV2 consists of a 3×3 convolution with a stride of 2 and filter number of 24. It is then followed by a 3×3 max-pooling with a stride of 2. We replace these two layers with a module consisting of a series of 1×1 convolution, 2×2 max-pooling, and shift operations, as shown in Table 3.1. Compared with the original 3×3 convolutions, our proposed module has more parameters (2144 vs. 648) and FLOPs (30.5M vs. 8.1M). But the implementation and execution cost of the proposed first stage is negligible compared to a 3×3 convolution layer. After training the network, we find that this module gives near equal accuracy to the original 3×3 convolution module. With our new module, we can eliminate the remaining 3×3 convolutions from our network, enabling us to allocate more computational resources to 1×1 convolutions and thereby increasing parallelism and throughput.

In addition to replacing all 3×3 convolutions, we also reduce the max-pooling kernel size from 3×3 to 2×2. Using the same pooling kernel size as the stride eliminates the need to buffer extra data on the pooling kernel boundaries, thereby achieving better efficiency. Our experiments also show that reducing the max-pooling kernel size does not impact accuracy.

We also modify the channel shuffle's order to make it more hardware efficient. ShuffleNetV2 uses transpose operation to mix channels from two branches. This is illustrated in Fig. 3.5(a), where blue and red rectangles represent channels from different branches. The transpose-based shuffling is not hardware friendly since it breaks the contiguous data layout. Performing channel shuffle in this manner will require multiple passes of memory read and write. We

Table 3.1: Macro-structure of DiracDeltaNet

Layer	Output size	Kernel size	Stride	#Repeat	Output channel
Image	224				3
Conv1	224	1	1	1	
Maxpool	112	2	2	1	32
shift	112	3	1	1	
Conv2	112	1	1	1	
Maxpool	56	2	2	1	64
shift	56	3	1	1	
Stage 2	28		2	1	128
	28		1	3	
Stage 3	14		2	1	256
	14		1	7	
Stage 4	7		2	1	512
	7		1	3	
Conv5	7	1	1	1	1024
GlobalPool	1	7		1	1024
FC				1	1000

Table 3.2: ShuffleNetV2-1.0x vs. DiracDeltaNet

	MACs	#Params	Top-1 acc	Top-5 acc
ShuffleNetV2-1.0x	146M	2.3M	69.4%	-
DiracDeltaNet	330M	3.3M	68.9%	88.7%

propose a more efficient channel shuffle showed in Fig. 3.5(b). We perform a circular shift to the feature map along the channel dimension. We can have the same number of channels exchanged between two branches while preserving the contiguity of the feature map and minimizing the memory accesses.

We name the modified ShuffleNetV2-1.0x model as DiracDeltaNet. The name comes from the fact that our network only contains 1×1 convolutions. With a kernel size of 1, the kernel functions can be seen as discrete 2D Dirac Delta functions. DiracDeltaNet's macro-structure is summarized in Table 3.1. Stage 2,3, 4 consist of chained DiracDeltaNet blocks depicted in Fig. 3.1 with different feature map sizes, channel sizes, and strides. We adopt the training recipe and hyperparameters described in [128]. We train DiracDeltaNet for 90 epochs with linear learning rate decay, the initial learning rate of 0.5, 1024 batch size, and 4e-5 weight decay. A comparison between ShuffleNetV2-1.0x and our DiracDeltaNet is summarized in Table 3.2.

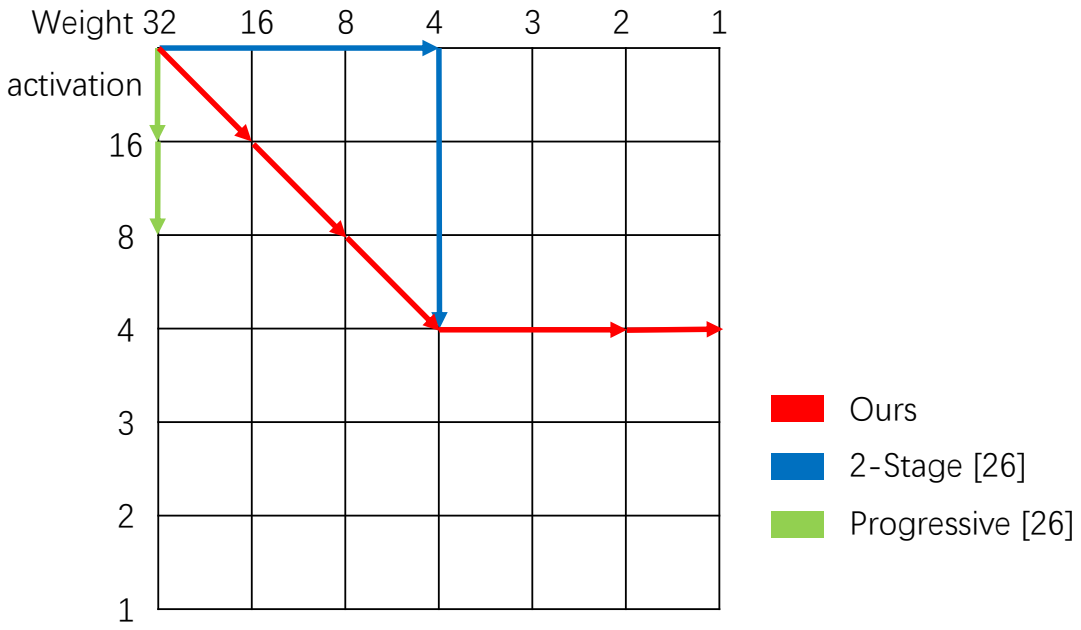


Figure 3.6: Quantization Grid

Table 3.3: Quantization Result on DiracDeltaNet

	full	w4a4
Top-1 Acc	68.9%	68.3%
Top-5 Acc	88.7%	88.1%

3.3.3 CNN Quantization

To further reduce the cost of DiracDeltaNet, we apply quantization to convert floating point weights and activations to low-precision integer values. For network weights, we follow DoReFa-Net [235] to quantize full-precision weights as

$$w_k = 2Q_k(\frac{\tanh(w)}{2\max(|\tanh(w)|)} + 0.5) - 1. \tag{3.1}$$

Here, w denotes the latent full-precision weight of the convolution kernel. $Q_k(\cdot)$ is a function that quantizes its input in the range of $[0, 1]$ to its nearest neighbor in $\{\frac{i}{2^k-1}|i = 0, \cdots 2^{k-1}\}$.

We follow PACT [37] to quantize each layer's activation as

$$\begin{aligned} y^l = PACT\left(x^l\right) = \frac{\left|x^l\right| - \left|x^l - \left|\alpha^l\right|\right| + \left|\alpha^l\right|}{2}, \\ y^l = Q_k\left(y^l / \left|\alpha^l\right|\right) \cdot \left|\alpha^l\right|. \end{aligned} \tag{3.2}$$

x^l is the activation of layer-l. $PACT(\cdot)$ is a function that clips the activation x^l to the range between $[0, |\alpha^l|]$. α^l is a layer-wise trainable upper bound, determined by the training of the network. It is observed that during training α^l can sometimes become a negative value, which affects the correctness of the PACT [37] function. To ensure α^l is always positive and to increase training stability, we use the absolute value of the trainable parameter α^l rather than its original value. y^l is the clipped activation from layer-l and it is further quantized to y_k^l, a k-bit activation tensor. Note that activations from the same layer share the same floating point coefficient α^l, but activations from different layers can have different coefficients. This is problematic for the concatenative skip connection, since if the coefficients α^l and α^{l-1} are different, we need to first cast y_k^{l-1} and y_k^l from fixed-point to floating point, re-calculate a coefficient for the merged activation, and quantize it again to new fixed-point numbers. This process is very inefficient.

In our experiment, we notice that most of the layers in the DiracDeltaNet have similar coefficients with values. Therefore, we rewrite equation (3.2) as

$$y^l = Q_k\left(y^l / |\alpha^l|\right) \cdot |s|. \tag{3.3}$$

where s is a coefficient shared by the entire network. This step ensures that activations from different layers of the network are quantized and normalized to the same scale of $[0, |s|]$. As a result, we can concatenate activations from different layers directly without extra computation. Moreover, by using the same coefficient s across the entire network, the convolution can be computed completely via fixed-point operations. The coefficient s can be fixed before or leave it as trainable. A general rule is that we should let s have similar values of α^l from different layers. Otherwise, if s/α^l is either too small or too large, it can cause gradient vanishing or exploding problems in training, which leads to a worse accuracy of the network.

In our network, we merge the PACT function and activation quantization into one module and name it ActQuant. The input to ActQuant is the output of 1×1 convolutions. Since the input and weight of the convolution are both quantized into fixed-point integers, the output is also integers. Then, ActQuant is implemented as a look-up table whose parameters are determined during training and fixed during inference.

We follow [241] to quantize the network progressively from full-precision to the desired low-precision numbers. The process is illustrated in Fig. 3.6, where the x-axis denotes bit-width of weights and the y-axis denotes the bit-width of activations. We start from the full-precision network, train the network to convergence, and follow a path to progressively reduce the precision for weights or activations. At each point, we fine-tune the network for 50 epochs with step learning rate decay. Formally, we denote each point in the grid as a quantization configuration $\mathscr{C}_{w,a}(N_w)$. Here w represents the bitwidth of weight. a is the bitwidth of activation. N_w is the network containing the quantized parameters. The starting configuration would be the full precision network $\mathscr{C}_{32,32}(N_{32})$. Starting from this configuration, one can either go down to quantize the activation or go right to reduce the bitwidth of weight. More aggressive steps can be taken diagonally or even across several grids. The two-stage and progressive optimization methods proposed in [241] can be represented as two paths in Fig. 3.6.

In our work, we start from $\mathscr{C}_{32,32}(N_{32})$. Then we use N_{32} to initialize N_{16} and obtain $\mathscr{C}_{16,16}(N_{16})$. And we apply step lr decay fine-tuning onto N_{16} to recover the accuracy loss due to the quantization. After several epochs of fine-tuning, we get the desired low-precision configuration $\mathscr{C}_{16,16}(N'_{16})$ with no accuracy loss. Following the same procedures, we are able to first go diagonally in the quantization grid to $\mathscr{C}_{4,4}(N_4)$ with less than 1% top-5 accuracy loss compared to its full precision counterpart.

We use a pre-trained ResNet50 label-refinery [13] to boost the accuracy of the quantized model. Even with such low-precision quantization, our quantized model still preserves a very competitive top-5 accuracy of 88.1%. Most of the previous quantization works [37, 235, 241] are only effective on large models such as VGG16, AlexNet, or ResNet50. Our quantization result is summarized in Table 3.3.

3.4 Synetgy Hardware Design

As mentioned in section 3.3.2, we aggressively simplified ShuffleNetV2's operator set. Our modified network is mainly composed of the following operators:

- 1×1 convolution
- 2×2 max-pooling
- shift
- shuffle and concatenation

Our accelerator, Synetgy, is tailored to support only the operators above. This allows us to design more specialized compute units with simpler control and further improve hardware efficiency. The compute of the fully connected layer can be mapped onto our convolution unit. Shuffle operation is not fully supported on FPGA. CPU-based memory copy is needed to maintain the memory layout. And the remaining average-pooling layer, which is not supported on the FPGA is offloaded to the ARM processor on the SoC platform. Algorithm-hardware co-design results in simplified operators and increased productivity for hardware implementation. The accelerator implementation only took two people working for one month using HLS.

3.4.1 The accelerator architecture

Fig. 3.7 shows the overall accelerator architecture design. Our accelerator, highlighted in light yellow, can be invoked by the CPU for computing one 1×1 Conv-Pooling-Shift-Shuffle subgraph at a time. The CPU provides supplementary support to the accelerator. Both the FPGA and the CPU are used to run the network.

In quantized DiracDeltaNet, weights are 4-bit, input and output activations are 4-bit, and the largest partial sum is 17-bit. The width of partial sum is determined by the input feature

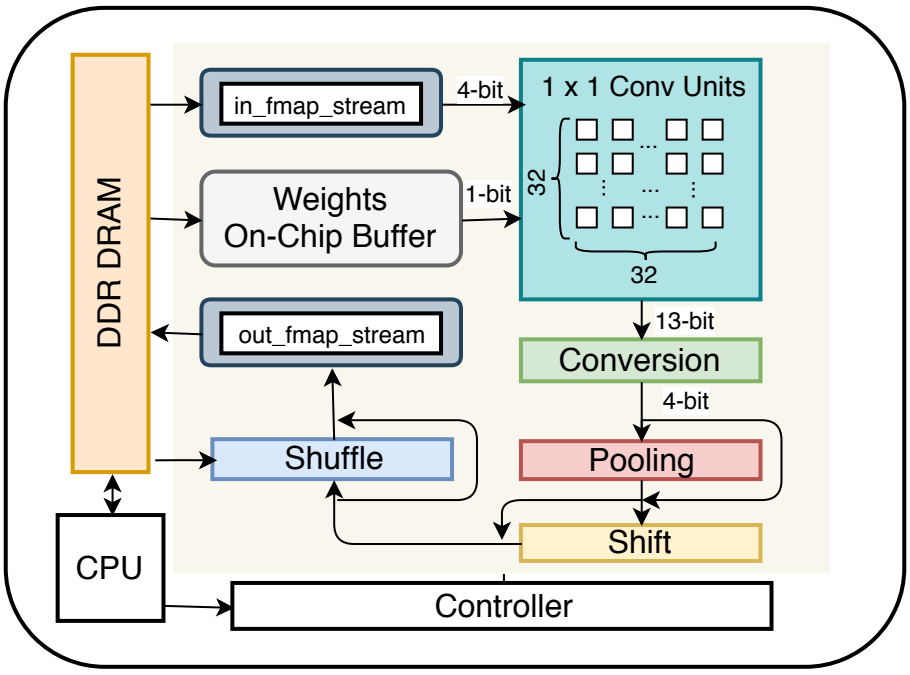


Figure 3.7: Accelerator Architecture

Table 3.4: Notations

Notation	Type	Description
WIDTH	variable	width of feature map
HEIGHT	variable	height of feature map
IC_TOTAL	variable	total input channel size
OC_TOTAL	variable	total output channel size
IC	constant: 32	parallelism on input channel dimension
OC	constant: 32	parallelism on output channel dimension

bit width and the largest channel size. Given that the largest channel size is 512, there are $2^4 \times 2^4 \times 512$ possible outcomes from the convolution, which requires 17 bits to represent.

3.4.1.1 Dataflow Architecture

Our hardware design is based on the dataflow architecture template [35, 213]. As illustrated in Fig. 3.7, we first extract a few process functions from the major operations including 1×1 convolution, 2×2 max-pooling, shift, shuffle and the memory load and store. We then chain

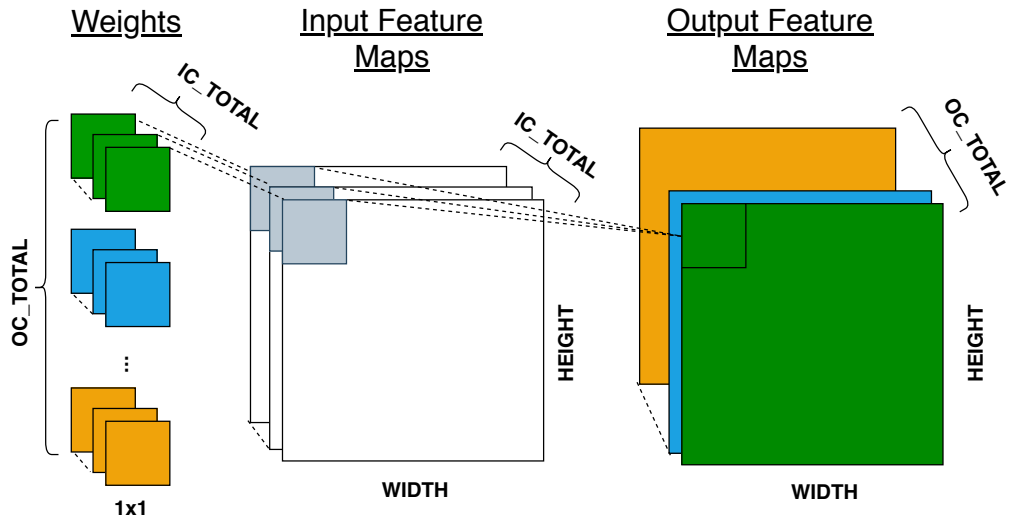


Figure 3.8: 1×1 Convolution

them together using FIFOs with blocking read and non-blocking write. Note that the write is blocking once the FIFO is full. All the process functions are running concurrently. The arrival of the data triggers the execution of each function. Therefore, more task-level parallelism can be explicitly exposed to the HLS tool in addition to the instruction-level parallelism.

3.4.1.2 Convolution Unit

The notations used in this section are listed in Table 3.4. As shown in Fig. 3.8, given an input feature map of size $WIDTH \times HEIGHT \times IC\_TOTAL$ and a weight kernel of size $IC\_TOTAL \times OC\_TOTAL$, the generated output feature map is of size $WIDTH \times HEIGHT \times OC\_TOTAL$ in 1×1 convolution. The 1×1 convolution is essentially a matrix-matrix multiplication.

Although [113] suggests a weight-stationary dataflow for 1 × 1 convolution dominant CNNs, we find it not applicable to our design as the bit width of weights is much smaller than the partial sums (4 bit vs. 17 bits). Transferring the partial sums on and off-chip will incur more traffic on the memory bus. Therefore, we adopt the output-stationary dataflow by retaining the partial sums in the local register file until an output feature is produced.

Fig. 3.9 shows how we schedule the workload onto the accelerator. Note that the nested loops starting at lines 17, 19 are automatically unrolled. Weights are prefetched onto on-chip BRAM $weight\_buf$. We first block our inputs so $IC \times OC$ multiplications can be mapped onto the compute units at each iteration (Line 13∼21). In every iteration, IC input features are fetched from the DRAM. They are convolved with OC number of weights of size IC and produce OC partial sums. Each iteration of the loop nest along the input channel dimension

```
 1 bw<4> in_fmap_dram[WIDTH*HEIGHT][IC_TOTAL/IC][IC]
 2 bw<4> out_fmap_dram[WIDTH*HEIGHT][OC_TOTAL/OC][OC]
 3 bw<4> weight_buf[OC_TOTAL/OC][IC_TOTAL/IC][OC][IC]
 4 bw<4> in_fmap_stream [IC]
 5 bw<1> weight_stream [OC][IC]
 6 bw<13> partial_sum_reg [OC]
 7 bw<4> out_fmap_stream [OC]
 8 #pragma HLS dataflow
 9 for idx in [0, WIDTH * HEIGHT):
10 | for oc_t in [0, OC_TOTAL/OC):
11 | | partial_sum_reg <- 0
12 | | for ic_t in [0, IC_TOTAL/IC):
13 | | #pragma HLS pipeline
14 | | | in_fmap_stream <- in_fmap_dram[idx][ic_t][:]
15 | | | weight_stream <- weight_buf[ic_t][oc_t][:][:]
16 | | | for ic in [0, IC):
17 | | | #pragma HLS|unroll
18 | | | | for oc in [0, OC):
19 | | | | #pragma HLS|unroll
20 | | | | | partial_sum_reg[oc] +=
21 | | | | | in_fmap_stream[ic] * weight_stream[oc][ic]
22 |  out_fmap_stream <- func_convert(partial_sum_reg)
23 |  out_fmap_dram[idx][oc_t][:] <- out_fmap_stream
```

Figure 3.9: Pseudo Code for Kernel Compute Scheduling

at line 12 takes $7 \sim 38$ cycles to finish based on the Vivado HLS report. Equivalently, it takes $7 \sim 38$ cycles to finish $IC \times OC$ 4/4 bit multiplication. The partial sums are stored in the registers, which can be simultaneously accessed in every cycle. The parameter IC and OC were tuned for the area performance tradeoff. Increasing them increases overall resource utilization but helps to reduce the total number of execution cycles.

Based on the roofline model [204], the attainable throughput is the compute-to-communication (CTC) ratio multiplied by the bandwidth when it is bandwidth bound. The CTC ratio of our compute unit for the input feature is OC_TOTAL (maximum number is 512 in DiracDeltaNet), which is a variable. A larger output channel size indicates a higher CTC ratio. According to our measurement, the maximum bandwidth of the DDR channel is 6GB/s, which means 6×2 Giga input features (1 Byte contains two 4-bit features) can be loaded. The theoretical memory bound throughput should be $512 \times 6 \times 2 = 6144$GMACs $= 12288$GOPs. For compute bound problems, the attainable throughput is dependent on the compute capability. In our case, it is $IC \times OC \times freq = 32 \times 32 \times 250MHz = 256GMACs$=512GOPs. Based on the analysis, the convolution unit will reach the bandwidth bound before it hits the computation roofline.

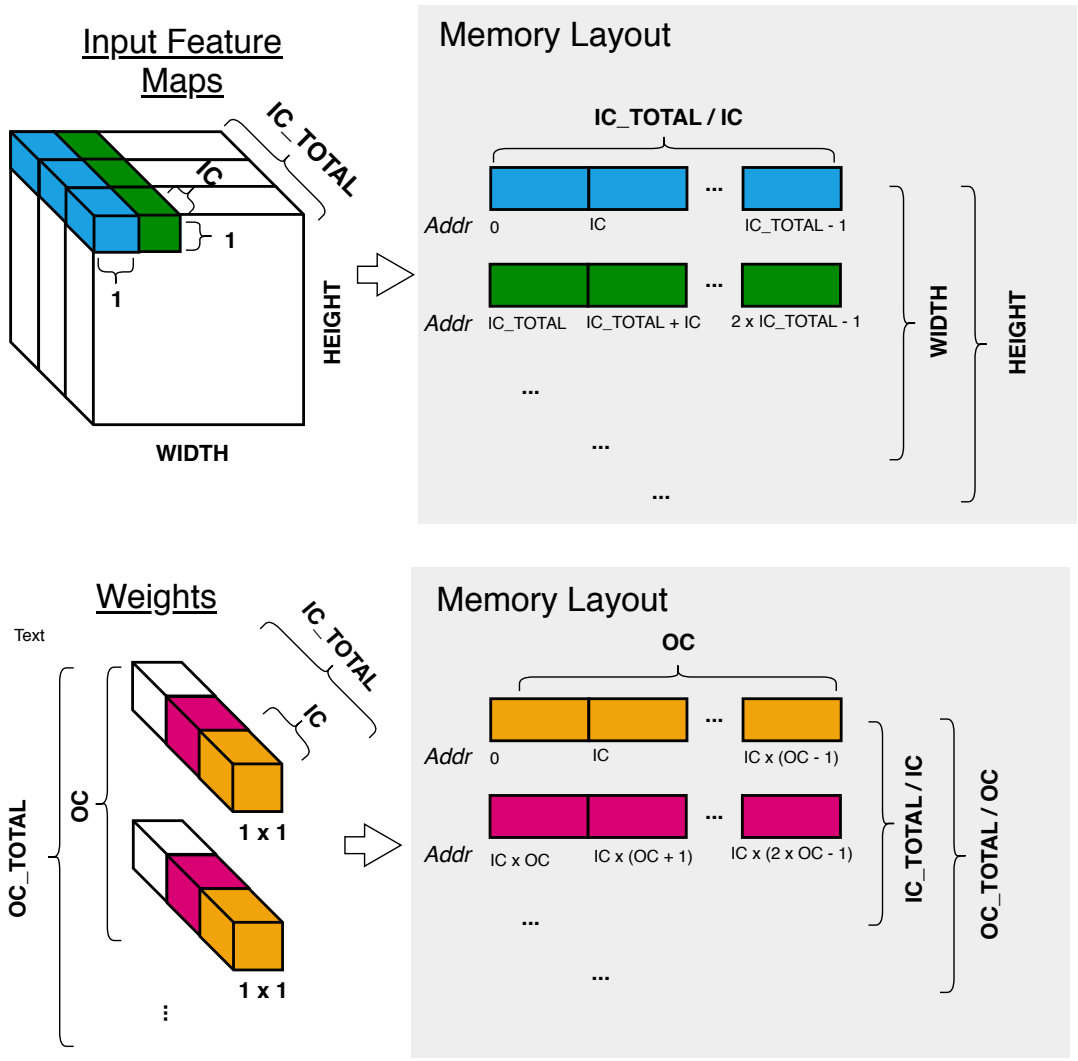


Figure 3.10: Input Layout in DRAM

3.4.1.3 Conversion Unit

The high bitwidth to low bitwidth conversion is performed immediately after the kernel computation. It is a step function with 16 intervals that converts 17-bit partial sum to 4-bit activation. The threshold values are different for each layer. All of the read-only threshold values are stored in on-chip BRAMs. An index number should be specified by the user function to select which set of threshold values to use for the compute of the current layer. In hardware, this unit is implemented by using 16 comparators. They are mapped onto a binary tree structure to reduce the circuit latency.

3.4.1.4 Pooling Unit

We adopt the line buffer design described in [231] to implement the 2×2 max-pooling layer. For every iteration, $(WIDTH + 1)$ of IC deep pixels are first fetched into the line buffers. Once the next pixel value is fetched, a 2×2 large sliding window is formed. For every two cycles, we compare the values in the 2×2 sliding window, output the largest one, and fetch the next two values. It takes IC_TOTAL/IC iterations to finish the compute.

3.4.1.5 Shift Unit

The line buffer design is also used for the shift operation. In the shift unit, the input images are first padded with one zero-value pixel at the width and height dimension. $(2 \times (WIDTH + 2) + 2)$ of pixels are then buffered, and a 3×3 sliding window is formed. The shift direction is different for different input channels. It is calculated based on the input channel index. After initialization, the unit is able to produce 1 output pixel per cycle.

3.4.1.6 Shuffle Unit

Shuffle is implemented by changing the address offset of output features during the writeback phase. Since the shuffle operation still requires us to concatenate the outputs from the previous DiracDeltaNet block to the current DiracDeltaNet block outputs, the CPU is used to copy the output from the previous DiracDeltaNet unit to the shuffled address. The memory copy operation should be done concurrently with the computation of the current DiracDeltaNet unit.

3.4.1.7 Fully Connected Unit

We don't explicitly design a dedicated unit to compute the FC layer. Instead, we map the compute of the FC layer onto our existing hardware convolution unit. The feature map size is 1 for the FC layer. While the convolution unit only supports 4-bit weight, the FC layer's computation is mapped in a bit-serial like manner. The convolution unit processes each bit of the FC weight iteratively, and bit shift is done by configuring the step function in the conversion unit.

3.4.2 Software

We use the ARM processor to control the layer-based accelerator and to compute the last 7×7 average-pooling layer that is not supported by the accelerator. The host application runs on a full Linux system on the ARM CPU, which controls the memory-mapped accelerator through the UIO driver interface. The Xilinx python-based PYNQ APIs [212] are used for fast deployment of the host software code on the Ultra 96 board.

Table 3.5: Resource Usage

LUT	FF	BRAM	DSP
51776 (73.4%)	42257 (29.9%)	159 (73.6%)	360 (100%)

Table 3.6: Performance comparison of Synetgy and the previous works.

	VGG-SVD [159]	AlexNet [123]	VGG16 [188]	VGG16 [66]	DoReFa [97]	FINN-R [19]	Ours
Platform	Zynq XC7Z045	Stratix-V	Stratix-V	Zynq 7Z020	Zynq 7Z020	Zynq ZU3EG	Zynq ZU3EG
Frame Rate (fps)	4.5	**864.7**	3.8	5.7	106.0	200.0	66.3
Top-1 Acc	64.64%	42.90%	66.58%	67.72%	46.10%	50.3%	**68.30%**
Top-5 Acc	86.66%	66.80%	87.48%	88.06%	73.10%	N/A	**88.12%**
Precision	16b	16b	8-16b	8b	2b	1-2b	4-4b
Frequency(MHz)	150	150	120	214	200	220	250
Power(W)	3.0	26.2	19.1	3.0	2.3	10.2	5.5

Table 3.7: Frame Rate on Different Batch Size

Batch Size	1	2	4	8	16
Frame Rate (fps)	41.4	53.6	62.6	65.6	66.3

3.5 Synetgy Experimental Results

We implement our accelerator, Synetgy, on the Ultra96 development board with Xilinx Zynq UltraScale+ MPSoC targeted at embedded applications. Table 3.5 shows the overall resource utilization of our implementation. We utilized 73% of the total LUTs on the FPGA, as the bit-level 4/4bit multiplications are mapped onto LUTs. BRAMs are mainly used for implementing the FIFO channels. DSPs are used for the address calculation for the AXI protocol. Our implementation runs at 250 MHz. Power measurements are obtained via a power monitor. We measured 5.3W with no workload running on the programming logic side and 5.5W max power on the Ultra96 power supply line when running our network.

We compare our accelerator against previous work in Table 3.6. As explained before, CNNs for ImageNet classification are usually orders of magnitude more complex than CIFAR10 classification. Therefore, we only compare accelerators targeting CNNs for ImageNet classification with reasonable accuracy. Our work focuses on achieving competitive accuracy while improving the actual inference speed in terms of frames per second. Our experiments show that we successfully achieve those two goals. From the table, we can make the following observations: 1) Synetgy achieves the highest top-1 and top-5 accuracy on ImageNet. The only previous work that comes close to our accuracy is [66], but its frame rate is 11.6× slower than ours. 2) Among the embedded accelerators whose top-1 accuracy is higher than 60%, which is a loose constraint, our model achieves the fastest inference speed. 3) Without the accuracy constraint, the speed of [19, 97, 123] can go as fast as 864.7 frames per second. But their accuracy is rather low. 4) The peak attainable throughput of our accelerator is 418

Table 3.8: Runtime Analysis for the First and Last DiracDeltaNet Blocks in Different Operator Configurations (Batch=10)

	Runtime (ms)	
	Block1	**Block2**
feature map size	28	7
in&out channel	128	512
conv only	1.531	0.989
conv+pool	1.530	0.993
conv+shift	1.537	0.996
conv+shuffle	4.409	1.636
overall	4.364	1.441

GOPs, which is close to the theoretical compute roofline. Our average throughput (47.09 GOPs) is currently limited by the low hardware utilization. The inefficiency is mainly from the software shuffle operations and the first convolution layer with an input dimension of 3 (much smaller than the hardware tiling factor IC). However, Synetgy still achieves a competitive frame rate, demonstrating the efficacy of our co-design methodology. We see the opportunity for significant frame rate improvement through further algorithm-hardware co-design.

The reported frame rate is achieved with batch size set to 16. There is a fixed software overhead for invoking the poll-based hardware accelerator. The computation latency of the DiracDelta Block1 in Table 3.8 is 0.15ms when the batch size is equal to 1. The latency for a single read on the accelerator control register is 0.40ms, which is greater than the actual compute time. In order to minimize this software overhead, we increase the batch size to schedule more computation running on the accelerator per invocation. Furthermore, the weights stored in on-chip BRAM get reused more when batch size is increased. The frame rates of implementations with different batch sizes are summarized in Table 3.7.

