HIGH-EFFICIENCY AND HIGH-USABILITY HETEROGENEOUS HARDWARE ACCELERATION WITH FPGAS

BY

SITAO HUANG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Doctoral Committee:

Professor Wen-mei Hwu, Co-Chair
Professor Deming Chen, Co-Chair
Professor Sanjay Patel
Assistant Professor Jian Huang
Professor Jason Cong, University of California, Los Angeles
Doctor Stephen Neuendorffer, Xilinx Research Labs

ABSTRACT

The exploding complexity and computation efficiency requirements of applications are stimulating a strong demand for hardware acceleration with heterogeneous platforms that may contain CPUs, GPUs, FPGAs, ASICs, and other customized accelerators. Among these processors and hardware accelerators, field-programmable gate arrays (FPGAs) provide flexible programmability, low computation latency, as well as fine-grained parallel processing capability, and have demonstrated outstanding performance and flexibility in many applications and scenarios. However, a high-quality FPGA design is very hard to create and optimize as it requires FPGA expertise and a long design iteration time. In contrast, software applications are typically developed in a shorter development cycle, with high-level languages like Python, which is at a much higher level of abstraction than all existing hardware design languages.

In this dissertation, we will first look into the basics of high-efficiency and high-usability FPGA accelerators in the heterogeneous hardware acceleration systems, including categories of accelerators, system architecture, design methodology, and so on. Secondly, we will discuss the optimization of heterogeneous systems that enables CPU-FPGA collaborative computing and improves system performance. Thirdly, we will look into our proposed high-level programming languages and optimization flows of FPGA accelerators, including language design, compiler design, optimization techniques, and so on. Finally, we will discuss how the proposed high-level programming and optimization flow can be used to program and optimize heterogeneous systems.

To my parents, for their love and support.

ACKNOWLEDGMENTS

Foremost, I would like to express my sincere gratitude to my advisors Professor Deming Chen and Professor Wen-mei Hwu for their continuous support of my study and research, and for their great patience, guidance, trust and encouragement. Their immense knowledge, brilliant intuition, and enthusiasm make them the role model of great researchers. I could not have imagined having better advisors for my study and research.

I would like to thank all my labmates in the ES-CAD research group and the IMPACT research group, for their great help and stimulating and exciting discussions on my research. I learned a lot and had a great time working with them. I would never have completed this thesis without their generous help.

My special thanks go to Junyao Wang, for all her love and support.

Lastly, I would like to thank my mother Yali Kang and my father Jianyuan Huang, for allowing me to realize my own potential, for their patience, encouragement, and support throughout my life.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

ASIC	Application-Specific Integrated Circuit
AST	Abstract Syntax Tree
CPU	Central Processing Unit
DNN	Deep Neural Network
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HLS	High-Level Synthesis
JIT	Just-In-Time
TPU	Tensor Processing Unit

CHAPTER 1

INTRODUCTION

The last decade has witnessed the exploding growth in data, applications, as well as computation needs from end users in various scenarios. The resulting growing complexity poses serious challenges on the design of highly efficient computing systems that can deliver desired computing performance. In many application scenarios, it is very hard to deliver desired performance by only using general-purpose processors like central processing units (CPUs). This limitation of general-purpose processors stimulates a strong demand for hardware acceleration. People start to introduce various types of accelerators into the computing system, such as graphics processing units (GPUs), field-programmable gate arrays (FPGAs), application-specific integrated circuits (ASICs), and other customized accelerators. Each of these accelerators has specialized capabilities in accelerating certain computation patterns or applications. The growing complexity of applications and data also drives the capability and complexity growth of modern computing systems. What makes an ideal hardware acceleration system, and how to design a great computing system remain challenging problems to solve.

As we will discuss later in this dissertation, there are three key aspects of hardware acceleration as follows.

❶ *Efficiency*: For target applications, the hardware system should be able to deliver as high execution performance as possible with minimum area, energy consumption, and design cost. This is the basic requirement of designing hardware acceleration systems and it has been the main theme of many prior research works.

❷ *Usability*: Hardware acceleration should be easily available and usable for any target applications, any target deployment environments, and any end users. This implies that hardware acceleration systems should be easy to design, debug, optimize, maintain, and use, in any given

deployment environments, without much human manual effort. This requires programming language and compiler innovations to fundamentally optimize the way of designing hardware systems.

❸ *Heterogeneity*: Computer systems are moving toward heterogeneous integration of diverse hardware accelerators with different characteristics. Modern computer systems should be extensible, flexible, and have the capabilities of supporting emerging hardware accelerators. This requires innovations in low-level hardware architecture, as well as co-design and co-optimization of the whole system stack.