We break down the runtime of the whole heterogeneous system by bypassing one component of the system and measure the runtime. We observe that the CPU-based memory copy for the shuffle operation significantly degrades the performance. However, all other non-CNN components (sw average pooling, FC, PYNQ API call) impact the overall performance slightly.

To further understand the efficiency of various operators (1×1 convolution, 2×2 max-pooling, shift, and shuffle) implemented on FPGA and CPU, we measure the runtime of the DiracDeltaNet blocks with different configurations on Synetgy. The result is summarized in Table 3.8. We test 2 blocks with different input feature map and channel sizes. Note that the theoretical OPs of Block1 and Block2 are the same. As shown in the table, pooling and shift incur almost no performance drop. This is because the process functions for performing

these operations do not impose new bottlenecks on the dataflow pipeline. Software memory copy latency of shuffle is more significant on Block1 than Block2. This is because memory copy overhead is proportional to $HEIGHT \times WIDTH \times OC\_TOTAL$. But total OPs $HEIGHT \times WIDTH \times IC\_TOTAL \times OC\_TOTAL$ remains the same, meaning a smaller feature map needs less memory copy. The memory copy overhead can be possibly alleviated by running bare-metal C code on the CPU.

3.6 Co-design for Object Detection

As discussed in the previous section, deploying deep learning models on embedded systems for computer vision tasks has been challenging due to limited compute resources and strict energy budgets. The majority of existing work focuses on accelerating image classification, while other fundamental vision problems, such as object detection, have not been adequately addressed. Compared with image classification, detection problems are more sensitive to the spatial variance of objects, and therefore, require specialized convolutions to aggregate spatial information. To address this need, recent work introduces dynamic deformable convolution to augment regular convolutions. Regular convolutions process a fixed grid of pixels across all the spatial locations in an image. In contrast, dynamic deformable convolution may access arbitrary pixels in the image, with the access pattern being input-dependent and varying with spatial location. These properties lead to inefficient memory accesses of inputs with existing hardware. In the CoDeNet work discussed in this chapter, we harness the flexibility of FPGAs to develop a novel object detection pipeline with deformable convolutions. We show the speed-accuracy tradeoffs for a set of algorithm modifications, including irregular-access versus limited-range and fixed-shape on a flexible hardware accelerator. While the use of convolution kernels for computer vision is well-established, researchers have constantly been proposing new operations and new network designs to increase the model capability and achieve a better speed-accuracy tradeoff for various tasks. Deformable convolution [44, 239] is one of the novel operations that leads to state-of-the-art accuracy for object recognition with more effective use of parameters. Many neural network designs with top accuracy [155, 225] for object detection on the COCO dataset [124] use deformable convolution in their design. Unlike conventional convolutions with fixed a geometric structure, deformable convolution is an input-adaptive operation that samples inputs from variable offsets generated based on the input features during inference. Compared to conventional convolutions, deformable convolution provides a performance advantage due to: *variable sampling scales* and *variable sampling geometry*. The sampling range at each point varies, allowing the network to capture objects of different scales. Also, the geometry of the sample points is not fixed, allowing the network to capture objects of different shapes. Several previous studies [125] [34] [121] [236] have also shown that deformable convolution design lies on the Pareto-frontier of the speed-accuracy tradeoff for object detection on GPUs.

There are several challenges in supporting deformable convolution on off-the-shelf embedded deep learning accelerators: (i) The memory accesses for the input feature maps

are irregular, depending on the dynamically generated offsets. Many existing accelerators' instruction set architecture and the control logic are insufficient in supporting the random memory access patterns. In addition, the less contiguous memory access patterns limit the length of bursting memory accesses and incur more memory requests. (ii) There is less spatial reuse for the input features. Many accelerators are designed for output-stationary or row-stationary dataflow, which leverages input reuse. With deformable convolution, due to the variable filter offsets, the loaded input pixel for the current output pixel can no longer be reused by its neighboring output pixels. The lack of reuse significantly affects performance. (iii) There is an increased memory bandwidth requirement for loading the variable offsets.

For this work, we leverage the efficiency and flexibility of FPGA and the readily available high-level design tools to address the challenges from deformable convolutions. Adopting theco-design methodology, we develop FPGA accelerators tailored to each algorithmic change and use these to study the accuracy-efficiency tradeoffs for each algorithmic modification.

We propose the following modifications to the deformable convolution operation to make it more hardware friendly:

1. Limit the adaptive offsets to a fixed range to allow buffering of inputs and exploit full input reuse.

2. Constrain the arbitrary offset displacements into a square shape to reduce the overhead from loading the offsets and to enable parallel accesses to on-chip memory.

3. Round the offset displacements to integers and remove the fractional, bilinear interpolation operation for calculating the final sampling value.

4. Use depth-wise convolution to reduce the total number of Multiply-Accumulate operations (MACs).

We evaluate each modification on an FPGA System-on-Chip (SoC) that includes both an FPGA fabric and a hardened CPU core. We leverage the shared last-level cache (LLC) included in its full hardened processor system to efficiently exploit the locality of deformable convolution with data-dependent memory access patterns. We then optimize the hardware based on each algorithm modification to demonstrate its advantage in efficiency over the original operation. With these proposed algorithm modifications, we devise a line-buffer design to efficiently support our optimized depthwise deformable convolutional operation.

To demonstrate the full capability of the co-designed operation, we also design an efficient deep neural network (DNN) model CoDeNet for object detection using ShuffleNetV2 [128] as the feature extractor. We quantize the network to 4-bit weights and 8-bit activations with a symmetric uniform quantizer using the block-wise quantization-aware fine-tuning process [51]. Our main contributions include:

1. Co-design of a deformable convolution operation on FPGA with hardware-friendly modifications (depthwise, rounded-offset, limited-range, limited shape), showing up to 9.76× hardware speedup.

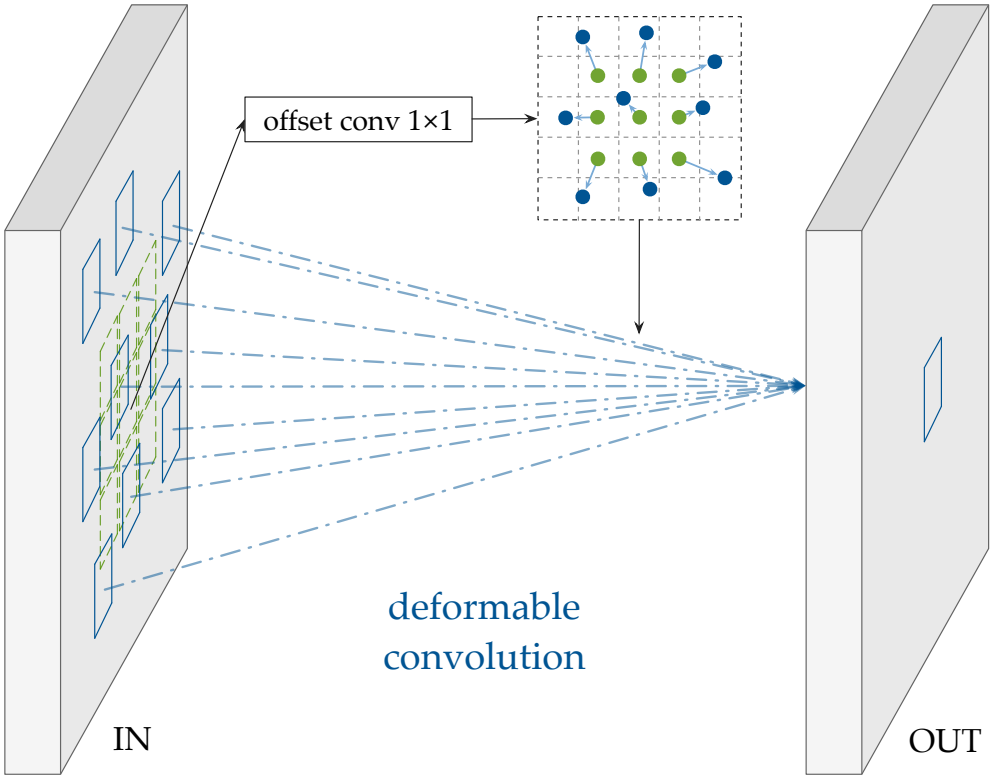


Figure 3.11: Deformable convolution with input-adaptive displacement offsets generation. Deformable convolution in our design first generates the sampling offsets from the input feature map a using a 1×1 convolution. Then it samples the same input feature map based on the generated offsets and performs a 3×3 convolution to aggregate the corresponding spatial features.

2. Development of an efficient DNN model for object detection with co-designed input-adaptive deformable convolution that achieves 67.1 AP50 on Pascal VOC with 2.9 MB parameters. The model is 20.9× smaller but 10% more accurate than the Tiny-YOLO.

3. Implementation of an FPGA accelerator to support the target neural network design that runs at 26 frames per second on Pascal VOC with 61.7 AP50.

3.7 CoDeNet Background and Motivation

3.7.1 Object Detection

Object detection is a more challenging task than image classification as it performs object localization in addition to object classification and requires prediction on spatially variant objects. Existing solutions for object detection can be categorized into two approaches: two-

stage detector and one-stage detector. In two-stage algorithms, the detector first proposes a set of regions of interest and then performs object classification on the selected regions. Faster R-CNN [166], a two-stage algorithm, introduces Region Proposal Network (RPN) for efficient region proposal. RPN is widely adopted in two-stage algorithms as it reduces the overhead of region proposals by sharing features from the main detection network. On the other hand, one-stage algorithms skip the region proposal stage and directly run detection over a dense sampling of all possible regions. Single Shot MultiBox Detector (SSD) [126], a popular one-stage detector, leverages a pyramidal feature hierarchy in the feature extraction network to efficiently encode objects in various sizes. You Only Look Once (YOLO) [163] [165] is another popular one-stage detector using fully convolutional network. The algorithm divides the input image into a grid with a fixed number of cells. Each cell in the grid predicts the bounding boxes of objects. A prediction of the bounding box comprises location information, confidence scores, and the conditional probability of the object class. The location information consists of the coordinates of the object center and the object size. The confidence scores indicate the probability of an object in these boxes.

In this work, we use a one-stage anchor-free detector called CenterNet [236] due to its better Pareto efficiency for the speed-accuracy tradeoff compared to the concurrent works [54] [115] [116] [237]. In contrast to most anchor-free detectors where Non-Maximum Suppression (NMS) mechanism is still required to remove the duplicated predictions, CenterNet directly generates the center points for each object without any post-processing. This property greatly reduces the complexity of implementing the detector pipeline in hardware.

As for the evaluation metrics for object detection, a common practice is to use the average precision (AP) and intersection over union (IoU). AP computes the average precision value achieved with different recall values. The precision value, calculated as $\frac{\text{true positive}}{\text{true positive+false positive}}$, indicates the percentage of predictions that are correct. The recall value, defined as $\frac{\text{true positive}}{\text{true positive+false negative}}$, measures the capability to correctly classify all positives. IoU is defined as the intersection between the predicted boxes and the target boxes over the union of the two. The default evaluation metric for VOC dataset [55] is AP50, which indicates that the prediction would be seen as correct if the corresponding IoU ≥ 0.5. The main metric for COCO is the mean of the average precisions at IoU from 0.5 to 0.95 with a step size of 0.05.

3.7.2 Deformable Convolution

Compared to image classification, one challenge in object detection is to capture geometric variations of each object, such as scale, pose, viewpoint, and part deformation. Besides, different objects located in different regions of the same image can be geometrically different, making it hard to capture all features in one pass. State-of-the-art approaches [34] [121] [125] [178] [236] address these challenges by harnessing deformable convolution [44] [239]. As demonstrated in Figure 3.11, deformable convolution samples the input feature map using the offsets dynamically predicted from the same input feature map, after which it performs a

(a) car (b) lawn

Figure 3.12: Example for the input-adaptive deformable convolution sampling locations and offset range distribution for different active detection units. (a) the sampling locations for the car as an active unit. (b) the sampling locations for lawn in the background.

regular convolution over the features sampled from the predicted offsets. The convolution layer for generating the offsets is typically composed of one 1×1 or 3×3 convolution layer. It is jointly trained with the rest of the network using standard backpropagation in an end-to-end manner. This way, the gradient updates not only the weights of the convolutions but also the sampling locations for the convolutions. Such operation design enables more flexible and adaptive sampling on different input feature maps.

Unlike the regular convolution with fixed geometry, the receptive fields of deformable convolution can be of various shapes to capture objects with different scales, aspect ratios, and rotation angles. In addition, deformable convolution is both spatial-variant and input-adaptive. In other words, its sampling patterns and offsets vary for different output pixels in the same input feature map and also vary across different input feature maps. In Figure 3.12(a)(b), we show how the sampling locations (red dots) change with the different active detection units (the object with a green dot on it). Most of the offsets are within the $[-1, 4]$ range for the example image. Albeit the operation augments and enhances the capability of the existing convolution for object detection, its dynamic nature poses extra challenges to the existing hardware.

3.7.3 Algorithm-hardware Co-design for Object Detection

Many prior acceleration works [240] [130] [138] [76] [229] [214] [203] have demonstrated the effectiveness of the co-design methodology for the deployment of real-time object detection on FPGAs. [130] customizes SSD300 [126] by replacing operations, such as dilated convolutions, normalization, and convolutions with larger stride, with more efficiently supported ones on FPGAs. [138] adapts YOLOv2 [165] by introducing a binarized network as the backbone for feature extraction to leverage the low-precision support of FPGA. Meanwhile, the FINN-R framework [19] further explores the benefits of integrating quantized neural networks (QNN) into Yolo-based object detection systems. Real-time object detection for live video streaming system [154] is developed with the FINN-based QNNs. [76] devised an automatic co-design flow on embedded FPGAs for the DJI-UAV [215] dataset with 95 categories targeting unmanned aerial vehicles. The flow first constructs DNN basic building blocks called bundles, estimates their corresponding latency and cost on hardware, and selects the ones on the Pareto front for latency and resources trade-off. Then it starts a two-phase DNN evaluation to search for the bundles on the Pareto front of the accuracy-latency trade-off and then fine-tune the design of the selected bundles. SkyNet [229] searched by this co-design flow achieves the best performance (based on a combination of throughput, power, and detection accuracy) on embedded GPUs and FPGAs. Differing from prior work, we study a novel and efficient operation, deformable convolution, for object detection. In addition to modifying the neural network design, we also co-design the operation for better hardware efficiency.

3.7.4 Quantization

Quantization [235] [95] [224] [51] [26] is a critical technique for efficiently deploying neural network models on embedded devices. It alleviates the memory bottleneck by compressing the weights in neural network models into ultra-low precision such as 4 bits. Moreover, quantizing both the weights and activations enables the use of cheaper low-precision integer arithmetics on hardware. For DNN deployment on embedded FPGAs without floating-point arithmetic support, quantization is one key and necessary modification.

However, directly performing aggressive layer-wise quantization can result in significant accuracy degradation [108]. Many prior works have attempted to address this accuracy drop with various techniques, such as non-uniform learnable quantizer [224], mixed-precision quantization [50], progressive fine-tuning [233] as well as group-wise [179] and channel-wise quantization [108]. Although these methods can better preserve the accuracy of the pre-trained model, they increase the complexity of hardware implementation. They can introduce non-negligible overhead on both latency and memory usage. Consequently, it is crucial to carefully consider the trade-off between accuracy and hardware efficiency when quantizing a model for edge devices. Quality of quantization is also strongly correlated to the network architecture and the target task. [108] shows that compact models are more difficult to quantize. Besides, compared to image classification, object detection is a more challenging task for ultra-low precision quantization because it requires accurate localization of specific

objects in an image. Even with quantization-aware fine-tuning, quantizing the detection models with naive quantization schemes can cause around 10% AP degradation on the COCO dataset [119]. This work takes advantage of mixed-precision quantization, where we have 4-bit for weights and 8-bit for activations. This can significantly reduce the accuracy degradation since activations are more sensitive compared to weights in object detectors.

3.8 CoDeNet Deformable Operation Co-design

Although deformable convolution augments the neural network design with input-adaptive sampling, it is challenging to provide efficient support for the operation in its original form on hardware accelerators due to the following reasons:

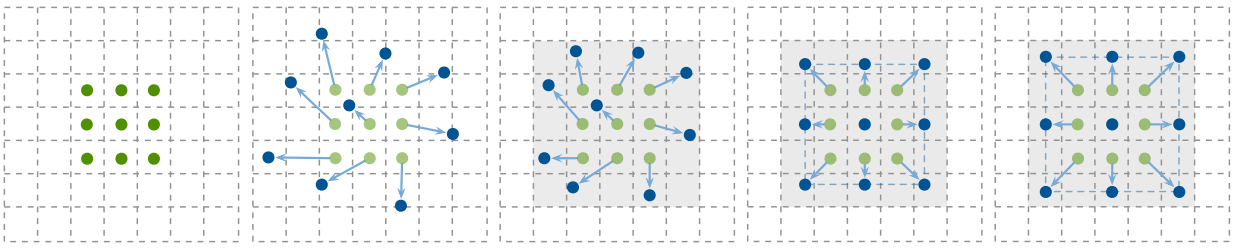
1. the limited reuse of input features
2. the irregular input-dependent memory access patterns
3. the computation overhead from the bilinear interpolation
4. the memory overhead of the deformable offsets

In this work, we perform a series of modifications to deformable convolution to enable more data reuse and a higher degree of parallelism for FPGA acceleration. A comprehensive ablation study is done to demonstrate the impact of each algorithmic modification on accuracy. We perform our study with standard object detection benchmarks, VOC, and COCO. We then design a specialized hardware engine optimized for each algorithmic modification on FPGA and show the performance improvement on FPGA from each modification. The accuracy and hardware efficiency tradeoff is studied for each modification we propose.

We will be using the following notations in the paper: n - batch size, h - height, w - width, ic - input channel size, oc - output channel size, k - kernel size, Δp - offsets.

3.8.1 Algorithm Modifications

We choose average precision (AP) as the main metric for benchmarking object detection performance on VOC and COCO datasets. ShuffleNet V2 [128] is used as the feature extractor in all experiments. As for decoder, we follow the practice of CenterNet [236] and use the stack of deformable convolution, nearest $2\times$ upsample, and ReLU activation layers. Table 3.9 lists the modifications we make to the original deformable convolution as well as a comparison among deformable convolutions of different forms and regular convolutions with varying sizes of the kernel. From the comparison, we see that the original deformable convolution achieves higher accuracy on Pascal VOC compared to convolution with 9×9 kernel (42.9 vs. 42.3) while requiring $\frac{9\times9}{3\times3} = 9\times$ fewer MACs and weight parameters. Here we discuss how we further improve the efficiency of deformable convolution for hardware step-by-step.



(a) normal (b) deform (c) bound (d) square (e) round

Figure 3.13: Major algorithm modifications for deformable convolution operational co-design. (a) is the default 3×3 convolutional filter. (b) is the original deformable convolution with unconstrained non-integer offsets. (c) sets an upper bound to the offsets. (d) limits the geometry to a square shape. (e) shows that the predicted offsets are rounded to integers.

Table 3.9: Ablation study of operation choices for object detection on VOC and COCO. The top half shows the baselines with various kernel sizes, from 3×3 to 9×9. The bottom half shows the comparison of different designs for deformable convolution.

Operation	Depthwise	Bound	Square	VOC			COCO					
				AP	AP50	AP75	AP	AP50	AP75	APs	APm	APl
3×3				39.2	60.8	41.2	21.4	36.5	21.5	7.3	24.1	33.0
3×3	✓			39.1	60.9	40.9	19.8	34.3	19.7	6.3	22.6	31.5
5×5	✓			40.6	62.4	42.6	21.3	36.4	21.3	6.7	23.7	34.2
7×7	✓			41.9	63.8	43.8	21.7	37.2	21.5	6.9	24.0	35.2
9×9	✓			42.3	64.8	44.3	22.2	37.8	22.1	7.0	24.3	35.4
deform	✓			42.9	64.4	45.7	23.0	38.4	23.3	6.9	24.4	37.8
deform	✓	✓		41.0	63.0	42.9	21.3	36.4	21.1	7.2	23.6	34.4
deform	✓	✓	✓	41.1	63.1	43.7	21.5	36.8	21.5	6.5	23.7	34.8

Depthwise Convolution We first replace the full 3×3 deformable convolutions with 3×3 depthwise deformable convolutions and 1×1 convolutions, similar to the depthwise separable convolution practice in Xception [39]. Such modification makes the whole network more uniform and smaller, so the weights of the deformable convolution can be all buffered on-chip for maximal reuse.

Bounded Range Our next algorithmic modification to facilitate efficient hardware acceleration is to restrict the offsets to a positive range. Such constraint limits the size of the working set of feature maps so that a pre-defined fixed-size buffer can be added to the hardware, in order to further exploit the temporal and spatial locality of the inputs. Assume a uniform distribution for the generated offsets in a 3×3 convolution kernel with stride 1, each pixel is expected to be used nine times. If all inputs within the range can be stored in the buffer, all except the first access to the same address will be from on-chip memory with $1 \sim 3$ cycle latency. We impose this constraint during training by adding a *clipping* operation after the offset generation layer to truncate offsets that are smaller than 0 or larger than N,

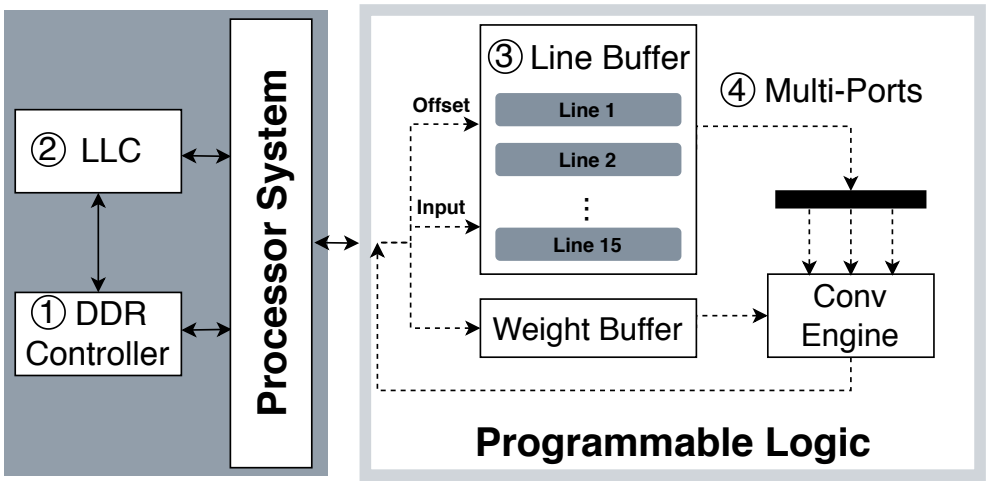


Figure 3.14: Hardware engine for deformable convolution.

so all offsets $\Delta p_x, \Delta p_y \in [0, N]$. Table 3.9 shows that setting the bound N to 7 results in 1.9 and 1.7 AP degradation on VOC and COCO respectively.

Square Shape Another obstacle to efficiently supporting the deformable convolution is its irregular data access patterns, which leads to serialized memory accesses to multi-banked on-chip memory. To address this issue, we further constrain the offsets to be on the edges of a square. Instead of using $3 \times 3 \times 2 = 18$ numbers to represent the Δp_x and Δp_y offsets for all nine samples, only one number Δp_d, representing the distance from the center to the sides of the square needs to be learned. This is similar to a dilated convolution with spatial-variant adaptive dilation factors. Adding this modification leads to a 0.1 and 0.2 AP increase on VOC and COCO.

Rounded Offsets In the original deformable design, the generated offsets are typically fractional, and a bilinear interpolation needs to be performed to produce the target sampling value. Bilinear interpolation calculates a weighted average of the neighboring pixels for a fractional offset based on its distance to the neighboring pixels. It introduces at least six multiplications to the sampling process of each input, which is a significant increase ($6 \times h \times w \times ic$) to the total FLOPs. We thus round the offsets to be integers during inference to reduce the total computation. The dynamically generated offsets are thus rounded to integers. In practice, we round the generated offset during the quantization step.

As shown in Table 3.9, together with the modifications above, our co-designed deformable convolution achieves 41.1 and 21.5 AP on VOC and COCO, respectively, which is 1.8 and 1.5 lower than the original depthwise deformable convolution. Note that the accuracy of the modified deformable convolution still achieves higher accuracy compared to the large 5×5 kernel, while requiring $\frac{3\times3}{5\times5} = 36\%$ fewer MACs and parameters.

Table 3.10: Co-designed hardware performance comparison. The top half shows the performance of codesigned hardware corresponding to each algorithmic changes to the default 3×3 convolution. The bottom half shows the results for the depthwise 3×3 convolution.

Operation	Deform	Bound	Square	Without LLC		With LLC	
				Latency (ms)	GOPs	Latency (ms)	GOPs
				43.1	112.0	41.6	116.2
default	✓			59.0	81.8	42.7	113.1
3×3 conv	✓	✓		43.4	111.5	41.8	115.5
	✓	✓	✓	43.4	111.5	41.8	115.6
				1.9	9.7	2.0	9.6
depthwise	✓			20.5	0.9	17.8	1.1
3×3 conv	✓	✓		3.0	6.2	3.4	5.5
	✓	✓	✓	2.1	9.2	2.3	8.2

3.8.2 Hardware Optimizations

Many hardware optimization opportunities are exposed after we perform the modifications as mentioned above to deformable convolution. We implement a hardware deformable convolution engine on FPGA SoC as shown in Figure 3.14 and tailor the hardware engine to each algorithm modification. The experiments are run on the Ultra96 board featuring a Xilinx Zynq XCZU3EG UltraScale+ MPSoC platform. The accelerator logic accesses the 1MB 16-way set-associative LLC through the Accelerator Coherency Port (ACP). The data cache uses a pseudo-random replacement policy. Table 3.10 lists the speed and throughput performance for different customized hardware running a kernel of size $h = 64, w = 64, k = 256, c = 256$. In all experiments, we round the dynamically generated offsets to integers. We use $8 \times 8 \times 9$ Multiply-Accumulate (MAC) units in the 3×3 convolution engine for all full convolution experiments and 16×9 MACs for depthwise convolution experiments.

Baseline The baseline hardware implementation for the original 3×3 deformable convolution directly accesses the DRAM without going through any cache or buffering. In Figure 3.14, the baseline implementation directly accesses the input and output data through HP ports and ① DDR controller. The input addresses are first calculated from the offsets loaded from DRAM. The 3×3 *Deform M2S* engine then fetches and packs the inputs into parallel data streams to feed into the MAC units in the 3×3 *Conv* engine. This baseline design resembles accelerator designs with only a scratchpad memory that cannot leverage the temporal locality of the dynamically loaded inputs for deformable convolution.

Caching One hardware optimization to leverage the temporal and spatial locality of the nonuniform input accesses is to add a cache to the accelerator system. As shown in Figure 3.14, we load the inputs from ② LLC through the ACP port in this implementation to reduce the memory access latency of the cached values. Since the inputs are sampled from offsets without specific patterns in the original deformable convolution, the cache provides

adequate support to buffer inputs that might be reused in the near future. As shown in Table 3.10, adding LLC results in 27.6% and 13.2% reduction in latency for the original full and depthwise deformable convolution, respectively.

Buffering With the bounded range modification to the algorithm, we are able to use the on-chip memory to buffer all possible inputs. Similar to a line-buffer design for the original 3×3 convolution that stores two lines of inputs to exploit all input locality, we store $2N$ lines of inputs so that it is sufficient to buffer all possible inputs for reuse. This implementation includes the ③ Line Buffer in Figure 3.14. With the effective buffering strategy, we can see in Table 3.10 that the latency of a bounded deformable is reduced by 26.4% and 85.3% for full and depthwise convolution, respectively, in a system without LLC. In a system with LLC, the reduction is 2.1% and 80.9%, respectively. The depthwise deformable convolution benefits more from adding the buffer as it is a more memory-bound operation. The compute-to-communication ratio for its input is *oc* times lower than the full convolution.

Parallel Ports The algorithm change to enforce a square-shape sampling pattern not only reduces the bandwidth requirements for loading the input indices in hardware, but also helps to improve the on-chip memory bandwidth. With a non-predictable memory access pattern to the on-chip memory, only one input can be loaded from the buffer at each cycle if all sampled inputs are store in the same line buffer. By constraining the shape of deformable convolution to a square with variable dilation, we are guaranteed to have three different line buffers storing three sampled points. We can thus have three parallel ports (④ Multi-ports in Figure 3.14) accessing different line buffers concurrently. This co-optimization improves the on-chip memory bandwidth and leads to another $\sim 30\%$ reduction in latency for depthwise deformable convolution.

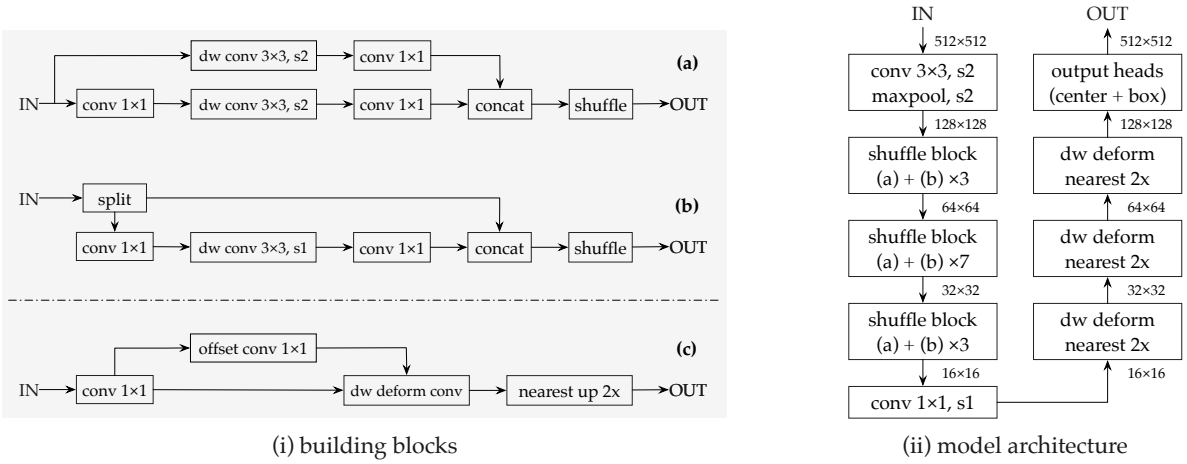
With the co-design methodology, our final result shows a 1.36× and 9.76× speedup, respectively, for the full and depthwise deformable convolution on the embedded FPGA accelerator. These optimizations can also be beneficial to other hardware with line buffer and parallel ports support.

3.9 CoDeNet Detection System Co-Design

In addition to the deformable convolution operation, the design of feature extractor, detection heads, and quantization strategy also significantly impact our detection system's accuracy and efficiency. In this section, we introduce an efficient detector and a specialized FPGA accelerator design to support it in CoDeNet.

3.9.1 CoDeNet Neural Network Design

To exploit the full potential of hardware acceleration, we carefully select and integrate the operations and building blocks in CoDeNet. We devise CoDeNet to have the following embedded hardware compatible properties compared to other off-the-shelf network designs: 1) more uniform operation types to reduce the control complexity in the accelerator and to



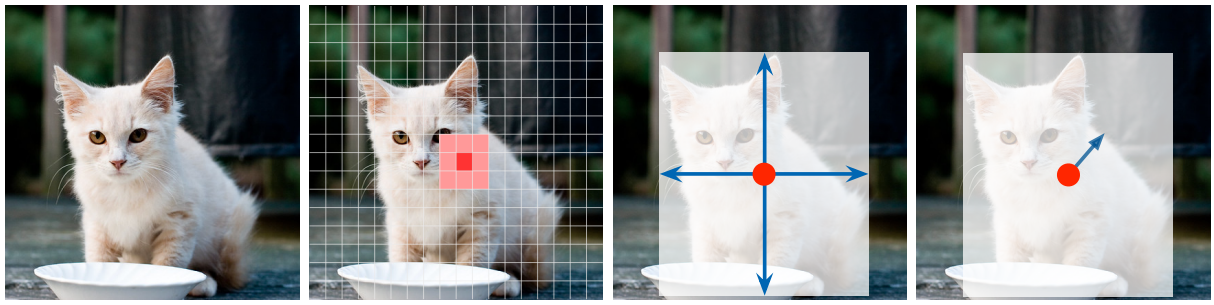
(i) building blocks (ii) model architecture

Figure 3.15: The architecture diagrams of our building blocks and model architecture.See section 3.9.1 for more details.

increase the accelerator utilization, 2) less computation to lower the overall latency to run on the embedded accelerator with limited compute capability, 3) smaller weights and inputs to be buffered on-chip for maximal reuse on the accelerator. Figure 3.15 shows the basic building blocks as well as the overall network architecture of CoDeNet.

Building Blocks and Feature Extractor The shaded part of Figure 3.15 shows the basic building blocks of CoDeNet. Building block (a) is used to down-sample the input images. A 3×3 depthwise convolution block with stride 2 is added to both of its branches together with 1×1 convolution to aggregate information across the channel dimension. Building block (b) splits the input features into two streams across the channel dimension. One branch is directly fed to the concatenation. The other streams through a sub-block of 1×1, 3×3 depthwise, and 1×1 convolution. This technique is referred to as identity mapping [79], which is commonly used to address the vanishing gradient problem during deep neural network training. Building blocks (a) and (b) together form a shuffle block in the ShuffleNetV2 feature extractor, as shown in the left branch of the overall architecture in Figure 3.15. We choose ShuffleNetV2 as it is one of the state-of-the-art efficient network design. ShuffleNetV2 1x configuration only requires 2.3M parameters (4.8× smaller than ResNet-18 [78]) and 146M FLOPs of compute with resolution 224 × 224 (12.3x smaller than ResNet-18). Its top-1 accuracy is 69.4% on ImageNet (0.36% lower than ResNet-18).

The deformable operation is used in building block (c). Building block (c) is used for upsampling the backbone features. The first 1×1 convolution is designed to map input channels to output channels. The following 3×3 depthwise deformable convolution samples the previous feature map, according to the offsets generated by 1×1 convolution. After that, a 2× upsampling layer, operated by a nearest neighbor kernel, is utilized to interpolate the higher resolution features. Note that, aside from the first layer, we only use 1×1 convolution and 3×3 depthwise (deformable) convolution in our build blocks. This way, the building



(a) image (b) center heatmap (c) width & height (d) local shift

Figure 3.16: The output heads of CenterNet for object detection. See section 3.9.1 for more details.

blocks of the whole network become more uniform and simple to support with specialized hardware.

Detection Heads As mentioned in Section 3.7.1, we use the anchor-free CenterNet [236] method to directly predict a gaussian distribution for object keypoints over the 2D space for object detection. Given an image $I \in \mathbb{R}^{W \times H \times 3}$, our feature extractor generates the final feature map $F \in \mathbb{R}^{\frac{W}{R} \times \frac{H}{R} \times D}$, where R is the output stride and D is the feature dimension. We set $R = 4$ and $D = 64$ for all the experiments. As illustrated in Figure 3.16, the outputs include:

1. the keypoint heatmap $\hat{Y} \in [0,1]^{\frac{W}{R} \times \frac{H}{R} \times C}$
2. the object size $\hat{S} \in \mathbb{R}^{\frac{W}{R} \times \frac{H}{R} \times 2}$
3. the local offset $\hat{O} \in \mathbb{R}^{\frac{W}{R} \times \frac{H}{R} \times 2}$

Here C is pre-defined as 20 and 80 for VOC and COCO, respectively. In order to reduce the computation, we follow the class-agnostic practice, using the single size and offset predictions for all categories. To construct bounding boxes from the keypoint prediction, we first collect the peaks in keypoint heatmap $\hat{Y}$ for each category independently. Then we only keep the top 100 responses which are greater than its eight-connected neighborhood. Specifically, we use the keypoint values $\hat{Y}_{x_i y_i c}$ as the confidence measure of the i-th object for category c. The corresponding bounding box is decoded as

$(\hat{x}_i + \delta\hat{x}_i - \hat{w}_i/2, \quad \hat{y}_i + \delta\hat{y}_i - \hat{h}_i/2, \hat{x}_i + \delta\hat{x}_i + \hat{w}_i/2, \quad \hat{y}_i + \delta\hat{y}_i + \hat{h}_i/2)$,

where $(\delta\hat{x}_i, \delta\hat{y}_i) = \hat{O}_{\hat{x}_i \hat{y}_i}$ is the offset prediction and $(\hat{w}_i, \hat{h}_i) = \hat{S}_{\hat{x}_i \hat{y}_i}$ is the size prediction.

Quantization Quantization is a crucial step towards the efficient deployment of the GPU pre-trained model on FPGA accelerators. Although many previous works treat quantization as a separate process outside the algorithm-hardware co-design loop, we note that quantization performance greatly depends on the network architecture. For example, the residual connection will enlarge the activation range of specific layers, making a uniform quantization setting sub-optimal. And it requires a special design for addition in int32 format; otherwise,

extra steps of quantization are needed to support the low-precision addition. With this prior knowledge, we use concatenation instead of residual connection throughout CoDeNet, and we do not use techniques such as layer aggregation [221] to achieve a simpler hardware design.

We adopt a symmetric uniform quantizer shown as follows:

$$X' = \text{clamp}(X, -t, t), \tag{3.4}$$

$$X^I = \lfloor \frac{X'}{\Delta} \rceil, \text{ where } \Delta = \frac{t}{2^{k-1}-1}, \tag{3.5}$$

$$Q(X) = \Delta X^I, \tag{3.6}$$

where Q stands for quantization operator, X is a floating-point input tensor (activations or weights), $\lfloor \cdot \rceil$ is the round operator, Δ is the quantization step (the distance between adjacent quantized points), X^I is the integer representation of X, and k is the quantization precision for a specific layer. Here, threshold value t determines the quantization range of the floating-point tensor, and the clamp function sets all elements smaller than $-t$ to $-t$, and elements larger than t to t. It should be noted that the threshold value t can be smaller than max or $|min|$ in order to get rid of outliers and better represent the majority of a specific tensor. In order to achieve better AP, we perform 4-bit channel-wise quantization [108] for weights. Meanwhile, to ease the hardware design and accelerate the inference, we choose a symmetric uniform quantizer rather than non-uniform quantizer, and we use 8-bit layer-wise quantization for activations. During quantization-aware fine-tuning, we use Straight-Through Estimator (STE) [16] to achieve the backpropagation of gradients through the discrete operation of quantization.

For the deformable convolution, quantization comprises two parts: 1) quantize the corresponding weights and activations, and 2) round and bound the sampling offsets of the deformable convolution. Compared to the standard convolution, the variable offsets will not significantly change the network's sensitivity or the allowable quantization bit-width. Regarding the original fractional offsets, we bound and round them to be integers within the range $[-8, 7]$. This modification eliminates the need for bilinear interpolation and results in a 1.9 AP drop on VOC as shown in Table 3.9.

3.9.2 Dataflow Accelerator

We develop a specialized accelerator to support the aforementioned CoDeNet design on an FPGA SoC. As shown in Figure 3.17, the FPGA SoC includes the programmable logic (PL), memory interfaces, a quad-core ARM Cortex-A53 application processor with 1MB LLC, etc. Our accelerator on the PL side communicates to the processor through an AXI system bus. The High Performance (HP) and Accelerator Coherency Port (ACP) interfaces on the AXI bus allow the accelerator to directly access the DRAM or perform cache-coherent accesses to the LLC and DRAM. The processor provides software support to invoke the accelerator and run functions not implemented on the accelerator.

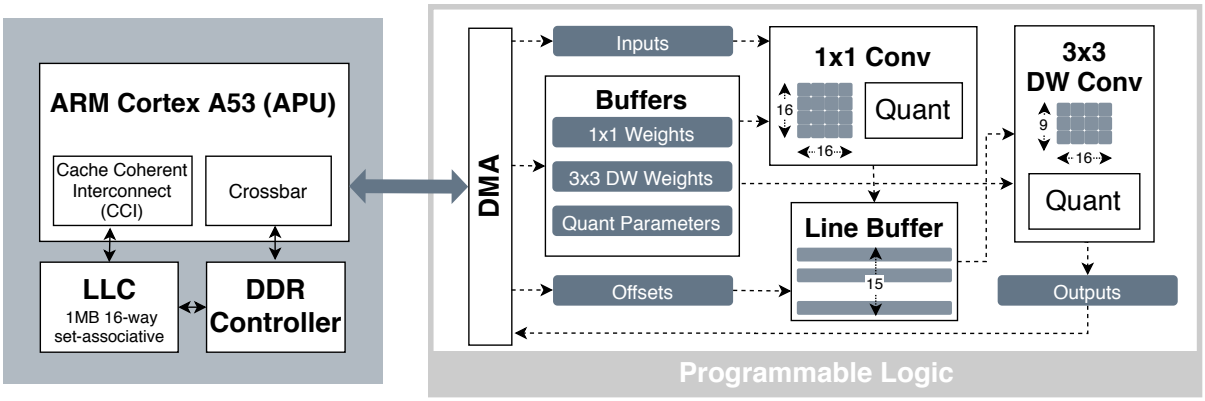


Figure 3.17: Architectural diagram of the FPGA accelerator.

With our co-design methodology, we are able to reduce the types of operations to support in the accelerator. Excluding the first layer for the full 3×3 convolution, CoDeNet only consists of the following operations: (i) 1×1 convolution, (ii) 3×3 depthwise (deformable) convolution, (iii) quantization, (iv) split, shuffle and concatenation.

This helps us simplify the complexity of the control logic and thus saves more FPGA resources for the actual computation. We partition the CoDeNet workload so that the frequently-called compute-intensive operations are offloaded to the FPGA accelerator while the other operations are run by software on the processor. The operations we choose to accelerate are 1×1 convolution, 3×3 depthwise (deformable) convolution, and quantization, with the other operations offloaded to the processor.

To leverage both the data-level and the task-level parallelism, we devise a spatial dataflow accelerator engine to execute a subgraph of the CoDeNet at a time and store the intermediate outputs to the DRAM. In the dataflow engine, the execution of compute units is determined by the arrival of the data and thus further reduces the overhead from the control logic. As illustrated in the architectural diagram in Figure 3.17, our accelerator executes 1×1 convolution with quantization and 3×3 depthwise (deformable) convolution with quantization in order. We implement the accelerator with Vivado HLS and its dataflow template. All functional engines are connected to each other through data FIFOs. Extra bypass signals can be asserted if the user would like to bypass either of the main computation blocks. By co-designing the network to use operations with fewer weight parameters, such as depthwise convolution, we are able to buffer the weights for all operations in the on-chip memory and enable the maximal reuse of the weights once they are on-chip. We also add a line buffer for the 3×3 depthwise (deformable) convolution to maximize the reuse of inputs on-chip. This optimization is enabled by the operation co-design discussed in Section 3.8.2. The line buffer stores 15 rows of the input image. The size of this buffer is larger than $15 \times w \times ic$ of any layers in the CoDeNet design. Our input tensors are laid out in the NHWC manner, allowing the data along the channel dimension C to be stored in contiguous memory blocks.

Table 3.11: Quantized CoDeNet on VOC object detection.

Detector	Resolution	DownSample	Weights	Activations	Model Size	MACs	AP50
Tiny-YOLO	416×416	MaxPool	32-bit	32-bit	60.5 MB	3.49 G	57.1
CoDeNet1× (config a)	256×256	Stride4	32-bit	32-bit	6.06 MB	0.29 G	53.0
			4-bit	8-bit	0.76 MB	0.29 G	51.1
CoDeNet1× (config b)	256×256	Stride2+MaxPool	32-bit	32-bit	6.06 MB	0.29 G	57.5
			4-bit	8-bit	0.76 MB	0.29 G	55.1
CoDeNet1× (config c)	512×512	Stride4	32-bit	32-bit	6.06 MB	1.14 G	64.6
			4-bit	8-bit	0.76 MB	1.14 G	61.7
CoDeNet2× (config d)	512×512	Stride4	32-bit	32-bit	23.2 MB	3.54 G	69.6
			4-bit	8-bit	2.90 MB	3.54 G	67.1
CoDeNet2× (config e)	512×512	Stride2+MaxPool	32-bit	32-bit	23.2 MB	3.58 G	72.4
			4-bit	8-bit	2.90 MB	3.58 G	69.7

1×1 **convolution** The compute engine for the 1×1 convolution is composed of 16×16 multiply-accumulate (MAC) units. At each round of the run, the engine takes 16 inputs along its channel dimension and broadcasts each of them to 16 MAC units. Meanwhile, it unicasts 16×16 weights for 16 input channels and 16 output channels to their corresponding MAC unit. There are 16 reduction trees of size 16 connected with the MAC units to generate 16 partial sums of the products. The partial sums are stored on the output registers and are accumulated across each round of the run. Every time the engine finishes the reduction along the input channel dimension, it feeds the values of the output registers to the output FIFO and resets their values to zero.

3×3 **depthwise (deformable) convolution** This engine directly reads 16 sampled 3×3 inputs from the line buffer design and multiplies them by 3×3 weights from 16 corresponding channels. Then it computes the outputs with 16 reduction trees to accumulate the partial sums along 3×3 spatial dimension. Both the original and the deformable depthwise convolutions can be run on this engine. The original depthwise operation is realized by hardcoding the offset displacement to be 1.

Quantization To convert the output from the 16-bit sum to 8-bit inputs, we add a quantization unit at the end of each compute engine. The quantization unit multiplies each output with a scale, and then adds a bias to it. It returns the lower 8 bits of the result as the quantized value. The parameters, such as the scale and bias for each channel, are preloaded to the on-chip buffer to save the memory access time. Note that we also merge the batch normalization and ReLU in this compute unit. We follow the practice introduced in [95] to perform integer inference for our quantized model.

Our accelerator design can execute $16 \times 1 \times 250 \times 2 = 128$ GOPs for 1×1 convolution and $9 \times 16 \times 250 \times 2 = 72$ GOPs for 3×3 depthwise convolution simultaneously. On our target FPGA with 6GB/s DDR bandwidth, we can load 4 Giga pairs of 8-bit inputs and 4-bit weights per second. The arithmetic intensity required to reach the compute bound, is $128/4 = 32$ OPs/pair for 1×1 convolution and $72/4 = 18$ OPs/pair for 3×3 depthwise convolution. Our buffering strategy allows us to reach the compute bound through the reuse of weights and the activations.

Table 3.12: Quantized CoDeNet on COCO object detection.

Detector	Weights	Model Size	MACs	AP	AP50	AP75	APs	APm	APl
CoDeNet1×	32-bit	6.07MB	1.24G	22.2	38.3	22.4	5.6	22.3	38.0
	4-bit	0.76MB	1.24G	18.8	33.9	18.7	4.6	19.2	32.2
CoDeNet2×	32-bit	23.4MB	4.41G	26.1	43.3	26.8	7.0	27.9	43.5
	4-bit	2.93MB	4.41G	21.0	36.7	21.0	5.8	22.5	35.7

3.10 CoDeNet Experimental Results

We implement CoDeNet in PyTorch, train it with a pretrained ShuffleNetV2 backbone, and quantize the network to use 8-bit activations and 4-bit weights. We devise several configurations of CoDeNet to facilitate the latency-accuracy tradeoffs for our final object detection solution on the embedded FPGAs. Different configurations of the CoDeNet are listed in Table 3.11 and 3.12 showing the accuracies for object detection on Pascal VOC and Microsoft COCO 2017 dataset.

In Table 3.11, we show different configurations of CoDeNet with an accuracy-efficiency trade-off. *config c, d* and *e* use image size 512×512, which is the default resolution of CenterNet. Compared to Tiny-YOLO, our *config c* model is 10× smaller without quantization and 79.6× smaller with quantization, while achieving higher accuracy. In addition, the total MACs count of our compact design is 3.1× smaller than Tiny-YOLO. It can be seen that quantizing the model to 4–8 bits causes a minor accuracy drop, but can significantly reduce the model size ($> 8\times$). To further save the MACs, we reduce the resolution to be 256×256, corresponding to *config a*, where we can still get 53 AP50 with about 1/4 total MACs compared with *config c*. Moreover, we found the downsampling strategy of the first layer play an important role. A larger stride for the first layer can benefit the speed (shown later in Table 3.13), but a smaller stride can process more information and therefore improve accuracy (corresponding to *config b*). For scenarios that require more accurate detectors, we expand the channel size of *config c* (CoDeNet1×) by a factor of 2, which gives us config d that can achieve 69.6 AP50. After quantization, *config d* has a 67.1 AP50 with comparable MACs but 21× smaller memory size compared to Tiny-YOLO. By doubling the channel size (CoDeNet2×) and using a smaller stride, we have *config e*, which can achieve the highest 72.4 AP50 among all the configurations.

Table 3.12 shows the accuracy of CoDeNets on the Microsoft COCO 2017 dataset. Microsoft COCO is a more challenging dataset compared to Pascal VOC. COCO has 80 categories, and Pascal VOC has 20. Our results are obtained with default 512×512 resolution, with stride 2convolution and maxpooling as the downsampling strategy. Besides AP50, COCO primarily uses AP as the evaluation metric, which is the average among AP[0.5:0.95] (namely AP50, AP55, ..., AP95). As shown in the table, CoDeNet1× can achieve 22.2 AP with model size 6.07 MB. Applying quantization will cause a minor accuracy degradation, but can get an 8× smaller model. The same trend holds for CoDeNet2× where our model can get 26.1 and

Table 3.13: Performance comparison with prior works.

	Platform	Input Resolution	Framerate (fps)	Test Dataset	Precision	Accuracy
DNN1 [76]	Pynq-Z1	-	17.4		a8	IoU(68.8)
DNN3 [76]	Pynq-Z1	-	29.7	DJI-UAV	a16	IoU(59.3)
Skynet [229]	Ultra96	160 × 360	25.5		w11a9	IoU(71.6)
AP2D [120]	Ultra96	224 × 224	30.5	AD2P	w(1-24)a3	IoU(55)
Finn-R [19] [154]	Ultra96	-	16	VOC07	w1a3	AP50(50.1)
Tiny-Yolo-v2 [56]	Zynq-706 XC7Z045	224 × 224	43.1		w16a16	AP50(48.5)
Ours (config a)		256 × 256	32.2			AP50(51.1)
Ours (config b)		256 × 256	26.9			AP50(55.1)
Ours (config c)	Ultra96	512 × 512	9.3	VOC07	w4a8	AP50(61.7)
Ours (config d)		512 × 512	5.2			AP50(67.1)
Ours (config e)		512 × 512	4.6			AP50(69.7)

Table 3.14: FPGA resource utilization.

LUT	FF	BRAM	DSP
34144 (48.4%)	41827 (29.6%)	216 (100%)	360 (100%)

21.0 AP, with and without quantization, respectively.

We evaluate our accelerator customized for each CoDeNet configurations on the Ultra96 development board with Xilinx Zynq XCZU3EG UltraScale+ MPSoC device. Our accelerator design runs at 250 MHz after synthesis, and place and route. Table 3.14 shows the overall resource utilization of our implementation. We observe a 100% utilization of both DSPs and BRAMs. Most DSPs are mapped to the 4-8 bit MAC units, and BRAMs are mainly used for the line buffer design. Our Power measurements are obtained via a power monitor. We measured 4.3W on the Ultra96 power supply line with no workload running on the programming logic side and 5.6W power when running our network. On CoDeNet *config a*, our accelerator achieves 5.75 fps / W in terms of power efficiency.

We provide a Pareto curve in Figure 3.18 showing the latency-accuracy tradeoff for various CoDeNet design points with acceleration. Configuration a and b in this curve are trained and inferenced with images of size 256×256 instead of the original size 512×512. The smaller input image size leads to $\sim 4\times$ reduction in MACs. In configuration a, c and d, the stride of the first layer is increased from 2 to 4, which greatly reduces the first layer runtime on the processor. In configuration d and e, we use the CoDeNet $2\times$ model, where the channel size is doubled in the network, to boost the accuracy. The latency evaluation on our accelerator is done with a batch size equal to 1 without any runtime parallelization. We run the first layer of the network on the processor for all configurations.

A comparison of our solutions against previous works is shown in Table 3.13. We found that very few prior works on embedded FPGAs attempt to target the standard dataset like VOC or COCO for object detection, primarily due to the challenges from limited hardware resources and inefficient model design. Two state-of-the-art FPGA solutions that meet the real-time requirement in the DAC-UAV competition target the DJI-UAV dataset for drone

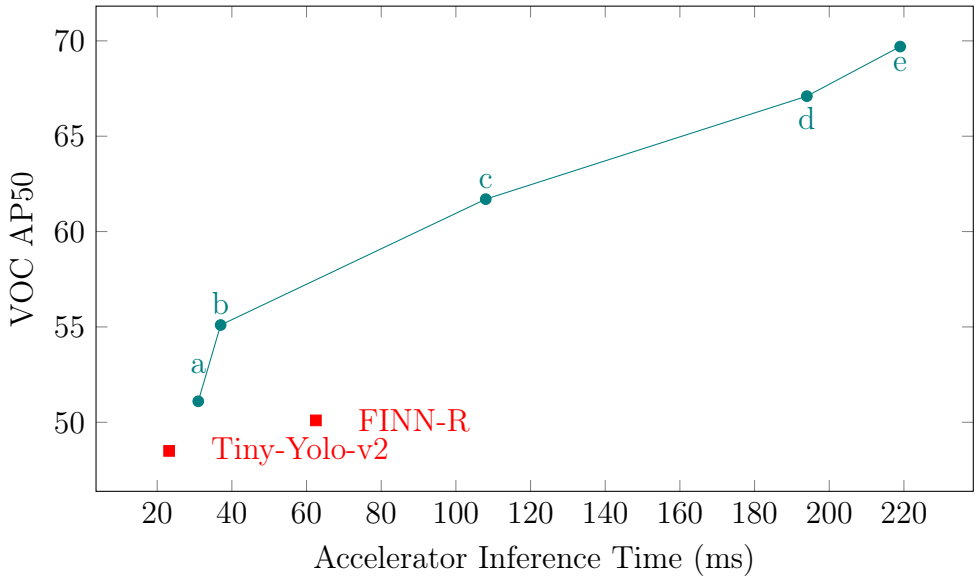


Figure 3.18: Latency-accuracy trade-off on VOC.

image detection. However, object detection on DJI-UAV is a less generic and less challenging task than object detection on VOC or COCO. The images in DJI-UAV dataset are taken from the top-down view. They typically contain very few overlapped objects. In addition, the DJI-UAV dataset is designed for single-object detection whereas VOC and COCO can be used for multi-object detection. Hence, in this work, we target VOC and COCO to provide a more general solution for multi-object detection and for images taken from the most common first-person view.

As shown in Figure 3.18 and Table 3.13, compared to the results from FINN-R [19] [154], the state-of-the-art embedded FPGA accelerator design targeting VOC, our configuration *a* and *b* (with single-batch inference latency of 31ms and 37ms respectively) achieve both higher accuracy, higher framerate, and lower latency. Another state-of-the-art work Tiny-Yolo-v2 [56] attains low latency but with lower accuracy. It also runs on a different FPGA platform.

3.11 Conclusion

In Synetgy, we adopt an algorithm-hardware co-design approach to develop a ConvNet accelerator and a novel ConvNet for image classification. Based on ShuffleNetV2, we optimize the network's operators by replacing all the 3×3 convolutions with shift operations and 1×1 convolutions. This allows us to build a compute unit exclusively customized for 1×1 convolutions for better efficiency. We quantize the network's weights to 4-bit and activations to 4-bit fixed-point numbers with less than 1% accuracy loss. These quantizations very well exploit the nature of FPGA hardware. As a result, DiracDeltaNet has a small parameter size,

low computational OPs, hardware-friendly skip connections, low precision, and simplified operators. These features allow us to implement highly customized and efficient accelerators on FPGA. We implement the network on Ultra96 Soc systems. The implementation only took two people one month using HLS tools. Our accelerator, Synetgy, achieves a top-5 accuracy of 88.1% on ImageNet, the highest among all the previously published embedded FPGA accelerators. It also reaches an inference speed of 66.3 FPS, surpassing prior works with similar accuracy by 11.6×.

In CoDeNEt, we evaluate algorithmic changes for deformable convolution with corresponding hardware optimizations and show a 1.36× and 9.76× speedup respectively for the full and depthwise deformable convolution on hardware with minor accuracy loss. We then **Co-De**sign a **Net**work **CoDeNet** with the modified deformable convolution for object detection and quantize the network to 4-bit weights and 8-bit activations. With our high-efficiency implementation, our solution reaches 26.9 frames per second with a tiny model size of 0.76 MB while achieving 61.7 AP50 on the standard object detection dataset, Pascal VOC. With our higher-accuracy implementation, our model gets to 67.1 AP50 on Pascal VOC with only 2.9 MB of parameters—20.9× smaller but 10% more accurate than Tiny-YOLO.

In both works, we performed detailed accuracy-efficiency trade-off studies for each hardware-friendly algorithmic modification with the goal of co-designing an efficient network and a real-time embedded accelerator optimizing for accuracy, speed, and energy efficiency. While there are many more opportunities for further optimization, we believe the work demonstrates the efficacy of our co-design methodology.

Finally, in our follow-up work called HAO [49], instead of manually exploring different co-design options, we further developed an automatic flow to perform design space search for both the algorithm design, hardware implementation, and quantization schemes.

Chapter 4

Scheduling and Hardware Co-design

Recent advances in Deep Neural Networks (DNNs) have led to active development of specialized DNN accelerators, many of which feature a large number of processing elements laid out spatially, together with a multi-level memory hierarchy and flexible interconnect. While DNN accelerators improve the peak throughput and data reuse opportunities, they also expose a large number of runtime parameters to the programmers who need to explicitly manage how computation is scheduled both *spatially* and *temporally*. In fact, different scheduling choices can lead to widely varying performance and efficiency differences, motivating the need for a fast and efficient search strategy to navigate the vast scheduling space.

In this chapter, we present CoSA, a constrained-optimization-based approach for scheduling DNN accelerators to address this challenge. Different from existing approaches that either rely on designers' heuristics or expensive iterative methods to prune the search space, the key idea of CoSA is to express the scheduling decisions as a constrained optimization problem that can be deterministically solved using advanced optimization techniques. CoSA leverages the regularities in DNN operators and hardware to formulate the DNN scheduling space into a mixed integer programming (MIP) problem with algorithmic and architectural constraints, where it can automatically generate a highly efficient schedule in a single pass.

4.1 Hardware-Aware Scheduling

Deep neural networks (DNNs) have gained major interest in recent years due to their robust ability to learn based on large amounts of data. DNN-based approaches have been applied to computer vision [78, 110, 164], machine translation [189, 198], audio synthesis [143], recommendation models [67, 139], autonomous driving [20] and many other fields. Motivated by the high computational requirements of DNNs, there have been exciting developments in both research and commercial spaces in building specialized DNN accelerators for both edge [30, 31, 53, 63, 147, 182, 186, 227] and cloud applications [6, 33, 58, 81, 98, 199].

State-of-the-art DNN accelerators typically incorporate large arrays of processing elements to boost parallelism, together with a deep multi-level memory hierarchy and a flexible

network-on-chip (NoC) to improve data reuse. While these architectural structures can improve the performance and energy efficiency of DNN execution, they also expose a large number of scheduling parameters to programmers who must decide when and where each piece of computation and data movement is mapped onto the accelerators both spatially and temporally. Here, we use *schedule* to describe how a DNN layer is partitioned spatially and temporally to execute on specialized accelerators. Given a target DNN layer and a specific hardware architecture, there could be millions, or even billions, of valid schedules with a wide range of performance and energy efficiency [146]. Considering the vast range of DNN layer dimensions and hardware architectures, there is a significant demand for a generalized framework to quickly produce efficient scheduling options for accelerators of varying hardware configurations.

Achieving high performance on a spatially distributed architecture requires several factors to be carefully considered, including tiling for good hardware utilization, pipelining data movement with compute, and maximizing data re-use. Previous scheduling frameworks have attempted to reflect these considerations by formulating an analytical cost model, pruning the scheduling space with known hardware constraints, and then exhaustively searching for the best candidate based on their cost models [28,45,146,216]. However, navigating the scheduling space in such a brute-force fashion can easily become intractable for larger DNN layers and more complex hardware architectures. Other notable efforts have employed feedback-driven approaches, such as black-box tuning, beam search, and other machine learning algorithms with iterative sampling [2,29,96]. However, these schedulers typically require massive training datasets and large-scale simulations to learn performance models, making it infeasible to extend them to other types of hardware accelerators, especially those still under development. Hence, there is a clear need for efficient scheduling mechanisms to *quickly* navigate the search space and produce *performant* scheduling options.

This chapter demonstrates CoSA, a constrained-optimization-based approach to schedule DNN accelerators. In contrast to prior work that either requires exhaustive brute-force-based or expensive feedback-driven approaches, CoSA expresses the DNN accelerator scheduling as a constrained-optimization problem that can be deterministically solved using today's mathematical optimization libraries in one pass. In particular, CoSA leverages the regularities in both DNN layers and spatial hardware accelerators where the algorithmic and hardware parameters can be clearly defined as scheduling constraints. Specifically, CoSA formulates the DNN scheduling problem as a prime-factor allocation problem that determines 1) tiling sizes for different memory levels, 2) relative loop ordering to exploit reuse, and 3) how computation should be executed spatially and temporally. CoSA constructs the scheduling constraints by exposing both the algorithmic behaviors, e.g., layer dimensions, and hardware parameters, e.g., memory and network hierarchies. Together with clearly defined and composable objective functions, CoSA can solve the DNN scheduling problem in one shot without expensive iterative search. Our evaluation demonstrates that CoSA-generated schedules outperform state-of-the-art approaches by 2.5× across different DNN network layers, while requiring 90× less scheduling time as it does not require iterative search.

In summary, this work makes the following contributions:

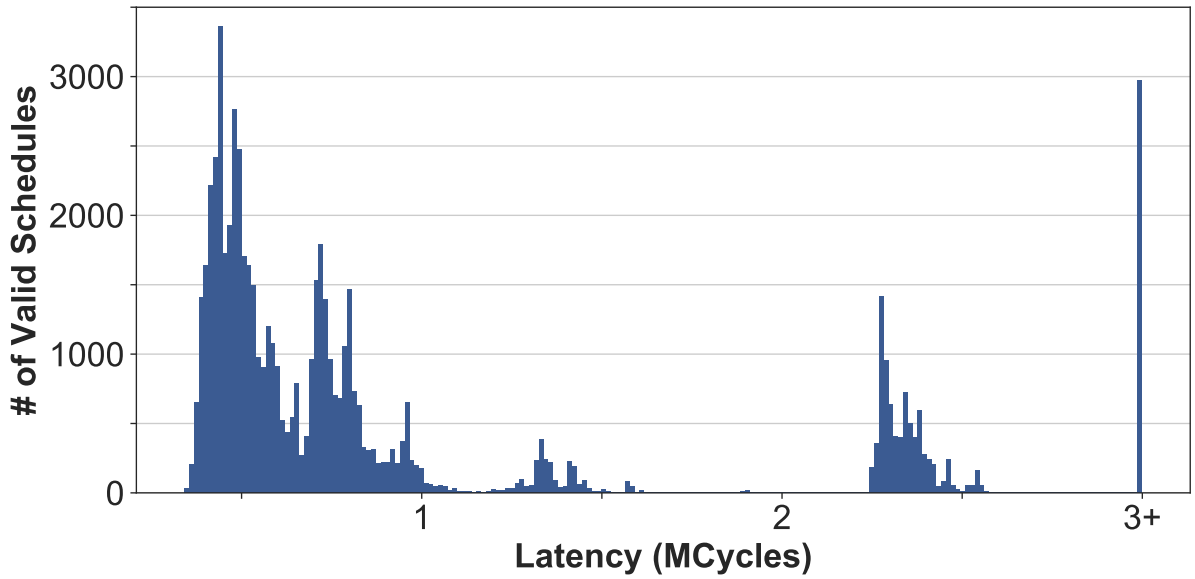


Figure 4.1: Execution latency histogram of 40K valid scheduling choices for a ResNet-50 layer on a spatial accelerator.

- We formulate DNN accelerator scheduling as a constrained-optimization problem that can be solved in a single pass. To the best of our knowledge, CoSA is the first constrained-optimization-based approach to tackle major DNN scheduling decisions in one shot.
- We take a communication-oriented approach in the CoSA formulation that highlights the importance of data transfer across different on-chip memories and exposes the cost through clearly defined objective functions.
- We demonstrate that CoSA can quickly generate high-performance schedules outperforming state-of-the-art approaches for different DNN layers across different hardware architectures.

4.2 Background and Motivation

In this section, we discuss the complexity of DNN scheduling space and the state-of-the-art schedulers to navigate the space.

4.2.1 DNN Scheduling Space

Scheduling is a crucial decision-making process for the compilers to effectively assign workload to compute resources. With the emergence of numerous DNN accelerators with diverse

architectures, there is a need for a fast, performant, and explainable approach to scheduling. Our work focuses on operator-level scheduling, which aims to optimize the performance of each operator, i.e. DNN layer, on specific hardware. Operator-level scheduling typically comprises three key loop optimizations: *loop tiling*, *loop permutation*, and *spatial mapping*. *Loop tiling* describes which loops are mapped to which memory hierarchy and the corresponding tile sizes. *Loop permutation* determines the relative order of the loops, while *spatial mapping* binds one or more loop dimensions to spatial hardware resources, such as parallel processing elements, instead of mapping them to temporal (i.e. sequential) execution. Each optimization can have a significant impact on the performance, and all three optimizations need to be considered together to achieve the best performance.

Consider scheduling a 3×3 convolution layer in ResNet50 [78] with 256 input and output channels, and an output dimension of 14×14, on an accelerator with five levels of memory. If we split each individual loop bound into its prime factors and assign each one to a memory level, we would have billions of schedules to consider. Among the randomly sampled schedules from all possible loop tilings, half of them fail to satisfy the buffer capacity constraints (e.g. a schedule is invalid if it requires a 4KB buffer, though the available buffer size is only 2KB.). Figure 4.1 shows the performance distribution of the valid schedules. We observe a wide performance difference among the valid schedules, with the best one outperforming the worst one by 7.2×. In addition, we observe clusters of schedules that have similar latencies in the Figure 4.1, revealing structure in the solution space.

4.2.2 State-of-the-art Schedulers

Given that the scheduling space for a DNN layer can have billions of valid schedules, finding a good schedule through exhaustive search can become an intractable problem. Table 4.1 shows some recent efforts to tackle this complexity.

4.2.2.1 Brute-force Approaches

Recent efforts combine exhaustive search with heuristics to manually prune the scheduling space [28, 45, 146, 194, 216]. To lower the cost of exhaustive search, schedulers in this category typically use a lightweight analytical model to estimate latency, throughput, and power consumption to compare all valid mappings of a given layer to find the best schedule. The disadvantages of this approach are two-fold. First, such a brute-force search tends to be exceedingly expensive for complex hardware architectures, making it infeasible to find a good schedule quickly. Second, the generated schedules often do not perform optimally since analytical models may fail to consider the communication latency across the spatial hardware.

4.2.2.2 Feedback-based Approaches

Other recent efforts use feedback-driven approaches along with machine learning or other statistical methods [2, 29, 80, 96, 101, 160] to improve the accuracy of the cost model and

Scheduler	Search Algorithm
Brute-force Approaches:	
Timeloop [146]	Brute-force & Random
dMazeRunner [45]	Brute-force
Triton [194]	Brute-force over powers of two
Interstellar [216]	Brute-force
Marvel [28]	Decoupled Brute-force
Feedback-based Approaches:	
AutoTVM [29]	ML-based Iteration
Halide [160]	Beamsearch [2], OpenTuner [7, 136]
FlexFlow [96]	MCMC
Gamma [101]	Genetic Algorithm
Constrained Optimization Approaches:	
Polly+Pluto [21, 22, 64]	
Tensor Comprehension [197]	Polyhedral Transformations
Tiramisu [11]	
CoSA	**Mixed Integer Programming (MIP)**

Table 4.1: State-of-the-art DNN accelerator schedulers.

search for the solution using black-box or gradient-based search. Although such approaches can potentially learn the distribution of the scheduling space, they typically require a large amount of training data due to their feedback-driven nature. As a result, these approaches are mainly applicable to post-silicon hardware where performing a large-scale measurement is possible but are not feasible for hardware under development.

4.2.2.3 Constrained-optimization Approaches

Constrained-optimization problems, in which objective functions are maximized or minimized subject to given sets of constraints, have demonstrated the ability to solve many complex large-scale problems in a reasonable time. Such methods have been widely used in architecture and systems research for instruction scheduling [36,140,141], high-level synthesis [42], memory partitioning [9,82] [41], algorithm selection [74,234], and program synthesis [5,14,149,150,185].

In particular, polyhedral transformation has leveraged constrained-optimization-based

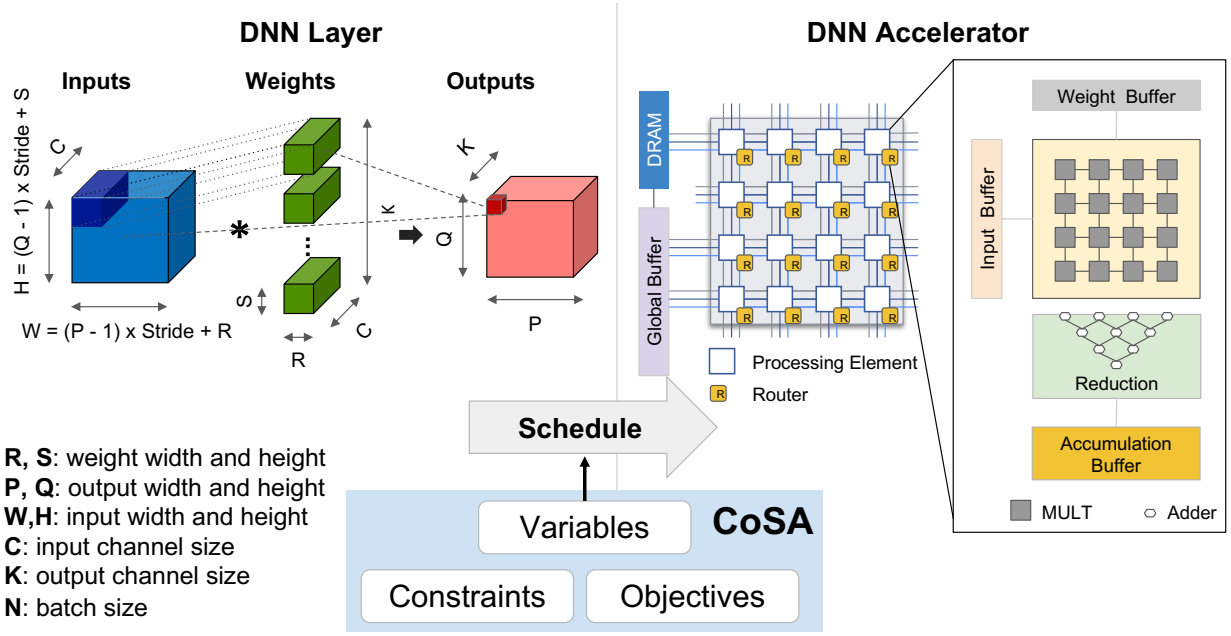


Figure 4.2: DNN scheduling problem formulation with CoSA. CoSA takes 1) DNN layer dimensions and 2) DNN accelerator parameters and expresses the scheduling problem into a constrained optimization problem to produce a performant schedule in one shot.

approach for auto-vectorization and loop tiling [1, 12, 22, 64, 107, 148]. Prior work targets general-purpose CPUs and GPUs that run with fine-grained instructions and hardware-managed cache, as opposed to the software-managed spatial accelerators that we target. In addition, existing polyhedral-based approaches [11, 12, 22] lack direct support for tile-size optimization. Instead, they take the tile size as input and apply a transformation based on the given tile size. Due to this limitation, the tile size decision cannot be co-optimized with other loop transformations, e.g. loop permutation, in one pass, leading to sub-optimal schedules.

To address the drawbacks of existing approaches and leverage the regularities from the DNN workloads and the accelerator design for optimization, CoSA employs constrained optimization to tackle the DNN scheduling problem in one pass. CoSA presents a unique domain-specific representation for DNN scheduling that better captures the utilization and communication cost and encodes different loop transformations, i.e., tiling size, loop permutation, and spatial mapping decisions, in one formulation. This unified representation enables us to solve for all three optimizations in one pass and produce efficient schedules for a complex accelerator system with a multi-level memory hierarchy.

```
//DRAM level
for q2 = [0 : 2) :
 // Global Buffer level
 for p2 = [0 : 7) :
  for q1 = [0 : 7) :
   for n0 = [0 : 3) :
    spatial_for r0 = [0 : 3) :
     spatial_for k1 = [0 : 2) :
      // Input Buffer level
      spatial_for k0 = [0 : 2) :
       // Weight Buffer level
       for c1 = [0 : 2) :
        for p1 = [0 : 2) :
         // Accumulation Buffer level
         for s0 = [0 : 3) :
          for p0 = [0 : 2) :
           spatial_for c0 = [0 : 8) :
            // Register
            for q0 = [0 : 2) :
```

Listing 4.1: An example schedule using the loop nest representation for a DNN layer of dimension $R = S = 3, P = Q = 28, C = 8, K = 4, N = 3$. Same variable prefix indicates tiles from the same problem dimension.

4.3 The CoSA Framework

To navigate the large scheduling space of DNN accelerators, we develop CoSA, a constrained-optimization-based DNN scheduler to automatically generate high-performance schedules for spatially distributed accelerators. CoSA not only deterministically solves for a good schedule in one pass without the need for exhaustive search or iterative sampling, but can also be easily applied to different network layers and hardware architectures. This section discusses the CoSA framework and how CoSA formulates the DNN scheduling problem with mixed integer programming (MIP).

4.3.1 CoSA Overview

CoSA optimizes operator-level schedules for mapping DNN layers onto spatial DNN accelerators. Specifically, CoSA formulates the scheduling problem as a constrained-optimization problem with *variables* representing the schedule, *constraints* representing DNN dimensions and hardware parameters, and *objective* functions representing goals, such as maximizing buffer utilization or achieving better parallelism. Figure 4.2 shows the target problem space of

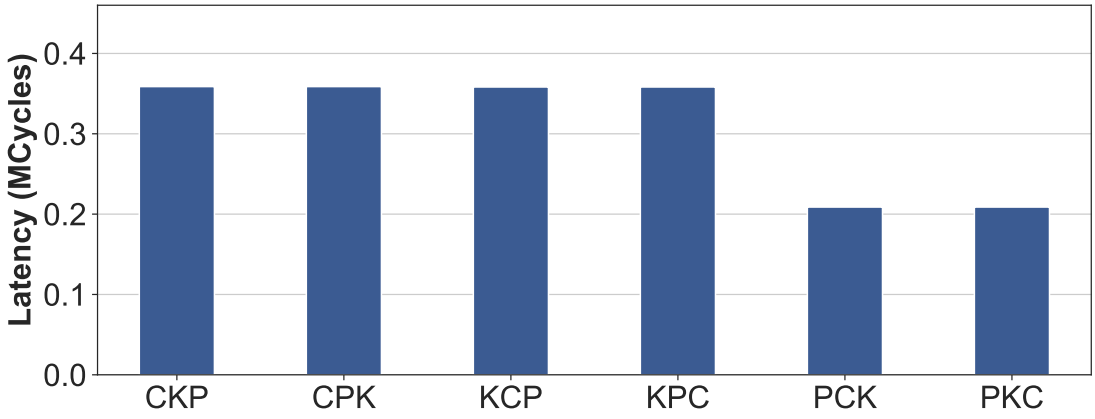


Figure 4.3: Performance comparison of schedules with different loop permutations for a convolution operator with the layer dimensions of $R = S = 3$, $P = Q = 8$, $C = 32$, $K = 1024$. The leftmost schedule (`CKP`) refers to a relative ordering where the input channel dimension (`C`) is the outermost loop and the output height dimension (`P`) is the innermost loop. Since this layer is weight-heavy, loop permutations that emphasize weight reuse, e.g., `PCK` and `PKC`, are more efficient.

CoSA. CoSA takes the specifications of the DNN layers and the underlying spatial accelerator as input constraints and generates a valid and high-performance schedule based on the objective functions in one pass.

4.3.1.1 Target Workload

The work targets the DNN operators that can be expressed by a nested loop with 7 variables as loop bounds: R, S, P, Q, C, K, N. R and S refer to the convolution kernel width and height, P and Q refer to the output width and height, C refers to the input channel size, K refers to the output channel size, and N refers to the batch size, as illustrated in Figure 4.2. The convolution operation computes the dot product of the filter size $R \times S \times C$ of inputs and weights to generate one point in the output. Matrix multiplications can be expressed in this scheme as well.

4.3.1.2 Target Architecture

CoSA targets spatial architectures with an array of processing elements (PEs) connected via an on-chip network and with multiple levels of memory hierarchy, a commonly adopted architecture template in today's DNN accelerator designs [32, 33, 60, 61, 81, 112, 153, 158, 175, 199, 216].

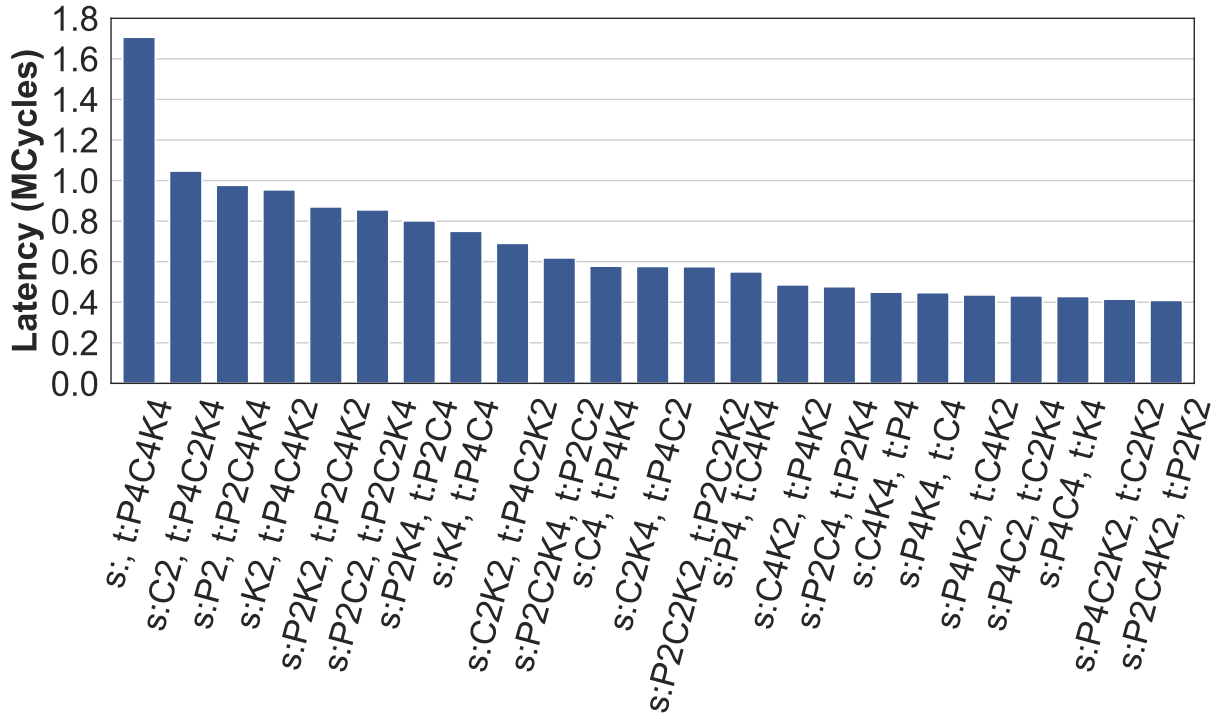


Figure 4.4: Performance comparison of schedules with different spatial mappings for a convolution operator with the layer dimensions of $R = S = 1$, $P = Q = 16$, $C = 256$, $K = 1024$. Factors in s list are for spatial mapping, and factors in t list are for temporal mapping. For example, `s:P4C4,t:K4` represents a mapping where a factor 4 of the `P` dimension and a factor 4 of the `C` dimension are mapped to spatial execution in a system with 16 PEs, leaving `K`'s factor 4 to temporal mapping.

4.3.1.3 Target Scheduling Decisions

CoSA-generated schedules describe how a specified DNN layer is executed on a given spatial architecture. Listing 4.1 shows an example of a schedule. Here, we use a loop-nest representation [146] to explicitly describe how the computation of a convolution layer is mapped to levels of memory hierarchies. We highlight three aspects of the schedule: 1) **loop tiling**, which describes which loops are mapped to which memory level and the values of the loop bounds; 2) **loop permutation**, which handles the relative ordering between loops in the same memory hierarchy; and 3) **spatial mapping**, which defines which loops are mapped to parallel spatial resources (shown as `spatial_for` loops in Listing 4.1). All three factors play a key role in the efficiency of the scheduling choice. Next, we highlight the implications of loop permutation and spatial mapping, both of which are less explored than the well-studied loop tiling.

Figure 4.3 illustrates the impact of **loop permutation** for a convolution layer on a given hardware design. All the schedules use the same loop tiling and spatial mapping except

the loop ordering at the global-buffer level, as indicated in the labels of the X-axis, where `CKP` means the input channel dimension (`C`) is the outermost loop, and the output height dimension (`P`) is the innermost loop. In this case, selecting P as the outermost loop, i.e. `PCK` and `PKC`, can lead to a 1.7× speedup for this layer, motivating the need to consider the implications of loop permutation in the scheduling problem.

Figure 4.4 shows the impact of **spatial mapping** on DNN execution. We notice that there is a 4.3× gap between best (rightmost) and worst (leftmost) schedules for the layer in consideration. The fundamental reason for the differences is the different communication traffic generated by different spatial mapping options. The best schedule, i.e., the rightmost schedule in the figure (`s:P2C4K2, t:P2K2`), is obtained when factors $P = 2$, $C = 4$, $K = 2$ are mapped to the spatial loops, which cannot be achieved by simply choosing either model or data parallelism in the spatial partition. As a result, a systematic evaluation of different spatial mapping choices is required to find a good schedule.

The rest of the section discusses how CoSA formulates the scheduling variables, constraints, and objectives to solve the DNN scheduling problem.

4.3.2 CoSA Variables and Constants

This section discusses the variables and constants, summarized in Table 4.2, used in CoSA formulation.

4.3.2.1 Variable Representation

CoSA Variables		CoSA Constants		Indices	
X	binary matrix to represent a schedule	**A**	layer dimension to data tensor mapping	i	memory level
				j	layer dimension
		B	memory level to data tensor mapping	n	prime factor index
				k	mapping choice
				z	permutation level
				v	data tensor

Table 4.2: CoSA Notations.

We devise a mathematical representation for the DNN schedules and formulate the scheduling problem as a prime-factor allocation problem. Given a layer specification, we first factorize each loop bound into its $prime\_factors$. If the loop bound themselves are large prime number, we can pad them and then factorize. We assign each prime factor to a *scheduling configuration* that is composed of a combination of three decisions: 1) the mapped memory level, 2) the permutation order, and 3) the spatial mapping. Each prime factor has exactly one scheduling configuration.

DNN Layer: $R = 3, S = 1, P = 1, Q = 1, C = 1, K = 4, N = 3$
$\rightarrow$ **Prime Factors:** = [[3],[1],[1],[1],[1],[2,2][3]]

Idx			Perm	Schedule									
j	Layer Dim.			R = 3		...	K = 4				N = 3	
n	Prime Factors			3		...	2		2		3	
k	s / t Mapping			s	t		s	t	s	t	s	t
i	Memory Levels	Register	...									
		...										
		InputBuf	...				✓					
		GlobalBuf	O_0									
			O_1									✓
			O_2						✓			
			...									
			O_Z	✓								

Table 4.3: Example binary matrix **X** representing a schedule. A checkmark in s, t indicates spatial or temporal mapping. A checkmark in $O_0, ..., O_Z$ indicates the rank for loop permutation. In this schedule, the loop tile of size 3 from problem dimension N is allocated within the GlobalBuf at the innermost loop level, assigned for temporal execution. Both loop tiles from K are mapped to spatial resources.

Here, we use a binary matrix **X** to represent the prime factor allocation, i.e., the scheduling space, shown in Table 4.3. The four dimensions of **X** are: 1) the layer dimension variables (indexed by j), 2) the prime factors of the loop bounds (indexed by n), 3) whether it is a spatial or temporal mapping (indexed by k), and 4) the memory and the permutation levels (indexed by i). With the prime factor decomposition, CoSA's encoding can represent all possible schedules and guarantees that the optimization solves for the full search space.

Table 4.3 shows an example binary matrix X that represents the schedule shown in Listing 4.1. First, CoSA performs the *tiling* optimizations by assigning the prime factors to different memory levels. For example, dimension K is split into two tiles, where the inner tile of size 2 is allocated to the input buffer, and the outer tile of size 2 is allocated in the global buffer. Second, mapping a prime factor to *spatial* execution is indicated by whether the factor is mapped to a spatial column s or a temporal column t in the table. In this example, both prime factors for K are spatially mapped. Finally, for loop *permutation*, we add rank indices $O_0, O_1, ..., O_Z$ to the memory level of interest, where only one prime factor can be mapped to each rank. The lowest-ranked factor is allocated to the innermost loop, while the highest-ranked factor is allocated to the outermost loop. In the example shown

	Related			Idx
	W	IA	OA	v
R	✓	-		
S	✓	-		
P		✓	✓	
Q		✓	✓	j
C	✓	✓		
K	✓		✓	
N		✓	✓	

	Related			Idx
	W	IA	OA	v
Register	✓	✓	✓	
AccBuf			✓	
WBuf	✓			i
InputBuf		✓		
GlobalBuf	✓	✓		
DRAM	✓	✓	✓	

Table 4.4: Constant binary matrices **A** (left) and **B** (right). **A** encodes how different layer dimensions associate with data tensors. **B** encodes which data tensor can be stored in which memory hierarchy.

in Table 4.3, the problem dimension N is mapped at the O_1 level in the global buffer for temporal mapping, which means the factor $N = 3$ will be assigned rank 1 in the global-buffer level. Without other factors in the global-buffer level, factor $N = 3$ with the smallest rank will become the innermost loop in permutation. For the ranking of permutation, we reserve enough slots for all prime factors at all memory levels. Not all the slots need to be filled since a prime factor can only be allocated to one memory level.

4.3.2.2 Constant Parameters

In addition to the loop-related variables, we have intrinsic relations across different components in the architecture and layer specifications which must be encoded by constant parameters. CoSA uses two constant binary matrices to encode the unique relations in the DNN scheduling space, shown in Tabel 4.4. The first binary constant matrix, A, encodes the association between layer dimensions (i.e., rows of the matrix) and data tensors (i.e., columns of the matrix). For each input (IA), weight (W), and output (OA) tensor, matrix A indicates which layer dimensions, i.e., R, S, P, Q, C, K, N, should be used to calculate the data transaction size as well as multicast and reduction traffic on the accelerators.

In addition, we introduce another binary matrix **B** to represent which memory hierarchy can be used to store which data tensor. DNN accelerators typically deploy a multi-level memory hierarchy, where each memory level can be used to store different types of data tensors. For example, matrix **B** shown in Table 4.4 represents an architecture that has dedicated input and weight buffers for input activation and weight, respectively, while providing a shared global buffer to store input and output activations.

4.3.3 CoSA Constraints

This section discusses the constraints derived from the target accelerator architecture that must be satisfied in CoSA and shows how to express them with CoSA variables and constants.

4.3.3.1 Buffer Capacity Constraint

To generate a valid schedule in a software-managed memory system, a key constraint is to ensure that the size of data to be sent to the buffer does not exceed the buffer capacity. The hardware memory hierarchy can be represented by the binary constant matrix **B** discussed earlier. For each memory buffer, based on the tensor-dimension correlation matrix **A**, we calculate the tiling size of each tensor by multiplying the relevant prime factors together indicated by **X**. Both spatial and temporal factors should be included in the buffer utilization. Let N_j be the number of prime factors for the layer dimension j. Then the utilization of the buffer level I can be expressed as:

$$\prod_{i=0}^{I-1} \prod_{j=0,n=0}^{6,N_j} \prod_{k=0}^{1} \begin{cases} prime\_factor_{j,n}, & X_{(j,n),i,k} A_{j,v} B_{I,v} = 1 \\ 1, & \text{otherwise} \end{cases} \tag{4.1}$$

We then set the upper bound of the buffer utilization to the capacity of different buffer sizes, represented using $M_{I,v}$. However, a problem with this utilization constraint is that it involves products of the decision variables **X**, making it nonlinear and infeasible to solve with standard constraint solvers. To address this limitation, we take the logarithm of both sides of the constraints to obtain a linear expression for the utilization and encode the if-else statement as:

$$\begin{aligned} U_{I,v} &= \sum_{i=0}^{I-1} \sum_{j=0,n=0}^{6,N_j} \sum_{k=0}^{1} \log(prime\_factor_{j,n}) A_{j,v} B_{I,v} X_{(j,n),i,k} \\ &\leq \log(M_{I,v}), \forall I \end{aligned} \tag{4.2}$$

To encode different precisions for different data tensors, we add the logarithm of the datatype sizes $precision_v$ to $U_{I,v}$.

4.3.3.2 Spatial Resource Constraint

Another set of CoSA constraints is from the limited number of spatial resources. At the chip level, there is a limited number of PEs. At the PE level, there is a limited number of multiply-and-accumulate (MAC) units. In CoSA, once a factor is assigned to spatial mapping in the configuration, it needs to satisfy: 1) each problem factor can only be mapped to either spatial or temporal execution, 2) factors that map to spatial execution do not exceed the resource limit in the architecture. These two constraints can be expressed in the equations below:

$$\sum_{k=0}^{1} X_{(j,n),i,k} == 1, \forall (j,n), i \tag{4.3}$$

$$\sum_{j=0,n=0}^{6,N_j} \log(prime\_factor_{j,n}) X_{(j,n),I,0} \leq \log(S_I), \forall I \tag{4.4}$$

where S_I is the number of available spatial resources at the level I.

4.3.4 Objective Functions

In this section, we describe the objective functions for CoSA. Each objective can be either used individually to optimize a single aspect of performance, e.g., utilization, compute, and communication, or combined with others.

4.3.4.1 Utilization-Driven Objective

High on-chip buffer utilization improves data-reuse opportunity. As demonstrated in the prior work [48], communication lower bounds can be achieved when the tiling block size is optimized for buffer utilization in a system with one-level cache. In this work, we formulate a utilization objective that aims to maximize the buffer utilization of all tensors, so the overall communication is minimized. We use the same formulation for the buffer utilization as in 4.3.3.1 and maximize the following linear utilization function:

$$\hat{Util} = \sum_{i=0}^{I-1} \sum_{v=0}^{2} U_{i,v} \tag{4.5}$$

Here, maximizing the sum of utilization for all buffer levels and all tensors in the logarithm form is equivalent to maximizing the geometric mean of the buffer utilization. Users can also attach weights to the different buffer levels or different data tensors if they want to optimize for the utilization of a specific level of the memory.

4.3.4.2 Compute-Driven Objective

The total number of compute cycles is another factor that affects the quality of schedules. In this formulation, we multiply all the temporal factors for the estimated compute cycles in each PE. Intuitively, this objective allows the constraint solver to exploit the parallelism in the system by mapping more iterations to the spatial resources than to temporal iterations. The objective can be expressed as a linear function again with logarithm taken:

$$\hat{Comp} = \sum_{i=0}^{I} \sum_{j=0,n=0}^{6,N_j} \log(prime\_factor_{j,n}) X_{(j,n),i,1} \tag{4.6}$$

4.3.4.3 Traffic-Driven Objective

Communication latency is a key contributing factor to the performance of spatial architecture. CoSA also includes a traffic-driven objective to capture the communication cost. Specifically, communication traffic can be decomposed into three terms: 1) data size per transfer, 2) spatial factors of multicast and unicast traffic, and 3) temporal iterations. Multiplying these three factors will get the total amount of traffic in the network. Next, we discuss how we capture each of these factors using CoSA's representation.

First, similar to the buffer utilization expression, data size per transfer can computed using the allocated prime factors in matrix $\mathbf{X}$, together with the dimension-tensor correlation matrix $\mathbf{A}$, as shown in the equation below:

$$D_v = \sum_{i=0}^{I-1} \sum_{j=0,n=0}^{6,N_j} \sum_{k=0}^{1} \log(prime\_factor_{j,n}) A_{j,v} X_{(j,n),i,k} \tag{4.7}$$

Second, spatial factors would incur different multicast, unicast, and reduction patterns. The dimension-tensor correlation matrix $\mathbf{A}$ discussed in Sec 4.3.2.2 can be used to indicate different traffic patters. Specifically, depending on whether the spatial dimension, indicated by the binary matrix $\mathbf{X}$, is related to the specific tensor in consideration, represented by the constant matrix $\mathbf{A}$, different traffic patterns, e.g., multicast vs. unicast or reduction vs. unicast, would occur.

Figure 4.5 shows how the intrinsic tensor-dimension correlation matrix $\mathbf{A}$ can be used to calculate different traffic patterns for different variables. For example, as shown in Figure 4.5a, if the dimension P is mapped spatially, $A_{P,\mathrm{W}} = 0$ implies multicast traffic for weight tensor W. Since weight is not related to P, when we send weights from global buffer to PEs, the weight traffic will be multicasted to the destination PEs. If the dimension C is mapped spatially, $A_{C,\mathrm{W}} = 1$ (Figure 4.5b) implies unicast traffic for weight tensor W as weight is related to C. Similarly, if the dimension C is mapped spatially, $A_{C,\mathrm{OA}} = 0$ (Figure 4.5c) implies reduction traffic for output tensor OA, where partially sum needs to be reduced across C before sending back to GB. If the dimension P is mapped spatially, $A_{P,\mathrm{OA}} = 1$ (Figure 4.5d) would indicate unicast traffic for output tensor OA, as each traffic contributes to different regions of the output. CoSA formulates this relationship in the following equation:

$$L_v = \sum_{j=0,n=0}^{6,N_j} \log(prime\_factor_{j,n}) X_{(j,n),I,0} A_{j,v} \tag{4.8}$$

The third term, temporal iteration is used to calculate the number of data transfers at the NoC level. We introduce a traffic iteration factor $\mathbf{Y}$ that is a function of $\mathbf{X}$ at the permutation level, $\mathbf{A}$, and $\mathbf{B}$. $\mathbf{Y}$ indicates if the outer NoC loop bound should be used for different variables. With $\mathbf{Y}$, we ensure that, for each variable, if a relevant factor term is seen inside the current loop level, the current loop level's factor should be used to compute the traffic iteration regardless of whether it is related to the data tensor of the variable of interest.

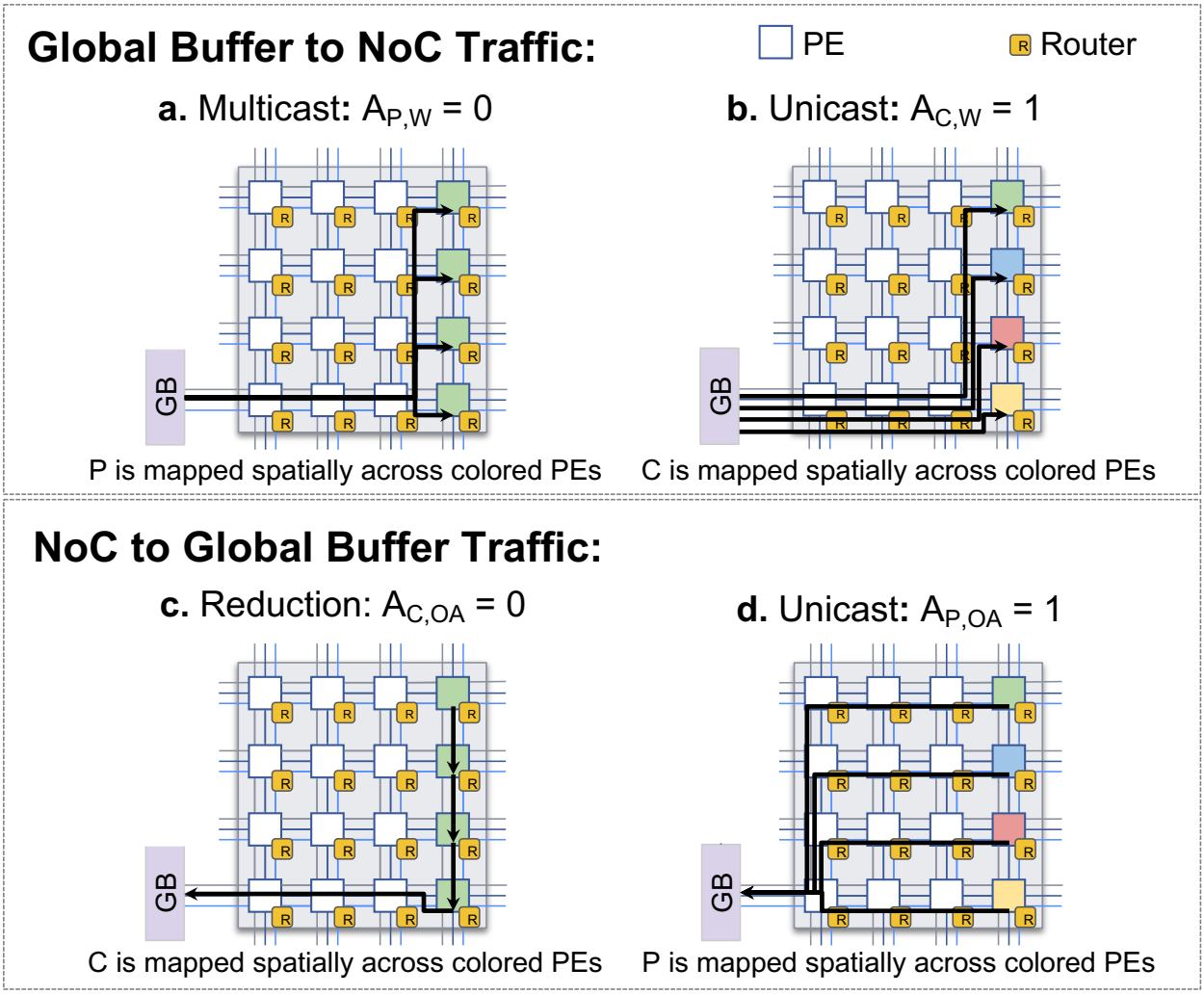


Figure 4.5: Different traffic patterns based on the constant matrix $\mathbf{A}$. The two figures (top) show how the constant $\mathbf{A}$ encodes the traffic types (multicast, unicast, reduction) for different data tensors from the global buffer to PEs. The figures on the bottom show its implication on output tensor reduction traffics.

This is a term that drives the reuse optimization. Mathematically, $\mathbf{Y}$ is constrained as:

$$\begin{aligned} Y_{v,z} &\geq \sum_{j=0,n=0}^{6,N_j} X_{(j,n),z,1} A_{j,v} B_{I,v}, \forall z, \forall v \\ Y_{v,z} &\geq Y_{v,z-1}, \forall z > 0, \forall v \end{aligned} \tag{4.9}$$

Where z represents the position index for permutation and Z equals the total valid levels for

permutation. The traffic iteration term can thus be expressed as:

$$T_v = \sum_{z=0}^{Z-1} \sum_{j=0,n=0}^{6,N_j} \log(prime\_factor_{j,n}) Y_{v,z} X_{(j,n),z,1} \tag{4.10}$$

This turns the linear objective into quadratic as we multiply $\mathbf{Y}$ with $\mathbf{X}$ to indicate whether there is a factor at the current permutation level.

After we calculate each individual term, we can combine them together for each tensor that contributes to the total traffic in the network. Similar to the logarithmic transformation we did earlier, instead of multiplying these three terms together, we take the logarithm on both sides to get a linear expression of the traffic, as shown in the equation below:

$$\hat{Traf} = \sum_{v=0}^{2} (D_v + L_v + T_v) \tag{4.11}$$

4.3.4.4 Overall Objective

One can construct a composite objective comprised of a linear combination of $\hat{Util}$, $\hat{Comp}$, and $\hat{Traf}$, where we want to minimize the compute and communication latency while maximizing the on-chip buffer utilization:

$$\hat{O} = -w_U \hat{Util} + w_C \hat{Comp} + w_T \hat{Traf} \tag{4.12}$$

where w_U, w_T, w_C are user-selected parameters controlling the importance of each objective. For a system with double-buffering optimization, w_T can be set to map the traffic sizes to the cycles for memory accesses. This brings $w_T \hat{Traf}$ to be of the same importance as $w_C \hat{Comp}$ in the optimization. Another formulation of the overall objective function to balance the memory access and compute cycles is to minimize the difference of the two terms: $\hat{D} = w_T \hat{Traf} - w_C \hat{Comp}$. The weights of different objectives can be determined by using a set of micro-benchmarks that characterize the compute, memory, and communication latencies of the target architecture.

4.3.5 Limitation of CoSA

CoSA leverages the regularity from both the problem and the architecture space, where it assumes a dense CNN workload and does not exploit the sparsity of the data. It also best targets hardware systems with deterministic behavior and explicitly managed scratchpads. This is because, in systems with non-deterministic behaviors, it can be challenging to construct optimization objectives that capture the impact of such behaviors. However, CoSA can be augmented with an iterative search on the objective functions and their corresponding hyperparameters to approximate the unknown hardware performance model and directly prune off the invalid points from the search space.

Arithmetic :		*Storage :*		*Network :*	
MACs	64 / PE	**Registers**	64B / PE	**Dimension**	4×4
Weight/Input Precision	8bit	**Accum. Buffer**	3KB / PE	**Router**	Wormhole
		Weight Buffer	32KB / PE	**Flit Size**	64b
Partial-Sum Precision	24bit	**Input Buffer**	8KB / PE	**Routing**	X-Y
		Global Buffer	128KB	**Multicast**	Yes

Table 4.5: The baseline DNN accelerator architecture.

4.4 Methodology

This section discusses the evaluation platforms we use followed by the experimental setup for CoSA evaluation.

4.4.1 Evaluation Platforms

We evaluate the schedules generated by CoSA on two platforms: 1) Timeloop for cycle performance and energy consumption, and 2) our cycle-exact NoC simulator for overall latency performance. The latter more accurately captures the communication overhead and concurrent hardware behaviors on a spatial architecture.

Timeloop provides microarchitecture and technology-specific energy models for estimating the performance and energy on DNN accelerators. Timeloop reports the performance in terms of the maximum cycles required for each processing element to complete the workload and to perform memory accesses, assuming perfect latency hiding with double buffering. The energy consumption in Timeloop is calculated by multiplying the access count on each hardware component with the energy per access and summing the products up. The access count is inferred from the schedule and the energy per access is provided by an energy reference table in Timeloop. The specific Timeloop version we use in this work is commit a9d08f0 from the GitHub repo.

NoC Simulator augments the Timeloop analytical compute model for PEs with a synthesizable NoC implementation to reflect the communication cost. Communication is one of the key contributing factors for latency in a NoC-based system, especially for the communication bound schedules.

To accurately characterize the end-to-end accelerator performance of CoSA generated schedules, We implement this cycle-exact transaction-based NoC simulation infrastructure in SystemC and Python. The NoC simulator is transaction-based and cycle-exact for modeling the on-chip traffic. Leveraging the synthesizable SystemC router design from Matchlib [104] that supports unicast and multicast requests, we construct a resizable 2-D mesh network and implement an X-Y routing scheme. The simulator captures both computation and communication latencies by concurrently modeling data transfers in the

Network	Number of Unique Layers
AlexNet [110]	8
ResNet-50 [78]	21
ResNeXt-50 (32x4d) [209]	19
Deepbench [46] (OCR and Face Recognition)	9

Table 4.6: Summary of DNN workloads used in this study

NoC, the PE executions, and off-chip DRAM accesses based on the DRAMSim2 model [168], where the impact of traffic congestion on the NoC can also be manifested.

4.4.2 Baseline Schedulers

We evaluate CoSA with respect to two other scheduling schemes: 1) a **Random** scheduler that searches for five different valid schedules, from which we choose the one with the best result for the target metric, and 2) the **Timeloop Hybrid** mapper in Timeloop [146] that randomly selects a tiling factorization, prunes superfluous permutations, and then linearly explores the pruned subspace of mappings before it proceeds to the next random factorization. For this mapper, we keep the default termination condition where each thread self-terminates after visiting 500 consecutive mappings that are valid yet sub-optimal. The mapper is run with 32 threads, each of which independently searches the scheduling space until its termination condition is met. Once all threads have terminated, Timeloop returns the best schedule obtained from all 16,000+ valid schedules.

4.4.3 Experiment Setup

Mixed-Integer Program (MIP) Solver: CoSA uses Gurobi [68], a general-purpose optimization solver for MIP and other constrained programming, as the solver. We specify the CoSA variables, constraints, and objective functions before we invoke the solver. The solver takes at most seconds to return a schedule for DNN layers.

DNN workloads: We measure the performance of CoSA-generated schedules over a wide range of DNN workloads targeting different DNN tasks with diverse layer dimensions, including: ResNet-50 [78], ResNeXt-50 (32x4d) [209], and Deepbench [46] (OCR and Face Recognition). As summarized in Table 4.6, we benchmark the schedule performance of a wide range of networks. The precision used for the benchmarks is 8-bit for the input and weights, and 24-bit for the partial sums. We do not pad the dimensions to be multiples of 2, as it incurs more overhead and outweighs the benefits it provides to allow more scheduling options.

	CoSA	Random (5×)	Timeloop Hybrid
Avg. Runtime / Layer	**4.2s**	4.6s	379.9s
Avg. Samples / Layer	**1**	20K	67M
Avg. Evaluations / Layer	**1**	5	16K+

Table 4.7: Time-to-solution Comparison. CoSA outputs only one valid schedule per layer. CoSA's runtime is 1.1× and 90× shorter than the Random and Timeloop Hybrid search, respectively.

Baseline architecture: We consider a spatial-array architecture like Simba [174] as our baseline. Detailed specifications of the hardware constructs are summarized in Table 4.5. We demonstrate that the CoSA framework is general to be applied for different architecture parameters while delivering high-performance scheduling options in one shot.

4.5 Evaluation

In this section, we demonstrate the improved time-to-solution, performance, and energy of CoSA compared to baseline schedulers, across different evaluation platforms and different DNN architectures on a diverse set of DNN layers.

4.5.1 Time to Solution

We compare the average time for CoSA and the baseline schedulers to generate the schedule of each layer from the four target DNN workloads. Table 4.7 shows that CoSA's optimization-driven approach offers more than 90× (4.2s vs. 379.9s) time-to-solution advantage over the Timeloop Hybrid search strategy. Timeloop Hybrid search sampled 67 million schedules per layer and evaluated more than 16 thousand valid ones among them, leading to a long runtime. With Random search, a random sampling of 20K samples in 4.6 seconds resulted in only five valid schedules, further demonstrating the need to have a constraint-based strategy to prune the invalid search space directly. In the following section, we show that CoSA not only shortens the time-to-solution but also generates high-quality schedules.

4.5.2 Evaluation on Timeloop Performance and Energy Models

We compare the performance of the Random search, the Timeloop Hybrid mapper, and the CoSA scheduler for four different DNN workloads. The evaluations are based on our baseline architecture described in Table 4.5 and the Timeloop evaluation platform mentioned in Section 4.4.1.

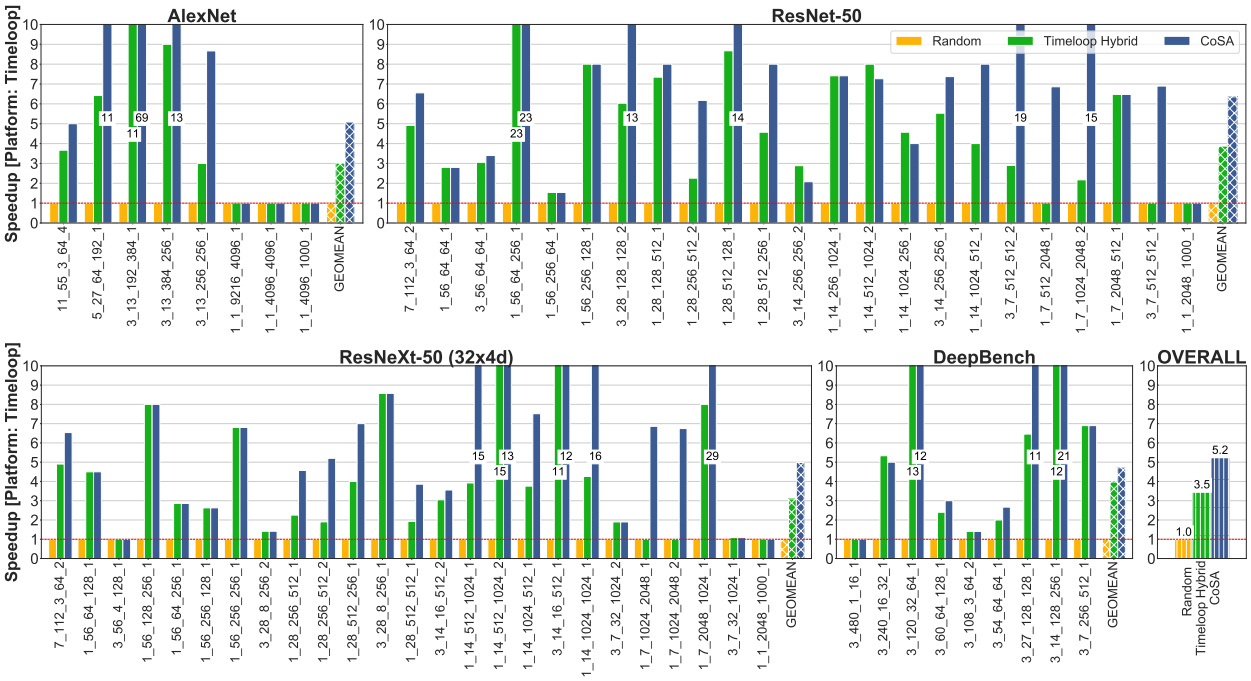


Figure 4.6: Speedup of different schedules relative to Random search on the baseline 4×4 NoC architecture. X-axis labels follow the naming convention `R_P_C_K_Stride` where $S = R$ and $Q = P$ in all workloads. CoSA achieves 5.2× and 1.5× higher geomean speedup across four DNN workloads compared to the Random and Timeloop Hybrid search.

4.5.2.1 Performance

Figure 4.6 shows the speedup reported by Timeloop for different scheduling schemes relative to Random search. Figure 4.6 demonstrates that the CoSA-generated schedules are not only valid but also outperform the ones generated by both Random search and Timeloop Hybrid search. The geometric mean of the speedups of CoSA schedules relative to the Random and Timeloop Hybrid search ones are 5.2× and 1.5× respectively across four DNNs.

In the few layers where Timeloop Hybrid search slightly outperforms CoSA, we find a higher iteration count at the DRAM level in Timeloop Hybrid schedules, which helps to reduce the size of each DRAM transaction and balance the pipeline. Fine-tuning the weights of the objective functions could be used to improve the CoSA-generated schedules further.

A more exhaustive Timeloop Hybrid search (32K valid schedules) improves only 7.5% in latency while increasing runtime by 2×. We find that even with 2× more valid samples evaluated, Timeloop Hybrid search still cannot generate schedules that are of similar efficiency to CoSA.

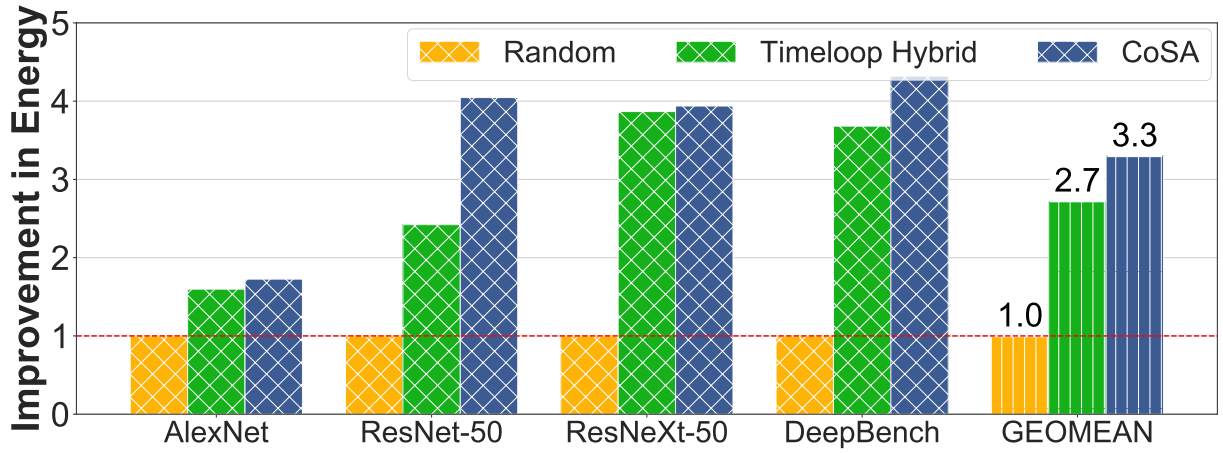


Figure 4.7: Improvements in total network energy reported by the Timeloop energy model. Energy estimations are normalized to results from Random search and are evaluated on the baseline 4×4 NoC.

4.5.2.2 Energy

We use the Timeloop energy model to evaluate the energy of different schedules. Because energy cost is highly correlated with the access count on each hardware component, our traffic objective in CoSA is used for the schedule optimization targeting energy efficiency. Figure 4.7 demonstrates that CoSA, using no simulation feedback, can generate schedules 22% more energy-efficient than the best Timeloop Hybrid solutions selected from 16,000+ valid schedules optimizing the energy.

4.5.2.3 Objective Breakdown

A detailed breakdown of the CoSA objective function on ResNet50 layer 3_7_512_512_1 is included in Figure4.8. Our overall objective function aims to capture an optimization heuristic to maximize the utilization and minimize the compute and traffic costs at the same time with a weighted sum of the three. Figure4.8 shows that CoSA achieves the lowest total objective among all approaches, and optimizes all three sub-objectives simultaneously. This observation on the objective values aligns with our empirical results in Figure 4.6, where CoSA schedule runs 7× faster than the ones generated by Random and Timeloop Hybrid search.

4.5.2.4 Different HW Architectures

We further explore the performance of CoSA with different DNN architecture parameters such as different PE array sizes and different SRAM buffer sizes. We apply the same weights for the evaluation on the same architecture and customize the objective weights in Eqn.4.12

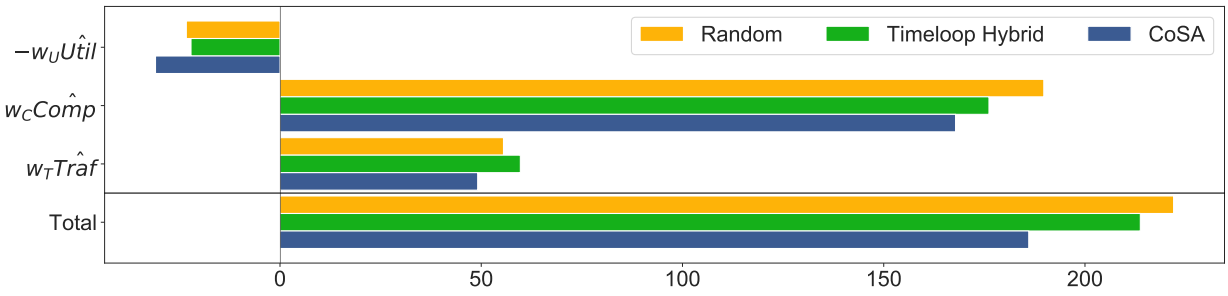
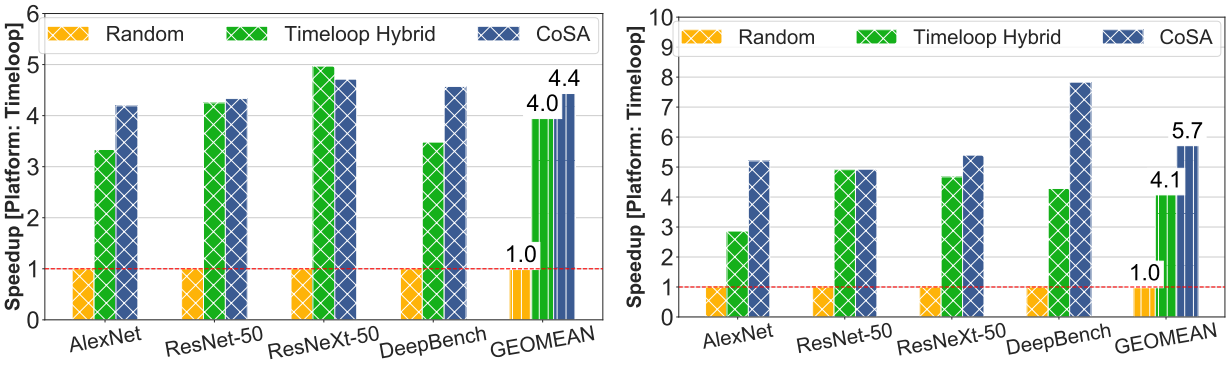


Figure 4.8: Objective function breakdown for ResNet-50 layer 3_7_512_512_1. The goal is to minimize the total objective in Eq. 4.12. CoSA achieves the lowest values for all objective functions on this layer among all approaches.



(a) 8×8 PEs

(b) Larger Buffers

Figure 4.9: Speedup relative to Random search reported by Timeloop model on different hardware architectures. CoSA's performance generalizes across different hardware architectures with different computing and on-chip storage resources.

using a micro-benchmark for different architectures. Figure4.9 shows the geomean speedup of CoSA across all networks on two different hardware architectures.

PE Array Dimension. We scale the number of PEs up by 4× and increase both the on-chip communication and DRAM bandwidth by 2× correspondingly. Both of these modifications significantly impact the compute and communication patterns of DNN layer executions. With a larger spatial array of arithmetic units, this case study presents a scheduling problem where decisions about spatial and temporal mapping can be especially crucial to attaining high performance. Figure 4.9a shows that CoSA achieves 4.4× and 1.1× speedup compared to Random and Timeloop Hybrid search respectively across four networks. This shows that the performance of our scheduler can scale and generalize to NoCs with more PEs, which tend to be more affected by communication costs.

SRAM Size. We also increase the sizes of the local and global buffers to demonstrate

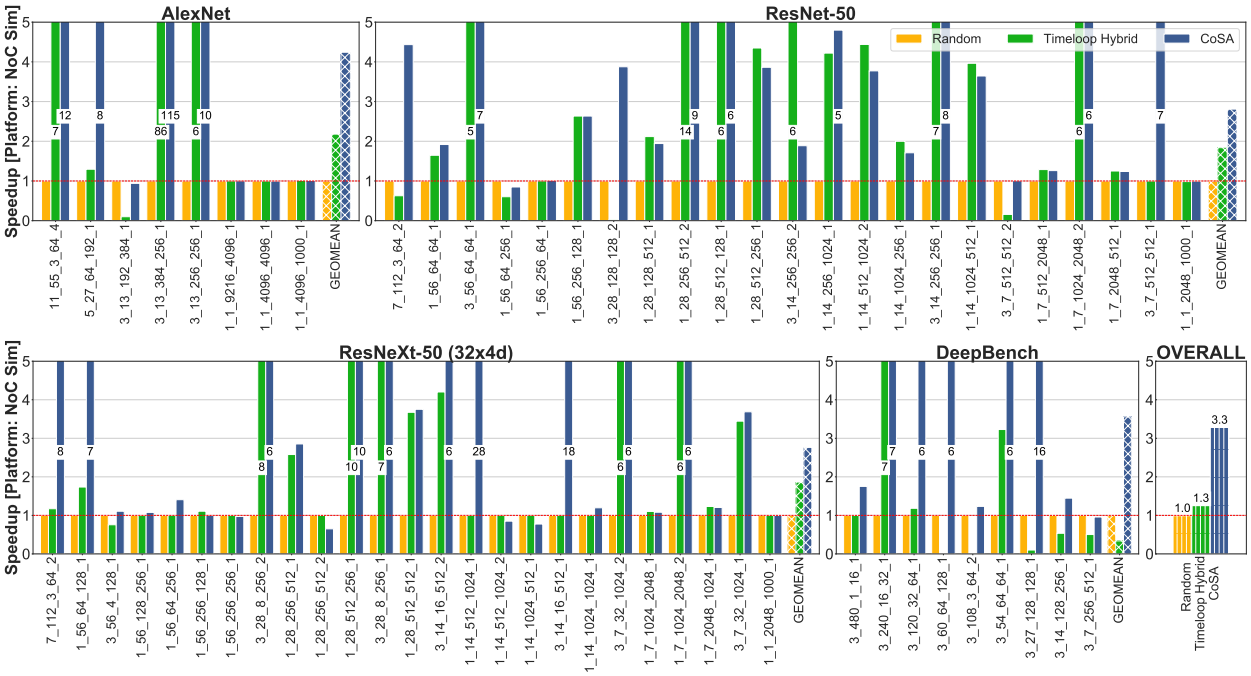


Figure 4.10: Speedup reported by NoC simulator relative to Random search on the baseline 4×4 NoC architecture. CoSA achieves 3.3× and 2.5× higher geomean speedup across four DNN workloads compared to the Random and Timeloop Hybrid search on the more communication sensitive NoC simulator.

that CoSA can achieve consistently good schedules across different architectures. The sizes of local buffers, i.e., accumulation, weight, and input buffers, are doubled; and the global buffer size is increased 8×. At the PE and global buffer level, modified memory capacities are likely to impact the optimal strategy for data reuse and NoC communication traffic reduction. With CoSA, we show 5.7× speedup over Random and 1.4× speedup over Timeloop Hybrid search in Figure 4.9b, demonstrating CoSA's capability across different architectures.

4.5.3 Evaluation on NoC Simulator

To further compare the quality of schedules generated by different scheduling schemes, we evaluate them on our NoC simulation platform. The NoC simulation platform more accurately captures the communication overhead from the on-chip network as compared to the Timeloop models.

Figure 4.10 shows the speedup relative to the Random baseline. We observe that CoSA-generated schedules outperform the baseline schedules for all four DNN workloads, with the greatest performance gains occurring for convolutional layers, e.g., DeepBench layers. Intriguingly, for these same layers, Timeloop Hybrid scheduler actually under-performs Random search as its internal analytical model does not accurately capture the communication

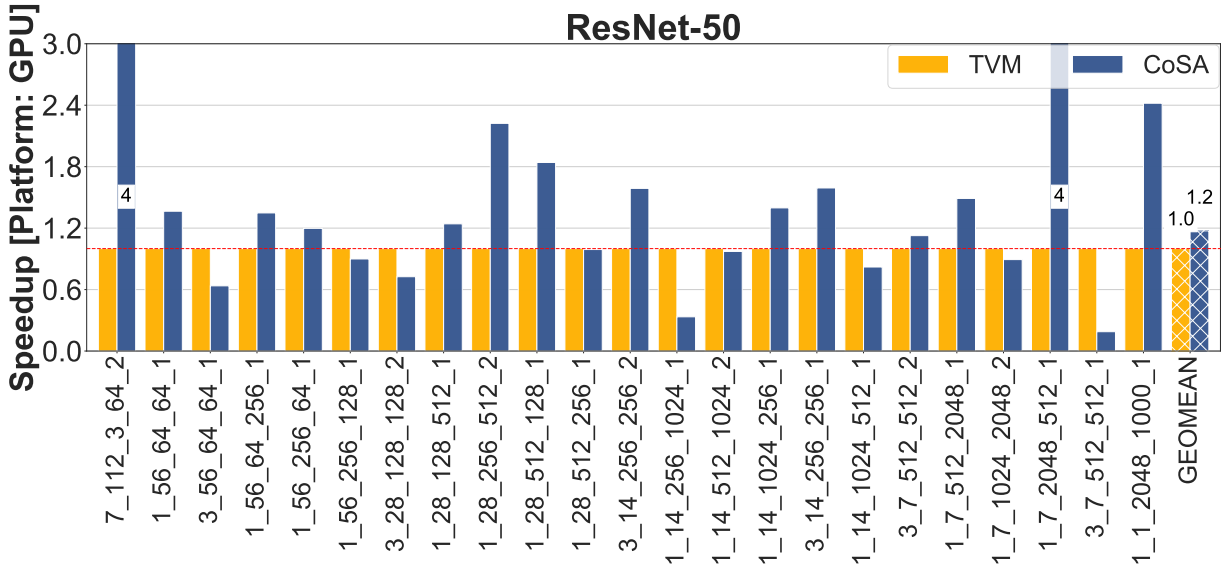


Figure 4.11: Speedup relative to TVM reported on K80 GPU.

traffic in the network. On the other hand, there is no significant difference between the performance of FC layers among different schedules, as the FC layers are heavily memory-bound with low PE utilization. The DRAM access time dominates in these layers even with the best schedules with respect to the reuse of buffered data. Overall, CoSA achieves a geometric average of up to 3.3× speedup relative to the best Random search solutions and 2.5× relative to Timeloop Hybrid search schedules across the four networks. Furthermore, unlike the iterative nature of Random and Timeloop Hybrid search schedules, CoSA schedules are consistently performant with the one-shot solution.

4.5.4 Evaluation on GPU

To show the potential use of CoSA for general-purpose hardware, we also formulate GPU scheduling as a constrained-optimization problem using CoSA. We evaluate the performance of CoSA on GPU and compare it against TVM [29].

Target GPU. We target NVIDIA K80 GPU with 2496 CUDA cores and a 1.5MB L2 cache. This GPU has a 48KB shared memory and 64KB local registers, shared by a maximum of 1024 threads in each CUDA thread block. The thread block is a programming abstraction that represents a group of threads that can be run serially or in parallel in CUDA. The maximum dimension of a thread block is (1024, 1024, 64). Violation of these constraints in the CUDA kernel results in invalid schedules.

Constraints. CoSA expresses the hardware constraints for GPU thread groups and shared/local memory similarly to how we specify the spatial resource and buffer capacity constraints in Section 4.3.3. Each thread group can be seen as a spatial level with a specific

size. The product of all three thread group sizes is enforced to be smaller than 1024. The share memory utilization is calculated as buffer capacity constraints, and the register utilization is calculated by multiplying the total number of threads with the inner loop register utilization.

Objective Functions. In CoSA, we compute the compute objective by discounting the total compute cycles with the total number of threads for GPU, to reflect the performance gain from thread-level parallelism. We then adjust the weights of the other objectives using a micro-benchmark.

We run TVM with the XGBoost tuner for 50 trials per layer as the baseline. CoSA generates valid schedules in one shot with a time-to-solution $2,500\times$ shorter than TVM (0.02s vs. 50s per layer). The CoSA-generated schedules achieve $1.10\times$ geomean speedup compared to the TVM schedules on ResNet50 as shown in Figure4.11.

4.6 Scheduling-Informed Hardware Design

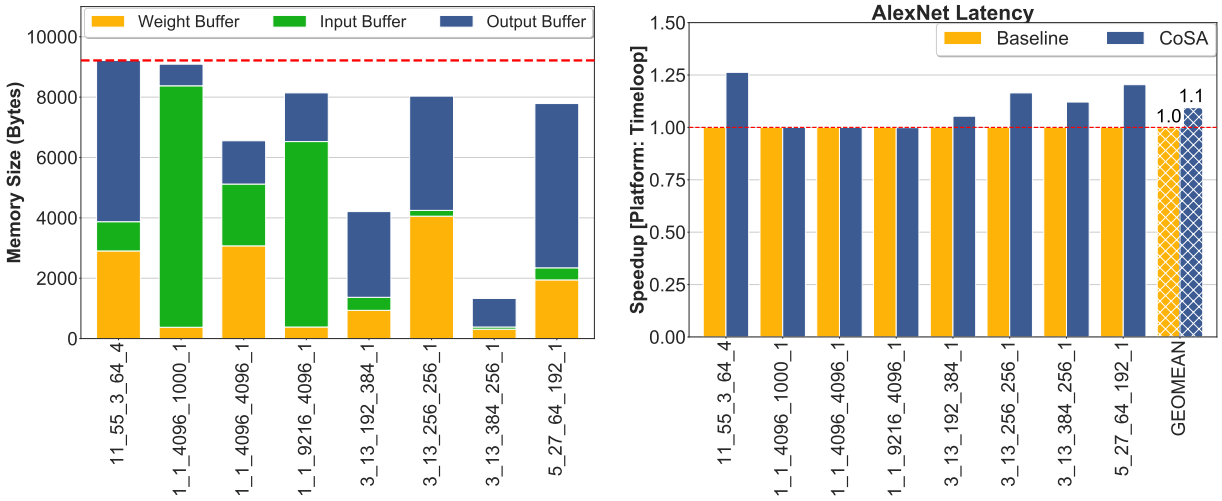
4.7 On-chip Memory Partitioning with CoSA

In addition to scheduling for a given hardware, due to the fast, one-shot nature of CoSA, it can also be applied for hardware and scheduling co-design problems. In particular, on-chip memory partitioning is a critical design decision that can greatly impact not only the overall area budget but also the scheduling decision, especially in architectures with multi-level private buffers for different data tensors. Given an on-chip memory budget, the memory partitioning algorithm determines the memory portion assigned to each local buffer. Current work on the design space exploration of accelerators for resource allocation [38, 100, 146, 216, 230] rely on the iterative scheduling schemes that are computationally expensive and can yield sub-optimal solutions. Our work is the first work that formulates both the scheduling decisions and the on-chip memory partitioning problem as a single optimization problem.

4.7.0.1 Formulation

To co-optimize the scheduling and memory partitioning decisions, we modify the formulations in Section 4.2.2 to include memory sizes also as CoSA variables. Instead of treating the log capacity $log(M_{I,v})$ for different buffers as constraints in Section 4.3.3.1, we turn them into MIP variables $m_{I,v}$ that represents the log value of the actual buffer size $w_{I,v}$. Assume we have H on-chip buffers, we then need to add another constraint to ensure the total size of all buffers does not exceed the allocated budget G:

$$\sum_{I=0}^{H-1}\sum_{v=0}^{2} w_{I,v} = \sum_{I=0}^{H-1}\sum_{v=0}^{2} 2^{m_{I,v}} \leq G \tag{4.13}$$



(a) CoSA-generated memory partitions for AlexNet layers.

(b) Speedup on hardware with CoSA-generated memory partitions for AlexNet layers.

4.7.0.2 Evaluation on On-chip Memory Partitioning

Co-optimizing hardware and schedule opens up new optimization opportunities to balance different tradeoffs in the scheduling space and hardware design. As a starting point, we demonstrate how CoSA can be extended to capture the design and scheduling tradeoffs. In particular, Figure 4.12a shows the on-chip memory partitions solved by CoSA for each layer in AlexNet. We observe significantly different preferred partitions for different layers, where some partition leads to up to an 85% reduction in the total SRAM size. At the same time, Figure 4.12b shows the corresponding performance of running each layer with CoSA-generated schedules on the co-optimized hardware design. We see an 11% improvement in the geomean speedup on the co-optimized hardware. This case study shows a promising application of CoSA in hardware-software co-design. CoSA can be further extended to co-optimize other hardware design decisions by relaxing the architectural constraints in a similar manner as illustrated in this case study.

4.8 Conclusion

In CoSA, we present an optimization-driven approach to DNN scheduling. Harnessing the regularities from DNN workloads and target accelerator designs, we formulate scheduling into a constrained optimization problem that can be solved directly without incurring the high cost of enumeration-based scheduling. We devise a single mathematical formulation to simultaneously solve all three key optimizations in scheduling: loop tiling, loop permutation, and spatial mapping. We implement a cycle-exact NoC simulator to evaluate different schedules more accurately. Comparing our results to schedules generated from the state-of-

the-art work, our approach achieves up to 2.5× speedup and 22% better energy-efficiency, with 90× shorter time-to-solution. We consistently observe improved scheduling performance across different DNN benchmarks and architecture variations.

In addition, we extend CoSA to co-optimize the hardware design and scheduling in unison, a process that is quite time- and resource-intensive today. By transforming the architectural constraints into variables in our formulation, our co-optimized, one-shot solutions can further improve performance by 11% while saving up to 85% on-chip memory.

Chapter 5

Machine Learning for Hardware Design

Machine learning (ML) is poised to revolutionize the performance of numerous applications. It has achieved unparalleled success in computer vision, NLP, computer graphics, work automation, etc. Since the compute improvement has been a key driving force behind the ML progress, we focused on advancing ML through co-designing and optimizing accelerator systems in the previous parts of the dissertation. Given the rise of ML and its success in many domains, we investigate applying machine learning to improve the accelerator design tool in this part of the dissertation.

In this chapter, we present AutoPhase, in which we investigate the effectiveness of deep reinforcement learning algorithms in addressing an NP-hard compiler optimization problem called the *phase-ordering* problem. In HLS-based hardware generation flow, phase-ordering can significantly affect the quality of results. It generally refers to the process of choosing a good order of the optimization passes to apply. In AutoPhase, we implemented a framework that takes a program and uses deep reinforcement learning to find a sequence of compilation passes that minimizes its execution time. Without loss of generality, we construct this framework in the context of the LLVM compiler toolchain and target high-level synthesis programs. We use random forests to quantify the correlation between the effectiveness of a given pass and the program's features. This helps us reduce the search space by avoiding phase orderings that are unlikely to improve the performance of a given program. We also compare the performance of AutoPhase to state-of-the-art algorithms that address the phase-ordering problem.

5.1 Machine Learning for Phase Ordering

The performance of the code a compiler generates depends on the order in which it applies the optimization passes. Choosing a good order–often referred to as the *phase-ordering* problem, is an NP-hard problem. As a result, existing solutions rely on a variety of heuristics. In this work, we evaluate a new technique to address the phase-ordering problem: deep reinforcement learning.

High-Level Synthesis (HLS) automates the process of creating digital hardware circuits from algorithms written in high-level languages. Modern HLS tools [27, 94, 211] use the same front-end as the traditional software compilers. They rely on traditional software compiler techniques to optimize the input program's intermediate representation (IR) and produce circuits in the form of RTL code. Thus, the quality of compiler frontend optimizations directly impacts the performance of HLS-generated circuits.

Program optimization is a notoriously difficult task. A program must be just in "the right form" for a compiler to recognize the optimization opportunities. This is a task a programmer might be able to perform easily but is often difficult for a compiler. To add to this complexity, often, the optimization is hardware-dependent. Despite a decade of research on developing sophisticated optimization algorithms, there is still a performance gap between the HLS generated code and the hand-optimized one produced by experts.

This work builds off the LLVM compiler [114]. However, our techniques, can be broadly applicable to any compiler that uses a series of optimization passes. In this case, the optimization of an HLS program consists of applying a sequence of analysis and optimization phases. Each phase in this sequence consumes the output of the previous phase, and generates a modified version of the program for the next phase. Unfortunately, these phases are not commutative, which makes the order in which these phases are applied very critical to the performance of the output.

Consider the program in Figure 5.1, which normalizes a vector. Without any optimizations, the `norm` function will take $\Theta(n^2)$ to normalize a vector. However, a smart compiler will implement the *loop invariant code motion (LICM)* [135] optimization, which allows it to move the call to `mag` above the loop, resulting in the code on the left column in Figure 5.2. This optimization brings the runtime down to $\Theta(n)$—a big speedup improvement. Another optimization the compiler could perform is *(function) inlining* [135]. With inlining, a call to a function is simply replaced with the body of the function, reducing the overhead of the function call. Applying inlining to the code will result in the code in the right column of Figure 5.2.

Now, consider applying these optimization passes in the opposite order: first inlining, then LICM. After inlining, we get the code on the left of Figure 5.3. Once again, we get a modest speedup, having eliminated n function calls, though our runtime is still $\Theta(n^2)$. If the compiler afterwards attempted to apply LICM, we would find the code on the right of Figure 5.3. LICM was able to successfully move the allocation of sum outside the loop. However, it was unable to move the instruction setting `sum=0` outside the loop, as doing so would mean that all iterations excluding the first one would end up with a garbage value for sum. Thus, the internal loop will not be moved out.

As this simple example illustrates, the order in which the optimization phases are applied can be the difference between the program running in $\Theta(n^2)$ versus $\Theta(n)$. It is thus crucial to determine the optimal phase ordering to maximize the circuit speeds. Unfortunately, not only is this a difficult task, but the optimal phase ordering may vary from program to program. Furthermore, it turns out that finding the optimal sequence of optimization phases is an NP-hard problem, and exhaustively evaluating all possible sequences is infeasible in practice.

```
__attribute__((const))
double mag(int n, const double *A) {
    double sum = 0;
    for(int i=0; i<n; i++){
        sum += A[i] * A[i];
    }
    return sqrt(sum);
}
void norm(int n, double *restrict out,
          const double *restrict in) {
    for(int i=0; i<n; i++) {
        out[i] = in[i] / mag(n, in);
    }
}
```

Figure 5.1: A simple program to normalize a vector.

```
void norm(int n, double *restrict out,
          const double *restrict in) {
    double precompute = mag(n, in);
    for(int i=0; i<n; i++) {
        out[i] = in[i] / precompute;
    }
}
```

```
void norm(int n, double *restrict out,
          const double *restrict in) {
    double precompute, sum = 0;
    for(int i=0; i<n; i++){
        sum += A[i] * A[i];
    }
    precompute = sqrt(sum);
    for(int i=0; i<n; i++) {
        out[i] = in[i] / precompute;
    }
}
```

Figure 5.2: Progressively applying LICM (left) then inlining (right) to the code in Figure 5.1.

In this work, for example, the search space extends to more than 2^{247} phase orderings.

The goal of AutoPhase is to provide a mechanism for automatically determining good phase orderings for HLS programs to optimize for the circuit speed. To this end, we aim to leverage recent advancements in deep reinforcement learning (RL) [72, 190] to address the phase ordering problem. With RL, a software agent continuously interacts with the environment by taking actions. Each action can change the state of the environment and generate a "reward". The goal of RL is to learn a policy—that is, a mapping between the observed states of the environment and a set of actions—to maximize the cumulative reward. An RL algorithm that uses a deep neural network to approximate the policy is referred to as a deep RL algorithm. In our case, the observation from the environment could be the program and/or the optimization passes applied so far. The action is the optimization pass to apply next, and the reward is the improvement in the circuit performance after applying this pass. The particular framing of the problem as an RL problem has a significant impact on the solution's effectiveness. Significant challenges exist in understanding how to formulate

```
void norm(int n, double *restrict out,
          const double *restrict in) {
    for(int i=0; i<n; i++) {
        double sum = 0;
        for(int j=0; j<n; j++){
            sum += A[j] * A[j];
        }
        out[i] = in[i] / sqrt(sum);
    }
}
```

```
void norm(int n, double *restrict out,
          const double *restrict in) {
    double sum;
    for(int i=0; i<n; i++) {
        sum = 0;
        for(int j=0; j<n; j++){
            sum += A[j] * A[j];
        }
        out[i] = in[i] / sqrt(sum);
    }
}
```

Figure 5.3: Progressively applying inlining (left) then LICM (right) to the code in Figure 5.1.

the phase ordering optimization problem in an RL framework.

We consider three approaches to represent the environment's state. The first approach is to directly use salient features from the program. The second approach is to derive the features from the sequence of optimizations we applied while ignoring the program's features. The third approach combines the first two approaches. We evaluate these approaches by implementing a framework that takes a group of programs as input and quickly finds a phase ordering that competes with state-of-the-art solutions. In this chapter, we present:

- An importance analysis on the features using random forests to significantly reduce the state and action spaces.
- A framework that integrates the current HLS compiler infrastructure with the deep RL algorithms.

We show that AutoPhase gets a 28% improvement over -O3 for nine real benchmarks. Unlike all state-of-the-art approaches, deep RL demonstrates the potential to generalize to thousands of different programs after training on a hundred programs.

5.2 Background and Motivation

5.2.1 Compiler Phase-ordering

Compilers execute optimization passes to transform programs into more efficient forms to run on various hardware targets. Groups of optimizations are often packaged into "optimization levels", such as -O0 and -O3, for ease. While these optimization levels offer developers a simple set of choices, they are handpicked by the compiler designers and often most benefit specific groups of benchmark programs. The compiler community has attempted to address the issue by selecting a particular set of compiler optimizations on a per-program or per-target basis for software [4, 7, 145, 195].

Since the search space of phase-ordering is too large for an exhaustive search, many heuristics have been proposed to explore the space by using machine learning. Huang *et al.*

tried to address this challenge for HLS applications by using modified greedy algorithms [89,90]. It achieved 16% improvement vs. -O3 on the CHstone benchmarks [77], which we used in this paper. In [3] both independent and Markov models were applied to automatically target an optimized search space for iterative methods to improve the search results. In [187], genetic algorithms were used to tune heuristic priority functions for three compiler optimization passes. Milepost GCC [59] used machine learning to determine the set of passes to apply to a given program, based on a static analysis of its features. It achieved an 11% execution time improvement over -O3, for the ARC reconfigurable processor on the MiBench program suite1. In [111] the challenge was formulated as a Markov process, and supervised learning was used to predict the next optimization based on the current program state. OpenTuner [7] autotunes a program using an AUC-Bandit-meta-technique-directed ensemble selection of algorithms. Its current mechanism for selecting the compiler optimization passes does not consider the order or support repeated optimizations. Wang *et al.* [201], provided a survey for using machine learning in compiler optimization where they also described that using program features might be helpful. NeuroVectorizer [69, 70] used deep RL for automatically tuning compiler pragmas such as vectorization and interleaving factors. NeuroVectorizer achieves 97% of the oracle performance (brute-force search) on a wide range of benchmarks.

5.2.2 Reinforcement Learning Algorithms

Reinforcement learning (RL) is a machine learning approach in which an agent continually interacts with the environment [99]. In particular, the agent observes the state of the environment, and based on this observation, takes an action. The goal of the RL agent is then to compute a policy–a mapping between the environment states and actions–that maximizes a long-term reward.

RL can be viewed as a stochastic optimization solution for solving Markov Decision Processes (MDPs) [15], when the MDP is not known. An MDP is defined by a tuple with four elements: $S, A, P(s,a), r(s,a)$ where S is the set of states of the environment, A describes the set of actions or transitions between states, $s' \sim P(s,a)$ describes the probability distribution of next states given the current state and action and $r(s,a) : S \times A \rightarrow R$ is the reward of taking action a in state s. Given an MDP, the goal of the agent is to gain the largest possible aggregate reward. The objective of an RL algorithm associated with an MDP is to find a decision policy $\pi^*(a|s) : s \rightarrow A$ that achieves this goal for that MDP:

$$\begin{aligned} \pi^* &= \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi(\tau)} \left[\sum_t r(s_t, a_t) \right] \\ &= \arg\max_{\pi} \sum_{t=1}^{T} \mathbb{E}_{(s_t,a_t)\sim\pi(s_t,a_t)} \left[r(s_t, a_t) \right] \end{aligned} \tag{5.1}$$

Deep RL leverages a neural network to learn the policy (and sometimes the reward function). Policy Gradient (PG) [191], for example, updates the policy directly by differentiating

the aggregate reward $\mathbb{E}$ in Equation 5.1:

$$\begin{aligned}\nabla_\theta J &= \nabla_\theta \mathbb{E}_{\tau\sim\rho_{\pi(\tau)}}\left[\sum_t r(s_t,a_t)\right] \\ &= \mathbb{E}_{\tau\sim\rho_{\pi(\tau)}}\left[(\sum_t \nabla_\theta log\pi_\theta(a_t|s_t))(\sum_t r(s_t,a_t))\right] \qquad (5.2)\\ &\approx \frac{1}{N}\sum_{i=1}^{N}\left[(\sum_t \nabla_\theta log\pi_\theta(a_{i,t}|s_{i,t}))(\sum_t r(s_{i,t},a_{i,t}))\right]\end{aligned}$$

and updating the network parameters (weights) in the direction of the gradient:

$$\theta \leftarrow \theta + \alpha\nabla_\theta J, \qquad (5.3)$$

Note that PG is an on-policy method in that it uses decisions made directly by the current policy to compute the new policy.

Over the past couple of years, a plethora of new deep RL techniques have been proposed [133, 169]. In this paper, we mainly focus on Proximal Policy Optimization (PPO) [172], Asynchronous Advantage Actor-critic (A3C) [133].

PPO is a variant of PG that enables multiple epochs of minibatch updates to improve the sample complexity. Vanilla PG performs one gradient update per data sample while PPO uses a novel surrogate objective function to enable multiple epochs of minibatch updates. It alternates between sampling data through interaction with the environment and optimizing the surrogate objective function using stochastic gradient ascent. It performs updates that maximizes the reward function while ensuring the deviation from the previous policy is small by using a surrogate objective function. The loss function of PPO is defined as:

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)] \qquad (5.4)$$

where $r_t(\theta)$ is defined as a probability ratio $\frac{\pi_\theta(\mathbf{a_t}|\mathbf{s_t})}{\pi_{\theta_{old}}(\mathbf{a_t}|\mathbf{s_t})}$ so $r(\theta_{old}) = 1$. This term penalizes policy update that move $r_t(\theta)$ from $r(\theta_{old})$. $\hat{A}_t$ denotes the estimated advantage that approximates how good $\mathbf{a_t}$ is compared to the average. The second term in the *min* function acts as a disincentive for moving r_t outside of $[1-\varepsilon, 1+\varepsilon]$ where ε is a hyperparameter.

A3C uses an actor (usually a neural network) that interacts with the critic, which is another network that evaluates the action by computing the value function. The critic tells the actor how good its action was and how it should adjust. The update performed by the algorithm can be seen as $\nabla_\theta log\pi_\theta(a_{i,t}|s_{i,t})\hat{A}_t$.

5.2.3 Evolutionary Algorithms

Evolutionary algorithms are another technique that can be used to search for the best compiler pass ordering. It contains a family of population-based meta-heuristic optimization algorithms

inspired by natural selection. The main idea of these algorithms is to sample a population of solutions and use the good ones to direct the distribution of future generations. Two commonly used Evolutionary Algorithms are Genetic Algorithms (GA) [85] and Evolution Strategies (ES) [43].

GA generally requires a genetic representation of the search space where the solutions are coded as integer vectors. The algorithm starts with a pool of candidates, then iteratively evolves the pool to include solutions with higher fitness by the three following strategies: selection, crossover, and mutation. Selection keeps a subset of solutions with the highest fitness values. These selected solutions act as parents for the next generation. Crossover merges pairs from the parent solutions to produce new offsprings. Mutation perturbs the offspring's solutions with a low probability. The process repeats until a solution that reaches the goal fitness is found or after a certain number of generations.

ES works similarly to GA. However, the solutions are coded as real numbers in ES. In addition, ES is self-adapting. The hyperparameters, such as the step size or the mutation probability, are different for different solutions. They are encoded in each solution, so good settings get to the next generation with good solutions. Recent work [170] has used ES to update policy weights for RL and showed it is a good alternative for gradient-based methods.

5.3 AutoPhase Framework for Automatic Phase Ordering

We leverage an existing open-source HLS framework called LegUp [27] that compiles a C program into a hardware RTL design. In [89], an approach is devised to quickly determine the number of hardware execution cycles without requiring time-consuming logic simulation. We develop our RL simulator environment based on the existing harness provided by LegUp and validate our final results by going through the time-consuming logic simulation. AutoPhase takes a program (or multiple programs) and intelligently explores the space of possible passes to figure out an optimal pass sequence to apply. Table 5.1 lists all the passes used in AutoPhase. The workflow of AutoPhase is illustrated in Figure 5.4.

5.3.1 HLS Compiler

AutoPhase takes a set of programs as input and compiles them to a hardware-independent intermediate representation (IR) using the Clang front-end of the LLVM compiler. Optimization and analysis passes act as transformations on the IR, taking a program as input and emitting a new IR as output. The HLS tool LegUp is invoked after the compiler optimization as a backend pass, which transforms LLVM IR into hardware modules.

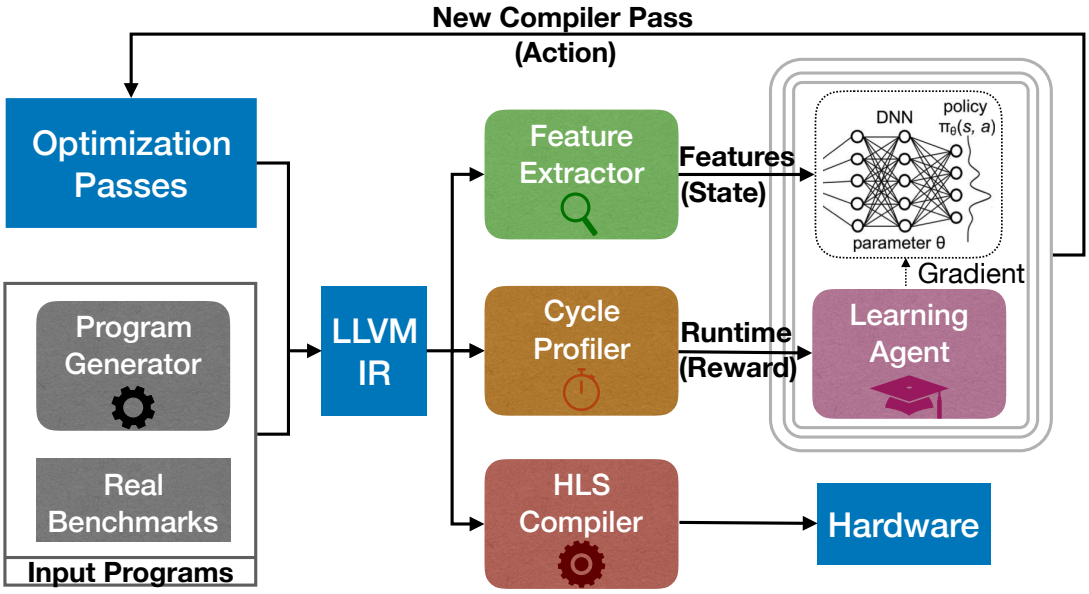


Figure 5.4: The block diagram of AutoPhase. The input programs are compiled to an LLVM IR using Clang/LLVM. The feature extractor and clock-cycle profiler are used to generate the input features (state) and the runtime improvement (reward), respectively, from the IR. The program features and runtime improvement are fed to the deep RL agent as input data to train on. The RL agent predicts the next best optimization passes to apply. After convergence, the HLS compiler is used to compile the LLVM IR to hardware RTL.

5.3.2 Clock-cycle Profiler

Once the hardware RTL is generated, one could run a hardware simulation to gather the cycle count results of the synthesized circuit. This process is quite time-consuming, hindering RL and all other optimization approaches. Therefore, we approximate cycle count using the profiler in LegUp [89], which leverages the software traces and runs 20× faster than hardware simulation. In LegUp, the frequency of the generated circuits is set as a compiler constraint that directs the HLS scheduling algorithm. In other words, the HLS tool will always try to generate hardware that can run at a certain frequency. In our experiment setting, without loss of generality, we set the target frequency of all generated hardware to 200MHz. We experimented with lower frequencies, too; the improvements were similar, but the cycle counts the different algorithms achieved were better as more logic could be fitted in a single cycle.

5.3.3 IR Feature Extractor

Wang *et al.* [201] proposed to convert a program into observation by extracting all the features from the program. Similarly, in addition to the LegUp backend tools, we developed analysis

passes to extract 56 static features from the program, such as the number of basic blocks, branches, and instructions of various types. We use these features as partially observable states for learning. We hope the neural network can capture the correlation between certain combinations of these features and certain optimizations. Table 5.2 lists all the features used.

5.3.4 Random Program Generator

As a data-driven approach, RL generalizes better if we train the agent on more programs. However, there are a limited number of open-source HLS examples online. Therefore, we expand our training set by automatically generating synthetic HLS benchmarks. We first generate standard C programs using CSmith [217], a random C program generator, which is originally designed to generate test cases for finding compiler bugs. Then, we develop scripts to filter out programs that take more than five minutes to run on the CPU or fail the HLS compilation.

5.3.5 Overall Flow of AutoPhase

We integrate the compilation utilities into a simulation environment in Python with APIs similar to an OpenAI gym [25]. The overall flow works as follows:

1. The input program is compiled into LLVM IR using the Clang/LLVM.
2. The IR Feature Extractor is run to extract salient program features.
3. LegUp compiles the LLVM IR into hardware RTL.
4. The Clock-cycle Profiler estimates a clock-cycle count for the generated circuit.
5. The RL agent takes the program features or the histogram of previously applied passes and the improvement in clock-cycle count as input data to train on.
6. The RL agent predicts the next best optimization passes to apply.
7. New LLVM IR is generated after the new optimization sequence is applied.
8. The machine learning algorithm iterates through steps (2)–(7) until convergence.

Note that AutoPhase uses the LLVM compiler and the passes used are listed in Table 5.2. However, adding support for any compiler or optimization passes in AutoPhase is very easy and straightforward. The action and state definitions must be specified again.

5.4 Correlation of Passes and Program Features

Similar to the case with many deep learning approaches, explainability is one of the major challenges we face when applying deep RL to the phase-ordering challenge. To analyze and understand the correlation of passes and program features, we use random forests [23] to learn the importance of different features. Random forest is an ensemble of multiple decision

Table 5.1: LLVM Transform Passes.

0	1	2	3	4	5	6	7	8	9	10
-correlated-propagation	-scalarrepl	-lowerinvoke	-strip	-strip-nondebug	-sccp	-globalopt	-gvn	-jump-threading	-globaldce	-loop-unswitch
11	12	13	14	15	16	17	18	19	20	21
-scalarrepl-ssa	-loop-reduce	-break-crit-edges	-loop-deletion	-reassociate	-lcssa	-codegenprepare	-memcpyopt	-functionattrs	-loop-idiom	-lowerswitch
22	23	24	25	26	27	28	29	30	31	32	33
-constmerge	-loop-rotate	-partial-inliner	-inline	-early-cse	-indvars	-adce	-loop-simplify	-instcombine	-simplifycfg	-dse	-loop-unroll
34	35	36	37	38	39	40	41	42	43	44	45
-lower-expect	-tailcallelim	-licm	-sink	-mem2reg	-prune-eh	-functionattrs	-ipsccp	-deadargelim	-sroa	-loweratomic	-terminate

Table 5.2: Program Features.

0	Number of BB where total args for phi nodes >5	28	Number of And insts
1	Number of BB where total args for phi nodes is [1,5]	29	Number of BB's with instructions between [15,500]
2	Number of BB's with 1 predecessor	30	Number of BB's with less than 15 instructions
3	Number of BB's with 1 predecessor and 1 successor	31	Number of BitCast insts
4	Number of BB's with 1 predecessor and 2 successors	32	Number of Br insts
5	Number of BB's with 1 successor	33	Number of Call insts
6	Number of BB's with 2 predecessors	34	Number of GetElementPtr insts
7	Number of BB's with 2 predecessors and 1 successor	35	Number of ICmp insts
8	Number of BB's with 2 predecessors and successors	36	Number of LShr insts
9	Number of BB's with 2 successors	37	Number of Load insts
10	Number of BB's with >2 predecessors	38	Number of Mul insts
11	Number of BB's with Phi node # in range (0,3]	39	Number of Or insts
12	Number of BB's with more than 3 Phi nodes	40	Number of PHI insts
13	Number of BB's with no Phi nodes	41	Number of Ret insts
14	Number of Phi-nodes at beginning of BB	42	Number of SExt insts
15	Number of branches	43	Number of Select insts
16	Number of calls that return an int	44	Number of Shl insts
17	Number of critical edges	45	Number of Store insts
18	Number of edges	46	Number of Sub insts
19	Number of occurrences of 32-bit integer constants	47	Number of Trunc insts
20	Number of occurrences of 64-bit integer constants	48	Number of Xor insts
21	Number of occurrences of constant 0	49	Number of ZExt insts
22	Number of occurrences of constant 1	50	Number of basic blocks
23	Number of unconditional branches	51	Number of instructions (of all types)
24	Number of Binary operations with a constant operand	52	Number of memory instructions
25	Number of AShr insts	53	Number of non-external functions
26	Number of Add insts	54	Total arguments to Phi nodes
27	Number of Alloca insts	55	Number of Unary operations

trees. The prediction made by each tree could be explained by tracing the decisions made at each node and calculating the importance of different features on making the decisions at each node. This helps us to identify the effective features and passes to use and show whether our algorithms learn informative patterns on data.

For each pass, we build two random forests to predict whether applying it would improve the circuit performance. The first forest takes the program features as inputs, while the second takes a histogram of previously applied passes. To gather the training data for the forests, we run PPO with a high exploration parameter value on 100 randomly generated programs to generate feature–action–reward tuples. The algorithm assigns higher importance

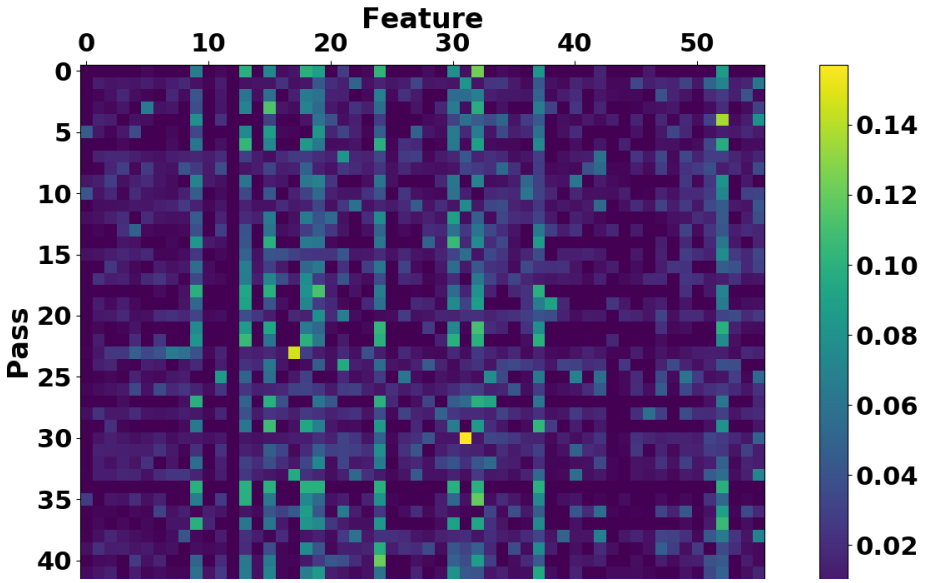


Figure 5.5: Heat map illustrating the importance of feature and pass indices.

to the input features that affect the final prediction more.

5.4.1 Importance of Program Features

The heat map in Figure 5.5 shows the importance of different features on whether a pass should be applied. The higher the value is, the more important the feature is (the sum of the values in each row is one). The random forest is trained with 150,000 samples generated from the random programs. The index mapping of features and passes can be found in Tables 5.1 and 5.2. For example, the yellow pixel corresponding to feature index 17 and pass index 23 reflects that *number-of-critical-edges* affects the decision on whether to apply *-loop-rotate* greatly. A critical edge in the control flow graph is an edge that is neither the only edge leaving its source block, nor the only edge entering its destination block. The critical edges can be commonly seen in a loop as a back edge, so the number of critical edges might roughly represent the number of loops in a program. The transform pass *-loop-rotate* detects a loop and transforms a while loop to a do-while loop to eliminate one branch instruction in the loop body. Applying the pass results in better circuit performance as it reduces the total number of FSM states in a loop.

Other expected behaviors are also observed in this figure. For instance, the correlation between *number of branches* and the transform passes *-loop-simplify*, *-tailcallelism* (which transforms calls of the current function *i.e.*, self recursion, followed by a return instruction with a branch to the entry of the function, creating a loop), *-lowerswitch* (which rewrites switch instructions with a sequence of branches). Other interesting behaviors are also captured. For example, in the correlation between *binary operations with a constant operand*

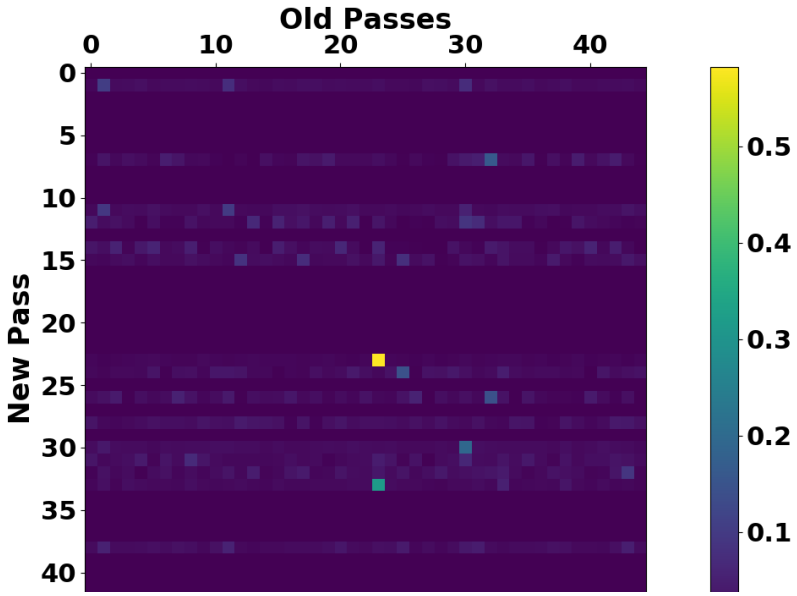


Figure 5.6: Heat map illustrating the importance of indices of previously applied passes and the new pass to apply.

and *-functionattrs*, which marks different operands of a function as read-only (constant). Some correlations are harder to explain, for example, *number of BitCast instructions* and *-instcombine*, which combines instructions into fewer simpler instructions. This is actually a result of *-instcombine* reducing the loads and stores that call bitcast instructions for casting pointer types. Another example is *number of memory instructions* and *-sink*, where *-sink* basically moves memory instructions into successor blocks and delays the execution of memory until needed. Intuitively, whether to apply *-sink* should be dependent on whether there is any memory instruction in the program. Our last example to show is *number of occurrences of constant 0* and *-deadargelim*, where *-deadargelim* helped eliminate dead/unused constant zero arguments.

Overall, we observe that all the passes are correlated to some features and are able to affect the final circuit performance. We also observe that multiple features are not effective at directing decisions, and training with them could increase the variance that would result in lower prediction accuracy of our results. For example, the total number of instructions did not directly indicate whether applying a pass would be helpful or not. This is because sometimes more instructions could improve the performance (for example, due to loop unrolling), and eliminating unnecessary code could also improve the performance. In addition, the importance of features varies among different benchmarks depending on the tasks they perform.

5.4.2 Importance of Previously Applied Passes

Figure 5.6 illustrates the impact of previously applied passes on the new pass to apply. The higher the value is, the more important having the old pass is. From this figure, we learn that for the programs we trained on passes *-scalarrepl, -gvn, -scalarrepl-ssa, -loop-reduce, -loop-deletion, -reassociate, -loop-rotate, -partial-inliner, -early-cse, -adce, -instcombine, -simplifycfg, -dse, -loop-unroll, -mem2reg, and -sroa*, are more impactful on the performance compared to the rest of the passes regardless of their order in the trajectory. Point (23,23) has the highest importance in which implies that pass *-loop-rotate* is very helpful and should be included if not applied before. By examining thousands of the programs, we find that *-loop-rotate* indeed reduces the cycle count significantly. Interestingly, applying this pass twice is not harmful if the passes were given consecutively. However, giving this pass twice with some other passes between them is sometimes very harmful. Another interesting behavior our heat map captured is the fact that applying pass 33 (*-loop-unroll*) after (not necessarily consecutive) pass 23 (*-loop-rotate*) was much more useful compared to applying these two passes in the opposite order.

5.5 Problem Formulation

5.5.1 The RL Environment Definition

Assume the optimal number of passes to apply is N and there are K transform passes to select from in total, our search space $\mathcal{S}$ for the phase-ordering problem is $[0, K^N)$. Given M program features and the history of already applied passes, the goal of deep RL is to learn the next best optimization pass a to apply that minimizes the long term cycle count of the generated hardware circuit. Note that the optimization state s is partially observable in this case as the M program features cannot fully capture all the properties of a program.

Action Space – we define our action space $\mathcal{A}$ as $\{a \in \mathbb{Z} : a \in [0, K)\}$ where K is the total number of transform passes.

Observation Space – two types of input features were considered in our evaluation: ① **program features** $\mathbf{o_f} \in \mathbb{Z}^M$ listed in Table 5.2 and ② **action history** which is a histogram of previously applied passes $\mathbf{o_a} \in \mathbb{Z}^K$. After each RL step where the pass i is applied, we call the feature extractor in our environment to return new $\mathbf{o_f}$, and update the action histogram element o_{a_i} to $o_{a_i} + 1$.

Reward – the cycle count of the generated circuit is reported by the clock-cycle profiler at each RL iteration. Our reward is defined as $R = c_{prev} - c_{cur}$, where c_{prev} and c_{cur} represent the previous and the current cycle count of the generated circuit respectively. It is possible to define a different reward for different objectives. For example, the reward could be defined as the negative of the area, and thus the RL agent will optimize for the area. It is also possible to co-optimize multiple objectives (e.g., area, execution time, power, etc.) by defining a combination of different objectives.

5.5.2 Applying Multiple Passes per Action

An alternative to the action formulation above is to evaluate a complete sequence of passes with length N instead of a single action a at each RL iteration. Upon the start of training a new episode, the RL agent resets all pass indices $\mathbf{p} \in \mathbb{Z}^N$ to the index value $\frac{K}{2}$. For pass p_i at index i, the next action to take is either to change to a new pass or not. By allowing positive and negative index update for each p, we reduced the total steps required to traverse all possible pass indices. The sub-action space a_i for each pass is thus defined as $[-1, 0, 1]$. The total action space $\mathcal{A}$ is defined as $[-1, 0, 1]^N$. At each step, the RL agent predicts the updates $[a_1, a_2, ..., a_N]$ to N passes, and the current optimization sequence $[p_1, p_2, ..., p_N]$ is updated to $[p_1 + a_1, p_2 + a_2, ..., p_N + a_N]$.

5.5.3 Normalization Techniques

In order for the trained RL agent to work on new programs, we need to properly normalize the program features and rewards, so they represent a meaningful state among different programs. In this work, we experiment with two techniques: ① taking the logarithm of program features or rewards and, ② normalizing to a parameter from the original input program that roughly depicts the problem size. For technique ①, note that taking the logarithm of the program features not only reduces their magnitude, it also correlates them in a different manner in the neural network. Since, $w_1 \log(o_{f_1}) + w_2 \log(o_{f_2}) = log(o_{f_1}^{w_1} o_{f_2}^{w_2})$, the neural network is learning to correlate the products of features instead of a linear combination of them. For technique ②, we normalize the program features to the total number of instructions in the input program ($\mathbf{o_{f\_norm}} = \frac{\mathbf{o_f}}{o_{f_{51}}}$), which is feature #51 in Table 5.2.

5.6 Evaluation

To run our deep RL algorithms, we use RLlib [122], an open-source library for reinforcement learning that offers both high scalability and a unified API for a variety of applications. RLlib is built on top of Ray [134], a high-performance distributed execution framework targeted at large-scale machine learning and reinforcement learning applications. We ran the framework on a four-core Intel i7-4765T CPUwith a Tesla K20c GPUfor training and inference.

We set our frequency constraint in HLS to 200MHz and use the number of clock cycles reported by the HLS profiler as the circuit performance metric. In [89], results showed a one-to-one correspondence between the clock cycle count and the actual hardware execution time under a certain frequency constraint. Therefore, better clock cycle count will lead to better hardware performance.

5.6.1 Performance

To evaluate the effectiveness of various algorithms for tackling the phase-ordering problem, we run them on nine real HLS benchmarks and compare the results based on the final HLS circuit

Table 5.3: The observation and action spaces used in the different deep RL algorithms.

	RL-PPO1	RL-PPO2	RL-PPO3	RL-A3C	RL-ES
Algorithms	PPO	PPO	PPO	A3C	ES
Observation	Program Features	Action History	Action History + Program Features	Program Features	Program Features
Action	Single-Action	Single-Action	Multiple-Action	Single-Action	Single-Action

performance and the sample efficiency against state-of-the-art approaches for overcoming the phase ordering, which include random search, Greedy Algorithms [89], OpenTuner [7], and Genetic Algorithms [57]. These benchmarks are adapted from CHStone [77] and LegUp examples. They are: *adpcm*, *aes*, *blowfish*, *dhrystone*, *gsm*, *matmul*, *mpeg2*, *qsort*, and *sha*. For this evaluation, the input features/rewards were not normalized, the pass length was set to 45, and each algorithm was run on a per-program basis. Table 5.3 lists the action and observation spaces used in all the deep RL algorithms.

The bar chart in Figure 5.7 shows the percentage improvement of the circuit performance compared to -O3 results on the nine real benchmarks from CHStone. The dots on the blue line in Figure 5.7 show the total number of samples for each program, which is the number of times the algorithm calls the simulator to gather the cycle count. `-O0` and `-O3` are the default compiler optimization levels. `RL-PPO1` is a PPO explorer where we set all the rewards to 0 to test if the rewards are meaningful. `RL-PPO2` is the PPO agent that learns the next pass based on a histogram of applied passes. `RL-A3C` is the A3C agent that learns based on the program features. `Greedy` performs the greedy algorithm, which always inserts the pass that achieves the highest speedup at the best position (out of all possible positions it can be inserted to) in the current sequence. `RL-PPO3` uses a PPO agent and the program features but with the action space described in Section 5.5.2. explained in Section 5.5.2. `OpenTuner` runs an ensemble of six algorithms, which includes two families of algorithms: particle swarm optimization [103] and GA, each with three different crossover settings. `RL-ES` is similar to A3C agent that learns based on the program features, but updates the policy network using the evolution strategy instead of backpropagation. `Genetic-DEAP` [57] is a genetic algorithm implementation. `random` randomly generates a sequence of 45 passes at once instead of sampling them one-by-one.

From `Greedy`, we see that always adding the pass in the current sequence that achieves the highest reward leads to sub-optimal circuit performance. `RL-PPO2` achieves higher performance than `RL-PPO1`, which shows that the deep RL captures useful information during training. Using the histogram of applied passes results in better sample efficiency, but using the program features with more samples results in a slightly higher speedup. `RL-PPO2`, for example, at the minor cost of 4% lower speedup, achieves 50× more sample efficiency than `OpenTuner`. Using ES to update the policy is supposed to be more sample efficient for problems with sparse rewards like ours; however, our experiments did not benefit from that. Furthermore, `RL-PPO3` with multiple action updates achieves a higher speedup than the other deep RL algorithms with a single action. One reason for that is the ability of `RL-PPO3` to explore more passes per compilation as it applies multiple passes simultaneously in between every compilation. On

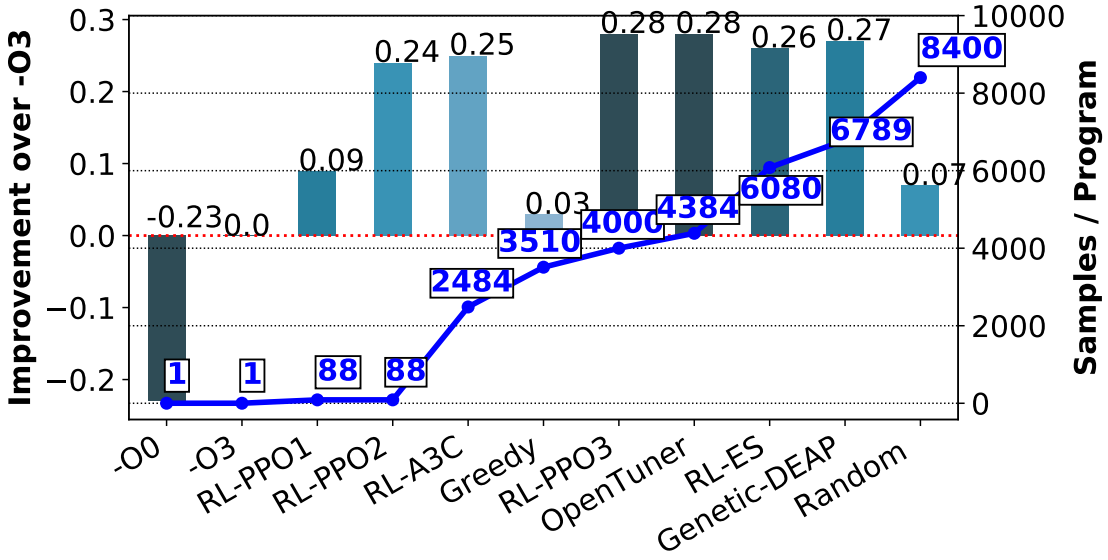


Figure 5.7: Circuit Speedup and Sample Size Comparison.

the other hand, the other deep RL algorithms apply a single pass at a time.

5.6.2 Generalization

With deep RL, the search should benefit from prior knowledge learned from other different programs. This knowledge should be transferable from one program to another. For example, as discussed in section 5.4 applying pass *-loop-rotate* is always beneficial, and *-loop-unroll* should be applied after *-loop-rotate*. Note that the black-box search algorithms, such as OpenTuner, GA, and greedy algorithms, cannot generalize. For these algorithms, rerunning a new search with many compilations is necessary for every new program, as they do not learn any patterns from the programs to direct the search and can be viewed as a smart random search.

To evaluate how generalizable deep RL could be with different programs and whether any prior knowledge could be useful, we train on 100 randomly generated programs using PPO. Random programs are used for transfer learning due to lack of sufficient benchmarks, and because it is the worst-case scenario, *i.e.*, they are very different from the programs that we use for inference. The improvement can be higher if we train on programs similar to the ones we run inference on. We train a network with 256×256 fully connected layers and use the histogram of previously applied passes concatenated to the program features as the observation and passes as actions.

As described in Section 5.5.3, we experiment with two normalization techniques for the program features: ① taking the logarithm of all the program features and ② normalizing the program features to the total number of instructions in the program. In each pass sequence, the intermediate reward was defined as the logarithm of the improvement in cycle count after

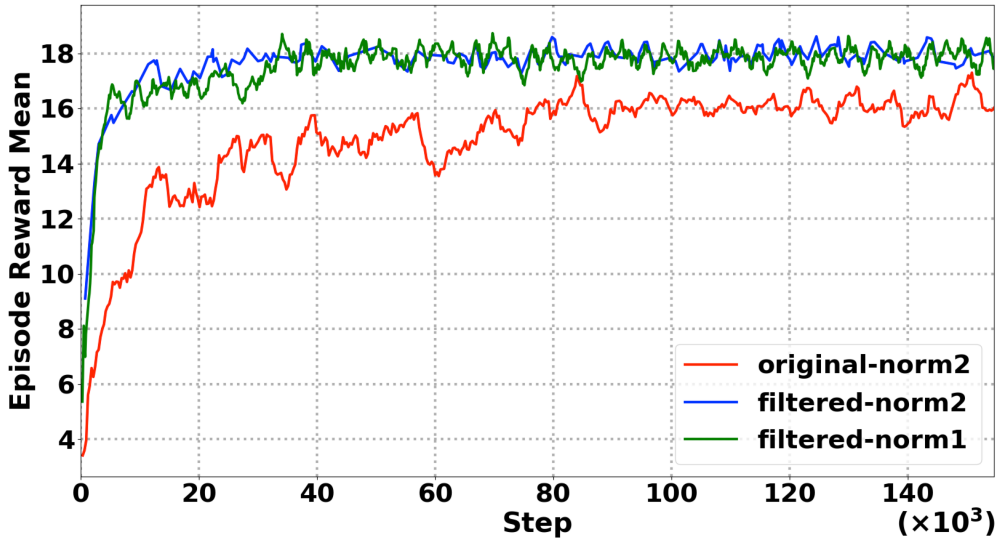


Figure 5.8: Episode reward mean as a function of step for the original approach where we use all the program features and passes and for the filtered approach where we filter the passes and features (with different normalization techniques). Higher values indicate faster circuit speed.

applying each pass. The logarithm was chosen so that the RL agent will not give much larger weights to big rewards from programs with longer execution time. Three approaches were evaluated: `filtered-norm1` uses the filtered (based on the analysis in Section 5.4 where we only keep the important features and passes) program features and passes from Section with normalization technique ①, `original-norm2` uses all the program features and passes with normalization technique ②, and `filtered-norm2` uses the filtered program features and passes from Section 5.4 with normalization technique ②. Filtering the features and passes might not be ideal, especially when different programs have different feature characteristics and impactful passes. However, reducing the number of features and passes helps to reduce variance among all programs and significantly narrow the search space.

Figure 5.8 shows the episode reward mean as a function of the step for the three approaches. We observe that `filtered-norm2` and `filtered-norm1` converge much faster and achieve a higher episode reward mean than `original-norm2`, which uses all the features and passes. At roughly 8,000 steps the `filtered-norm2` and `filter-norm1` already achieve a very high episode reward mean, with minor improvements in later steps. Furthermore, the episode reward mean of the filtered approaches is still higher than that of `original-norm2` even when we allowed it to train for 20 times more steps (*i.e.*, 160,000 steps). This indicates that filtering the features and passes significantly improved the learning process. All three approaches learned to always apply pass *-loop-rotate*, and *-loop-unroll* after *-loop-rotate*. Another useful pass that the three approaches learned to apply is *-loop-simplify*, which performs several transformations to transform natural loops into a simpler form that enables subsequent

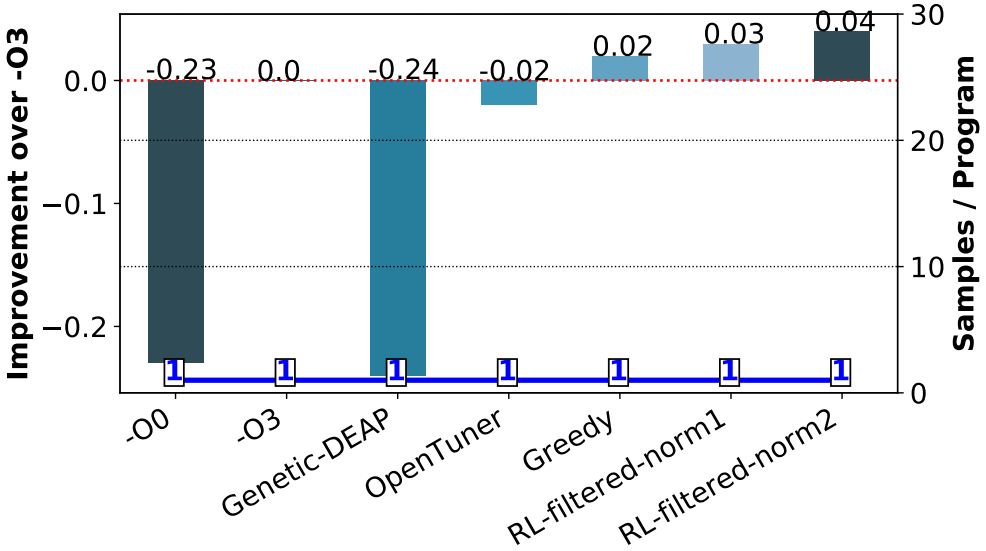


Figure 5.9: Circuit Speedup and Sample Size Comparison for deep RL Generalization.

analyses and transformations.

We now compare the generalization results of `filtered-norm2` and `filtered-norm1` with the other black-box algorithms. We use 100 randomly generated programs as the training set and nine real benchmarks from CHStone as the testing set for the deep RL-based methods. With the state-of-the-art black-box algorithms, we first search for the best pass sequences that achieved the lowest aggregated hardware cycle counts for the 100 random programs and then directly apply them to the nine test set programs. In Figure 5.9, the bar chart shows the percentage improvement of the circuit performance compared to -O3 on the nine real benchmarks, the dots on the blue line show the total number of samples each inference takes for one new program.

This evaluation shows that the deep RL-based inference achieves higher speedup than the predetermined sequences produced by the state-of-the-art black-box algorithms for new programs. The predetermined sequences that are overfitted to the random programs can cause poor performance in unseen programs (*e.g.*, -24% for `Genetic-DEAP`). Besides, normalization technique ② works better compared to normalization technique① for deep RL generalization (4% vs 3% speedup). This indicates that normalizing the different instructions to the total number of instructions *i.e.*, the distribution of the different instructions in Technique ② represents more universal characteristics across different programs, while taking the log in Technique ① only suppresses the value ranges of different program features. Furthermore, when we use other 12,874 randomly generated programs as the testing set with `filtered-norm2`, the speedup is 6% compared to -O3.

5.7 Conclusions

In AutoPhase, we introduce an approach based on deep RL to improve the performance of HLS designs by optimizing the order in which the compiler applies optimization phases. We use random forests to analyze the relationship between program features and optimization passes. We then leverage this relationship to reduce the search space by identifying the most likely optimization phases to improve the performance, given the program features. Our RL-based approach achieves 28% better performance than compiling with the -O3 flag after training for a few minutes, and a 24% improvement after training for less than a minute. Furthermore, we show that, unlike prior work, our solution shows the potential to generalize to a variety of programs. While in this paper we have applied deep RL to HLS, we believe that the same approach can be successfully applied to software compilation and optimization.

As we advance, we harness RL and ML techniques to tackle a broader range of hardware and systems problems, including Halide scheduling [71], Analog circuit sizing [173], and verification [91].

Chapter 6

Discussion and Future Work

This dissertation describes the three-pronged co-design approach we employed to tackle the research question we asked in the Introduction (Chapter 1): "*How to develop the most efficient accelerator systems for deep learning in a timely and cost-effective manner?*". This thesis demonstrates that the synergy among algorithm, software, and hardware in co-designing deep learning accelerator systems has enabled substantially more optimization opportunities to improve the end-to-end system (Chapter 2-4). Furthermore, we harness the advancements in machine learning to help solve for NP-hard optimization problems in hardware design and compiler transforms (Chapter 5). While the co-design methodologies demonstrated in this thesis make incremental progress towards the development of state-of-the-art accelerator systems, there are still many opportunities to enhance in the modern-day systems to be further enhanced. Aside from human heuristics, more advanced machine learning and optimization algorithms for automatic design space exploration deserve more study, especially for handling the discrete and more intractable co-design space.

6.1 Discussion

This section discusses various insights and lessons learned from the thesis.

6.1.1 Co-design of algorithm, software, and hardware

The full potential of hardware acceleration can be realized through a holistic examination of the end-to-end system and innovations at different levels of the acceleration stack. The thesis covers multiple works that consider the algorithm, software, and hardware together in the design process to achieve better overall performance. There are three lessons learned from this research.

First, it is an effective tactic to start the co-design optimization by pinpointing the performance bottleneck. Although ideally, we should consider all possible system components in the search space, it is infeasible in real scenarios due to various resource and time constraints.

For instance, given a compute-bound 3x3 convolution workload on an FPGA accelerator with PE utilization of 100%, updating the algorithm operation and quantization to allow for more computing resources is preferred over changing the mapping algorithm for better reuse.

Second, modification to different components in co-design presents different costs and benefits. Assuming we can reach the same accelerator speedup by modifying only one out of the three key components, changing the algorithm and software would be faster and less risky than changing the hardware. The hardware design iteration is considerably more protracted than the one for software and the algorithm. In addition, once the accelerator design is sent for tape-out, it becomes difficult to fix any bugs in hardware while it remains feasible to update software and algorithm designs. However, if the benefits of a hardware change outweigh its development costs, or the hardware is easily programmable, as in the case of FPGA, there is no reason not to co-optimize hardware with the rest of the system.

Third, one practical strategy adopted in the thesis for rapidly finding performant solutions in the co-design space is to prune the space with explicit system constraints and data-driven analysis. The co-design exposes us to a significantly larger design space to explore. However, many existing machine learning and optimization algorithms do not scale with the number of variables or the variables' dimensions. Pruning the search space helps to reduce the number of samples required for an exhaustive search or even the number of variable dimensions.

6.1.2 Automatic Design and Verification Methodology

Design automation for hardware is a rich and vibrant research space. There exist a number of NP-hard problems in the automation of hardware design and verification, such as resource allocation, scheduling, logic synthesis, placement and routing, etc. As briefly mentioned in Chapter 2, one critical unsolved hurdle in design automation such as High-Level Synthesis (HLS) is the programming model and language abstraction. A fully automatic design flow demands a language abstraction that is sufficiently powerful to describe the target application yet easy to use, meanwhile comprehensively exposes potential optimizations.

We have investigated various programming paradigms ranging from binary, imperative, objective-oriented, to functional. There are several insights drawn from these attempts. First, witnessing the rising popularity of Python, I learn a rule of thumb for designing a good hardware language is to keep the abstractions simple and let the compiler and runtime automatically handle the error-prone and repetitive details. In addition, the evaluation of various properties of abstraction is objective, but the ease of use metric can be very subjective. The personal preference for different hardware languages is strongly influenced by education and past programming experiences. For example, some programmers find it natural to specify loops in recursion as in functional languages, but others don't. Lastly, domain-specific languages (DSL) like Halide [161] are a promising direction as they allow users to specify the desired behaviors in fewer lines of code and ease the analysis needed from the compiler to enable optimization passes. Furthermore, such abstraction has explicitly exposed optimization opportunities at different levels to the user and the compiler. This

property makes it possible to generate high-performance mapping onto various platforms in DSL.

We also observe two main obstacles to the adoption of novel HLS and HDL abstraction. One is the high adoption cost. Many companies have in-house legacy code and scripts that are incompatible with the new languages. However, rewriting the codebase can incur very high overheads. The second obstacle is the lack of comprehensive verification support in the new language ecosystem. Since verification accounts for more than half of the hardware development cycle (∼56% project time according to a recent study [180]), the hardware language should be designed with verification in mind.

6.1.3 Machine Learning and Optimization for Hardware

Although ML for everything has become a hot topic nowadays, we argue that ML might not be universally suitable for all problems. From our experience with AutoPhase (Chapter 5), we find that machine learning, albeit its unprecedented performance on certain tasks, has its own limitations. First, the training of ML, in particular DL and RL, requires much data to converge. The overall training process would be extremely costly if obtaining feedback or labels is very time-consuming. Even if we have enough resources to gather enough data, ML might not be the most efficient algorithm to use for optimization. We should consider expressing more task information such as constraints and objectives in mathematical format and see if the problem can be solved using standard mathematical programming. Intuitively, it should be more efficient for the user to specify specific constraints or properties for optimization than learning it from data using ML. Besides, the existing ML model design might not be powerful enough to encode all the structural or inductional information of input, potentially leading to a sub-optimal solution in the search.

Mathematical optimization, on the other hand, also has its drawbacks. An obvious one is that not all problems can be modeled within the solvable mathematical optimization paradigms. Linear programming, for instance, requires all constraints and objectives to be linear. Sometimes, we have to relax and approximate requirements to enable the optimization solver for complex problems, which could result in incorrect or unoptimized solutions.

Ideally, we want to apply ML to transfer the learned heuristics for optimization from task to task and apply mathematical optimization techniques to solve well-defined subproblems when feasible.

6.2 Future Work

For future research, I think there are three challenges and opportunities that are worth pursuing. The first is to examine more applications and exploit the co-design opportunities in them. The next unaddressed challenge is the proper abstraction for programming the heterogeneous systems that are more and more prevailing. Finally, it is critical to look into universally required machine learning and optimization techniques.

6.2.1 Co-Design for Broader Applications

With the pervasive needs for perception, recognition, and action at the edge, and the compute-hungry workloads on the cloud, more opportunities for acceleration are emerging. This thesis mainly focuses on deep learning inferencing. There are lots of workloads with high compute demand that we can apply our co-design methodology to accelerate. An immediate and promising extension to this thesis is AI training. With the prevalence of IoT devices and higher connectivity among them, edge devices are likely to become "smarter" and undertake more compute-hungry perception and learning tasks. However, due to the concerns of long turnaround latency, privacy, and security, future AI training is likely to be a combination of large-scale pretraining remote in the datacenter and efficient fine-tuning and adaptation in the local edge devices. In the near future, it is thus critical to enable more efficient data collection and deep learning training capability on the edge platforms for real-time adaptation.

6.2.2 Programming Abstraction for Heterogeneous Systems

With the prevalence of accelerators in the commodity computing platforms at scale, the programming abstraction is pivotal to user productivity and system performance. Given the complexity of modern applications and the underlying heterogeneous systems, an expressive yet simple concurrency abstraction is needed. Based on our previous study, Go [52] could be a good candidate for describing concurrency in both software and hardware. Its CSP concurrency model defines clear boundaries among sequential subroutines running on different hardware, allowing for flexible workload partitioning among accelerators and processors. Some interesting future directions include: How to partition workload to various resources? How to place different workloads? How to route the different traffics?

6.2.3 Machine Learning and Optimization for Systems

As mentioned in the previous sections, many NP-hard problems exist in hardware design automation and heterogeneous systems scheduling. Designing machine learning and optimization techniques that can efficiently navigate the large search space would be critical to the success of many other fields. It is worth looking into the following three research directions: 1) scalable machine learning algorithms for sequential decisions with large action and state space; for instance, how to formulate specific applications to reduce the action and state space? 2) methods to integrate heuristics and traditional optimization techniques such as linear programming with machine learning to reduce the solution space and improve sample efficiency, and 3) effective featurization and encoding of target problems to enable transfer learning.

6.3 Closing Remarks

There are three key aspects of graduate schools I benefited from: collaboration, exploration, and focus. I believe collaboration is key to innovation and productivity. Through my collaborators specialized in different research domains, I obtained extensive knowledge and deep insights that I could not learn from the textbooks. Together, my collaborators and I have accomplished several engineering-heavy projects that would have been impossible to finish independently. More importantly, interacting with people is beneficial to our mental health since we are social creatures. Exploration driven by curiosity has also played an essential role in my graduate school journey. Without the urge to explore more about the field we study, I would not have taken many seemingly unrelated courses and done internships at various amazing companies. Exploration helps me to broaden my horizons and gain more novel ideas in research. Meanwhile, focus became more critical towards the end of the graduate study. It allows me to allocate resources towards my goal wisely.

Overall, going through graduation study is similar to training an reinforcement learning agent. I first need to perform explorations to cover enough research space to find feasible regions with promising rewards. I then need to exploit these regions with clear directions to obtain higher rewards. As in RL, there are tradeoffs in exploration and exploitation. Finding the right balance in life and research is also key to success. In this process, I further leverage the knowledge transferred from other agents and their past experiences to facilitate my own learning. Unfortunately, randomness is something that cannot be avoided in the process and still impacts the outcome of each trial.

Finally, now it may seem reasonable to imagine RL agents getting their graduate school degrees in the future with more disruptive technology revolutions.

Bibliography

[1] Aravind Acharya, Uday Bondhugula, and Albert Cohen. Polyhedral auto-transformation with no integer linear programming. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michael Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 2019.

[3] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, John Thomson, Marc Toussaint, and Christopher KI Williams. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO)*, pages 295–305. IEEE Computer Society, 2006.

[4] Lelac Almagor, Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steven W Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. *ACM SIGPLAN Notices*, 2004.

[5] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. Search-based program synthesis. *Communications of the ACM*, 2018.

[6] Amazon. AWS Inferentia: High Performance Machine Learning Inference Chip. `https://aws.amazon.com/machine-learning/inferentia/`, 2018.

[7] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[8] Krste Asanović, Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The

Rocket Chip Generator. Technical Report UCB/EECS-2016-17, University of California, Berkeley, Apr 2016.

[9] Oren Avissar, Rajeev Barua, and Dave Stewart. Heterogeneous memory management for embedded systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2001.

[10] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C Ling, and Gordon R Chiu. An opencl deep learning accelerator on arria 10. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 2017.

[11] R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *International Symposium on Code Generation and Optimization (CGO)*, 2019.

[12] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven Van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2015.

[13] Hessam Bagherinezhad, Maxwell Horton, Mohammad Rastegari, and Ali Farhadi. Label refinery: Improving imagenet classification through label progression. *arXiv preprint arXiv:1805.02641*, 2018.

[14] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2006.

[15] Richard Bellman. A markovian decision process. In *Journal of Mathematics and Mechanics*, pages 679–684, 1957.

[16] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[17] David Biancolin, Sagar Karandikar, Donggyu Kim, Jack Koenig, Andrew Waterman, Jonathan Bachrach, and Krste Asanovic. FASED: Fpga-accelerated simulation and evaluation of dram. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 2019.

[18] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti,

et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[19] Michaela Blott, Thomas B Preußer, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks. In *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2018.

[20] Mariusz Bojarski, Philip Yeres, Anna Choromanska, Krzysztof Choromanski, Bernhard Firner, Lawrence Jackel, and Urs Muller. Explaining how a deep neural network trained with end-to-end learning steers a car. *arXiv preprint arXiv:1704.07911*, 2017.

[21] Uday Bondhugula, Aravind Acharya, and Albert Cohen. The pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2016.

[22] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[23] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[24] Benjamin Brock, Aydın Buluç, and Katherine Yelick. Bcl: A cross-platform distributed container library. *arXiv preprint arXiv:1810.13029*, 2018.

[25] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[26] Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Zeroq: A novel zero shot quantization framework. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 13169–13178, 2020.

[27] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D Brown, and Jason H Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):24, 2013.

[28] Prasanth Chatarasi, Hyoukjun Kwon, Natesh Raina, Saurabh Malik, Vaisakh Haridas, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. Marvel: A data-centric compiler for dnn operators on spatial accelerators. *arXiv preprint arXiv:2002.07752*, 2020.

[29] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[30] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, March 2014.

[31] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016.

[32] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2019.

[33] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DaDianNao: A Machine-learning Supercomputer. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014.

[34] Yuntao Chen, Chenxia Han, Yanghao Li, Zehao Huang, Yi Jiang, Naiyan Wang, and Zhaoxiang Zhang. Simpledet: A simple and versatile distributed framework for object detection and instance recognition. *The Journal of Machine Learning Research (JMLR)*, 2019.

[35] Shaoyi Cheng and John Wawrzynek. High level synthesis with a dataflow architectural template. *arXiv preprint arXiv:1606.06451*, 2016.

[36] S Alexander Chin and Jason H Anderson. An architecture-agnostic integer linear programming approach to cgra mapping. In *Design Automation Conference (DAC)*, 2018.

[37] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. *arXiv:1805.06085*, 2018.

[38] Kanghyun Choi, Deokki Hong, Hojae Yoon, Joonsang Yu, Youngsok Kim, and Jinho Lee. Dance: Differentiable accelerator/network co-exploration. *arXiv preprint arXiv:2009.06237*, 2020.

[39] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[40] Jason Cong, Zhenman Fang, Michael Gill, and Glenn Reinman. PARADE: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2015.

[41] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2011.

[42] Jason Cong and Zhiru Zhang. An efficient and versatile scheduling algorithm based on sdc formulation. In *Design Automation Conference (DAC)*, 2006.

[43] Edoardo Conti, Vashisht Madhavan, Felipe Petroski Such, Joel Lehman, Kenneth Stanley, and Jeff Clune. Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[44] Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Li, Guodong Zhang, Han Hu, and Yichen Wei. Deformable convolutional networks. In *Proceedings of the International Conference on Computer Vision (ICCV)*, 2017.

[45] Shail Dave, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, and Aviral Shrivastava. DMazeRunner: Executing perfectly nested loops on dataflow accelerators. *ACM Transactions on Embedded Computing Systems*, 2019.

[46] DeepBench. http://www.github.com/baidu-research/deepbench.

[47] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[48] Grace Dinh and James Demmel. Communication-optimal tilings for projective nested loops with arbitrary bounds. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, 2020.

[49] Zhen Dong, Yizhao Gao, Qijing Huang, John Wawrzynek, Hayden KH So, and Kurt Keutzer. Hao: Hardware-aware neural architecture optimization for efficient inference. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021.

[50] Zhen Dong, Zhewei Yao, Daiyaan Arfeen, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. HAWQ-V2: Hessian aware trace-weighted quantization of neural networks. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[51] Zhen Dong, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Hawq: Hessian aware quantization of neural networks with mixed-precision. In *Proceedings of the International Conference on Computer Vision (ICCV)*, 2019.

[52] Alan AA Donovan and Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[53] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[54] Kaiwen Duan, Song Bai, Lingxi Xie, Honggang Qi, Qingming Huang, and Qi Tian. Centernet: Keypoint triplets for object detection. In *Proceedings of the International Conference on Computer Vision (ICCV)*, 2019.

[55] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision (IJCV)*, 2010.

[56] Fasih Ud Din Farrukh, Chun Zhang, Yancao Jiang, Zhonghan Zhang, Ziqiang Wang, Zhihua Wang, and Hanjun Jiang. Power efficient tiny yolo cnn using reduced hardware resources based on booth multiplier and wallace tree adders. *IEEE Open Journal of Circuits and Systems*, 2020.

[57] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 2012.

[58] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, G. Weisz, Lisa Woods, Sitaram Lanka, Steve Reinhardt, Adrian Caulfield, Eric Chung, and Doug Burger. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018.

[59] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 2011.

[60] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2017.

[61] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. Tangram: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2019.

[62] Amir Gholami, Kiseok Kwon, Bichen Wu, Zizheng Tai, Xiangyu Yue, Peter Jin, Sicheng Zhao, and Kurt Keutzer. Squeezenext: Hardware-aware neural network design. *arXiv preprint arXiv:1803.10615*, 2018.

[63] Google. Edge TPU. `https://cloud.google.com/edge-tpu/`. Accessed: 2018-12-05.

[64] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2011.

[65] Vivien Gueant. iPerf-The ultimate speed test tool for TCP, UDP and SCTPTest the limits of your network Internet neutrality test. `https://iperf.fr/iperf-doc.php`. Accessed: 2021-07-01.

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

[67] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang. The architectural implications of facebook's dnn-based personalized recommendation. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2020.

[68] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2020.

[69] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Sophia Shao, Krste Asanovic, and Ion Stoica. Learning to vectorize using deep reinforcement learning. In *Workshop on ML for Systems at NeurIPS*, December 2019.

[70] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Sophia Shao, Krste Asanovic, and Ion Stoica. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. *International Symposium on Code Generation and Optimization (CGO)*, 2020.

[71] Ameer Haj-Ali, Hasan Genc, Qijing Huang, William Moses, John Wawrzynek, Krste Asanović, and Ion Stoica. Protuner: tuning programs with monte carlo tree search. *arXiv preprint arXiv:2005.13685*, 2020.

[72] Ameer Haj-Ali, Nesreen K. Ahmed, Ted Willke, Joseph Gonzalez, Krste Asanovic, and Ion Stoica. A view on deep reinforcement learning in system optimization. *arXiv preprint arXiv:1908.01275*, 2019.

[73] Haj-Ali, Ameer, Qijing Huang, John Xiang, William Moses, Krste Asanovic, John Wawrzynek, and Ion Stoica. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. *Proceedings of Machine Learning and Systems (MLSys)*, 2, 2020.

[74] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, Shih-Wei Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, 1996.

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

[76] Cong Hao, Xiaofan Zhang, Yuhong Li, Sitao Huang, Jinjun Xiong, Kyle Rupnow, Wenmei Hwu, and Deming Chen. Fpga/dnn co-design: An efficient design methodology for 1ot intelligence on the edge. In *Design Automation Conference (DAC)*, 2019.

[77] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *International Symposium on Circuits and Systems*, 2008.

[78] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[79] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016.

[80] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W Fletcher. Mind mappings: enabling efficient algorithm-accelerator mapping space search. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2021.

[81] G. Henry, P. Palangpour, M. Thomson, J. S. Gardner, B. Arden, J. Donahue, K. Houck, J. Johnson, K. O'Brien, S. Petersen, B. Seroussi, and T. Walker. High-performance deep-learning coprocessor integrated into x86 soc with server-class cpus industrial

product. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2020.

[82] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2020.

[83] Daniel S. Hirschberg, Ashok K. Chandra, and Dilip V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[84] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[85] John H Holland. Genetic algorithms. *Scientific american*, 267(1):66–73, 1992.

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

[87] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[88] Qijing Huang, Aravind Kalaiah, Minwoo Kang, James Demmel, Grace Dinh, John Wawrzynek, Thomas Norell, and Yakun Sophia Shao. CoSA: Scheduling by Constrained Optimization for Spatial Accelerators. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2021.

[89] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Stephen Brown, and Jason Anderson. The effect of compiler optimizations on high-level synthesis for FPGAs. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2013.

[90] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Nazanin Calagar, Stephen Brown, and Jason Anderson. The effect of compiler optimizations on high-level synthesis-generated hardware. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2015.

[91] Qijing Huang, Hamid Shojaei, Fred Zyda, Azade Nazi, Shobha Vasudevan, Sat Chatterjee, and Richard Ho. Faster coverage convergence with automatic test parameter tuning in constrained random verification. In *Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE)*, 2021.

[92] Qijing Huang, Christopher Yarp, Sagar Karandikar, Nathan Pemberton, Benjamin Brock, Liang Ma, Guohao Dai, Robert Quitt, Krste Asanovic, and John Wawrzynek. Centrifuge: Evaluating Full-System HLS-Generated Heterogeneous-Accelerator SoCs using FPGA-Acceleration. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2019.

[93] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[94] Intel. Intel FPGA SDK for OpenCL.

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

[96] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of Machine Learning and Systems (MLSys)*, 2019.

[97] Li Jiao, Cheng Luo, Wei Cao, Xuegong Zhou, and Lingli Wang. Accelerating low bit-width convolutional neural networks with embedded fpga. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

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

[99] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. In *Reinforcement learning: A survey*, volume 4, pages 237–285, 1996.

[100] Sheng-Chun Kao, Geonhwa Jeong, and Tushar Krishna. ConfuciuX: Autonomous Hardware Resource Assignment for DNN Accelerators using Reinforcement Learning. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2020.

[101] Sheng-Chun Kao and Tushar Krishna. GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2020.

[102] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018.

[103] James Kennedy. Particle swarm optimization. *Encyclopedia of machine learning*, pages 760–766, 2010.

[104] Brucek Khailany, Evgeni Krimer, Rangharajan Venkatesan, Jason Clemons, Joel S. Emer, Matthew Fojtik, Alicia Klinefelter, Michael Pellauer, Nathaniel Pinckney, Yakun Sophia Shao, Shreesha Srinath, Christopher Torng, Sam Likun Xi, Yanqing Zhang, and Brian Zimmer. A modular digital vlsi flow for high-productivity soc design. In *Design Automation Conference (DAC)*, 2018.

[105] Donggyu Kim, Christopher Celio, David Biancolin, Jonathan Bachrach, and Krste Asanovic. Evaluation of risc-v rtl with fpga-accelerated simulation. In *First Workshop on Computer Architecture Research with RISC-V*, 2017.

[106] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[107] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. When polyhedral transformations meet simd code generation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[108] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[109] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[110] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet Classification with Deep Convolutional Neural Networks. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2012.

[111] Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2012.

[112] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Programmable Interconnects. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2018.

[113] Kiseok Kwon, Alon Amid, Amir Gholami, Bichen Wu, Krste Asanovic, and Kurt Keutzer. Co-design of deep neural nets and neural net accelerators for embedded vision applications. *arXiv preprint arXiv:1804.10642*, 2018.

[114] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, 2004.

[115] Hei Law and Jia Deng. Cornernet: Detecting objects as paired keypoints. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

[116] Hei Law, Yun Teng, Olga Russakovsky, and Jia Deng. Cornernet-lite: Efficient keypoint based object detection. *arXiv preprint arXiv:1904.08900*, 2019.

[117] Jure Leskovec and Andrej Krevl. Snap: Stanford metwork analysis project. `https://snap.stanford.edu`. Accessed: 2021-07-01.

[118] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. A high performance fpga-based accelerator for large-scale convolutional neural networks. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[119] Rundong Li, Yan Wang, Feng Liang, Hongwei Qin, Junjie Yan, and Rui Fan. Fully quantized network for object detection. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[120] Shuai Li, Kuangyuan Sun, Yukui Luo, Nandakishor Yadav, and Ken Choi. Novel cnn-based ap2d-net accelerator: An area and power efficient solution for real-time applications on mobile fpga. *Electronics*, 9(5):832, 2020.

[121] Yanghao Li, Yuntao Chen, Naiyan Wang, and Zhaoxiang Zhang. Scale-aware trident networks for object detection. In *Proceedings of the International Conference on Computer Vision (ICCV)*, 2019.

[122] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Rllib: Abstractions for distributed

reinforcement learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2018.

[123] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. Fp-bnn: Binarized neural network on fpga. *Neurocomputing*, 275:1072–1086, 2018.

[124] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2014.

[125] Shu Liu, Lu Qi, Haifang Qin, Jianping Shi, and Jiaya Jia. Path aggregation network for instance segmentation. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[126] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016.

[127] LowRISCV. Diplomacy and TileLink from the Rocket Chip.

[128] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. *arXiv preprint arXiv:1807.11164*, 2018.

[129] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 2017.

[130] Yufei Ma, Tu Zheng, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Algorithm-hardware co-design of single shot detector for fast object detection on fpgas. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2018.

[131] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. `https://www.cs.virginia.edu/stream/`.

[132] Mentor. Catapult high-level synthesis.

[133] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2016.

[134] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A Distributed Framework for Emerging AI Applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[135] Steven S. Muchnick. Advanced compiler design and implementation. In *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[136] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 2016.

[137] Hiroki Nakahara, Tomoya Fujii, and Shimpei Sato. A fully connected layer elimination for a binarizec convolutional neural network on an fpga. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[138] Hiroki Nakahara, Haruyoshi Yonekawa, Tomoya Fujii, and Shimpei Sato. A lightweight yolov2: A binarized cnn with a parallel support vector regression for an fpga. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 2018.

[139] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems, 2019.

[140] Tony Nowatzki, Newsha Ardalani, Karthikeyan Sankaralingam, and Jian Weng. Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2018.

[141] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. A general constraint-centric scheduling framework for spatial architectures. *ACM SIGPLAN Notices*, 2013.

[142] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, et al. Can fpgas beat gpus in accelerating next-generation deep neural networks? In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 2017.

[143] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[144] OpenAI. Ai and compute. `https://openai.com/blog/ai-and-compute`. Accessed: 2021-07-01.

[145] Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *International Symposium on Code Generation and Optimization (CGO)*, 2006.

[146] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019.

[147] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[148] Eunjung Park, John Cavazos, Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and P Sadayappan. Predictive modeling in a polyhedral optimization space. *International journal of parallel programming*, 2013.

[149] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. Swizzle inventor: Data movement synthesis for gpu kernels. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2019.

[150] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[151] Luca Piccolboni, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P Carloni. Broadening the exploration of the accelerator design space in embedded scalable platforms. In *High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[152] Luca Piccolboni, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P Carloni. Cosmos: Coordination of high-level synthesis and memory optimization for hardware accelerators. *ACM Transactions on Embedded Computing Systems (TECS)*, 2017.

[153] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[154] Thomas B Preußer, Giulio Gambardella, Nicholas Fraser, and Michaela Blott. Inference of quantized neural networks on heterogeneous all-programmable devices. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018.

[155] Siyuan Qiao, Liang-Chieh Chen, and Alan Yuille. Detectors: Detecting objects with recursive feature pyramid and switchable atrous convolution. *arXiv preprint arXiv:2006.02334*, 2020.

[156] Qijing Huang, Haj-Ali, Ameer, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. AutoPhase: Compiler Phase-Ordering for HLS with Deep Reinforcement Learning. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019.

[157] Qijing Huang, Wang, Dequan, Dong, Zhen, Yizhao Gao, Yaohui Cai, Tian Li, Bichen Wu, Kurt Keutzer, and John Wawrzynek. Efficient Deployment of Input-adaptive Object Detection on FPGAs. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 2021.

[158] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2020.

[159] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 2016.

[160] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Fredo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 2013.

[161] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2013.

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

[163] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[164] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-time Object Detection. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[165] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[166] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems (NIPS)*, 2015.

[167] M Roozmeh. High level synthesis of bitonic sorting algorithm, 2016.

[168] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE computer architecture letters*, 2011.

[169] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635, 2011.

[170] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[171] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[172] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[173] Keertana Settaluri, Ameer Haj-Ali, Qijing Huang, Kourosh Hakhamaneshi, and Borivoje Nikolic. Autockt: Deep reinforcement learning of analog circuit designs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2020.

[174] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2019.

[175] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek

Khailany, and Stephen W. Keckler. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2019.

[176] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2014.

[177] Yakun Sophia Shao, Sam (Likun) Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. Co-Designing Accelerators and SoC Interfaces using gem5-Aladdin. In *International Symposium on Microarchitecture (MICRO)*, 2016.

[178] Evan Shelhamer, Dequan Wang, and Trevor Darrell. Blurring the line between structure and learning to optimize and adapt receptive fields. *arXiv preprint arXiv:1904.11487*, 2019.

[179] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2019.

[180] Siemens. The 2020 wilson research group functional verification study. `https://blogs.sw.siemens.com/verificationhorizons/2021/01/06/part-8-the-2020-wilson-research-group-functional-verification-study`. Accessed: 2021-07-01.

[181] SiFive. Sifive tilelink specification, August 2017.

[182] Frans Sijstermans. The NVIDIA Deep Learning Accelerator. In *Hot Chips*, 2018.

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

[184] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-scale Image Recognition. *CoRR*, abs/1408.1556, 2014.

[185] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[186] Jinook Song, Yunkyo Cho, Jun-Seok Park, Jun-Woo Jang, Sehwan Leev, Joon-Ho Song, Jae-Gon Lee, and Inyup Kang. An 11.5 tops/w 1024-mac butterfly structure dual-core sparsity-aware neural processing unit in 8nm flagship mobile soc. In *Proceedings of the International Solid State Circuits Conference (ISSCC)*, 2019.

[187] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2003.

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

[189] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2014.

[190] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[191] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2000.

[192] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[193] Zhangxi Tan, Andrew Waterman, Henry Cook, Sarah Bird, Krste Asanović, and David Patterson. A case for FAME: FPGA architecture model execution. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 290–301, 2010.

[194] Philippe Tillet, HT Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the International Workshop on Machine Learning and Programming Languages*, 2019.

[195] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I August. Compiler optimization-space exploration. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 204–215. IEEE Computer Society, 2003.

[196] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 2017.

[197] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor

comprehensions: Framework-agnostic high-performance machine learning abstractions, 2018.

[198] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[199] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[200] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

[201] Z. Wang and M. OBoyle. Machine learning in compiler optimization. In *Machine Learning in Compiler Optimization*, volume 106, pages 1879–1901, Nov 2018.

[202] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *Design Automation Conference (DAC)*, 2017.

[203] Yasitha M Wijesinghe, Jayathu G Samarawickrama, and Dileeka Dias. Hardware and software co-design for object detection with modified vibe algorithm and particle filtering based object tracking. In *2019 14th Conference on Industrial and Information Systems (ICIIS)*, 2019.

[204] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. *Communications of the Association for Computing Machinery*, 2009.

[205] Bichen Wu, Forrest N Iandola, Peter H Jin, and Kurt Keutzer. Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In *CVPR Workshops*, 2017.

[206] Bichen Wu, Alvin Wan, Xiangyu Yue, Peter Jin, Sicheng Zhao, Noah Golmant, Amir Gholaminejad, Joseph Gonzalez, and Kurt Keutzer. Shift: A zero flop, zero parameter alternative to spatial convolutions. *arXiv preprint arXiv:1711.08141*, 2017.

[207] Bichen Wu, Alvin Wan, Xiangyu Yue, and Kurt Keutzer. Squeezeseg: Convolutional neural nets with recurrent crf for real-time road-object segmentation from 3d lidar point cloud. *arXiv preprint arXiv:1710.07368*, 2017.

[208] Bichen Wu, Xuanyu Zhou, Sicheng Zhao, Xiangyu Yue, and Kurt Keutzer. Squeezesegv2: Improved model structure and unsupervised domain adaptation for road-object segmentation from a lidar point cloud. *arXiv:1809.08495*, 2018.

[209] Saining Xie, Ross Girshick, Piotr Dollar, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[210] Xilinx. General Matrix Operation. `https://github.com/Xilinx/gemx`.

[211] Xilinx. Vivado design suite user guide - high-level synthesis, June 2015.

[212] Xilinx. *PYNQ Introduction*, 2018. `https://pynq.readthedocs.io/en/v2.3`.

[213] Xilinx. Vivado Design Suite User Guide - High-Level Synthesis (UG902), 2018.

[214] Ke Xu, Xiaoyun Wang, and Dong Wang. A scalable opencl-based fpga accelerator for yolov2. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019.

[215] Xiaowei Xu, Xinyi Zhang, Bei Yu, X Sharon Hu, Christopher Rowen, Jingtong Hu, and Yiyu Shi. Dac-sdc low power object detection challenge for uav applications. *IEEE Transactions on pattern analysis and machine intelligence (TPAMI)*, 2019.

[216] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, 2020.

[217] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294, 2011.

[218] Yifan Yang, Qijing Huang, Bichen Wu, Tianjun Zhang, Liang Ma, Giulio Gambardella, Michaela Blott, Luciano Lavagno, Kees Vissers, John Wawrzynek, et al. Synetgy: Algorithm-Hardware Co-design for Convnet Accelerators on Embedded FPGAs. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 2019.

[219] Yifan Yang, Qijing Huang, Bichen Wu, Tianjun Zhang, Liang Ma, Giulio Gambardella, Michaela Blott, Luciano Lavagno, Kees Vissers, John Wawrzynek, et al. Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded FPGAs. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 2019.

[220] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

[221] Fisher Yu, Dequan Wang, Evan Shelhamer, and Trevor Darrell. Deep layer aggregation. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[222] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 2015.

[223] Chi Zhang and Viktor Prasanna. Frequency domain acceleration of convolutional neural networks on cpu-fpga shared memory system. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 2017.

[224] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

[225] Hang Zhang, Chongruo Wu, Zhongyue Zhang, Yi Zhu, Zhi Zhang, Haibin Lin, Yue Sun, Tong He, Jonas Mueller, R Manmatha, et al. Resnest: Split-attention networks. *arXiv preprint arXiv:2004.08955*, 2020.

[226] Jialiang Zhang and Jing Li. Improving the performance of opencl-based fpga accelerator for convolutional neural network. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 2017.

[227] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-X: An Accelerator for Sparse Neural Networks. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2016.

[228] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[229] Xiaofan Zhang, Yuhong Li, Cong Hao, Kyle Rupnow, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. Skynet: A champion model for dac-sdc on low power object detection. *arXiv preprint arXiv:1906.10327*, 2019.

[230] Yongan Zhang, Yonggan Fu, Weiwen Jiang, Chaojian Li, Haoran You, Meng Li, Vikas Chandra, and Yingyan Lin. Dna: Differentiable network-accelerator co-search. *arXiv preprint arXiv:2010.14778*, 2020.

[231] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 2017.

[232] H. Zhong, X. Liu, Y. He, and Y. Ma. Shift-based Primitives for Efficient Convolutional Neural Networks. *ArXiv e-prints*, September 2018.

[233] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044*, 2017.

[234] R. Zhou and T. M. Jones. Janus: Statically-driven and profile-guided automatic dynamic binary parallelisation. In *International Symposium on Code Generation and Optimization (CGO)*, 2019.

[235] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[236] Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. Objects as points. *arXiv preprint arXiv:1904.07850*, 2019.

[237] Xingyi Zhou, Jiacheng Zhuo, and Philipp Krahenbuhl. Bottom-up object detection by grouping extreme and center points. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

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

[239] Xizhou Zhu, Han Hu, Stephen Lin, and Jifeng Dai. Deformable convnets v2: More deformable, better results. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[240] Yuhao Zhu, Anand Samajdar, Matthew Mattina, and Paul Whatmough. Euphrates: algorithm-soc co-design for low-power mobile continuous vision. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 2018.

[241] Bohan Zhuang, Chunhua Shen, Mingkui Tan, Lingqiao Liu, and Ian Reid. Towards effective low-bitwidth convolutional neural networks. In *Proceedings of the Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[242] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. *arXiv:1707.07012*, 2017.