Computing efficiency has always been among the top design requirements of hardware accelerators. Researchers have been looking into and optimizing many different metrics of accelerator efficiency, including processing latency, throughput, execution time, thermal performance, etc. On the contrary, the design efficiency and usability of hardware accelerators has not attracted enough attention from researchers. However, usability also plays a key role in hardware accelerator design. Highly usable accelerator designs are the ones that allow people to easily deploy the accelerator in different applications and scenarios, and the accelerator is supported by a complete set of tool chains that simplify the programming and maintenance of the accelerator. As we point out in this dissertation, having proper programming languages and compilation flows is the key to highly usable accelerator design. Heterogeneity is the trend of future hardware acceleration systems. As the complexity of application requirements grows, it is hard for homogeneous systems to deliver the performance required. Introducing heterogeneous components into the computing system and leveraging the unique capabilities of each component are becoming the trend of hardware acceleration system design. As systems moving toward heterogeneity, challenges in designing and optimizing these heterogeneous systems arise. As we will show in this dissertation, collaborative computing and high-level programming and optimization flows are the keys to solve these challenges.

Among the processors and hardware accelerators used in the computing system, field-programmable gate arrays (FPGAs) provide flexible programmability, low computation latency, as well as fine-grained parallel processing capability, and have demonstrated outstanding performance and flexibility in many applications and scenarios. However, a high-quality FPGA

accelerator design is very hard to create and optimize as it requires FPGA expertise and a long design iteration time. In contrast, software applications are typically developed in a short development cycle, with high-level languages like Python, which is at a much higher level of abstraction than all existing hardware design languages. How to create and optimize FPGA designs more efficiently becomes a key challenge to solve.

In this dissertation, we will focus on the techniques that tackle the challenges in the design and optimization of highly efficient, highly usable heterogeneous hardware acceleration systems that are built with FPGAs.

The rest of this dissertation is organized as follows. Chapter 2 provides an overview on the basics of high-efficiency and high-usability FPGA accelerators in the heterogeneous hardware acceleration systems, including categories of accelerators, system architecture, design methodology, and so on. Chapter 3 uses sparse DNN inference as an example application for case study and demonstrates the design and optimization of an application specific accelerator. Chapter 4 discusses the collaboration of computing components (CPU-FPGA) in a heterogeneous system and demonstrates task-level design and optimization of heterogeneous systems. Chapter 5 introduces our proposed Python-based high-level programming language and synthesis flow, PyLog, which greatly eases the challenges in FPGA accelerator design. Chapter 6 uses quantization in ReRAM-based DNN accelerators as an example to discuss design space exploration in accelerator design. Finally, Chapter 7 concludes this dissertation.

CHAPTER 2

EFFICIENT HARDWARE ACCELERATORS

In this chapter, we will briefly review the types of hardware accelerators that are widely used, the latest hardware features provided by recent hardware platforms, and the design methodology of FPGA accelerators.

2.1 Common Accelerator Types

General-Purpose Graphics Processing Unit (GPGPU). Originally, GPUs are the specialized processing units in the computing system that accelerate the creation and manipulation of images for display. GPUs can be found in many different devices from edge devices like mobile phones and smart watches, to high-performance systems like workstations and servers. During the recent decades, people use GPUs to do general-purpose computing and realize that GPUs can deliver great computation throughput in many application domains including deep learning, graph processing, scientific computing, high-performance computing, and so on.

Field-Programmable Gate Array (FPGA). An FPGA is a special integrated circuit chip that can be configured into digital logic specified by users. FPGAs contain arrays of programmable gate logic blocks, interconnect blocks, as well as other specialized hardware blocks like memory, DSPs, specialized IPs and so on. These components in the FPGAs are the basic configurable blocks. Even though the typical operating frequency of FPGAs is not as high as application-specific integrated circuit (ASIC), FPGAs provide flexible programmability, low computation latency, as well as fine-grained parallel processing capability.

Application-Specific Integrated Circuit (ASIC). An ASIC is a specialized integrated circuit chip that is customized to accelerate a specific type of application or computation pattern. As it is designed to perform only one

type or one set of operations, typically an ASIC is highly optimized and can deliver very good computation performance. However, as it is customized circuit, it usually has no or very limited programmability.

Accelerators with Emerging Architectures. In recent years, people have been exploring the alternative design options of accelerators. Concepts of analog computing, in-memory computing, near-storage computing, etc., have been proposed. Some of these accelerators use emerging analog circuit techniques to build accelerators, which provide energy efficient acceleration solutions. In Chapter 6, we will look into the optimization of ReRAM-based DNN accelerators. Some of them bring computing units into memory or storage, which reduces data access latency.

2.2 Heterogeneous Systems

One recent trend of computing systems is heterogeneity. Different types of processors and accelerators provide different computing capabilities, and introducing heterogeneous components into the system increases system flexibility and efficiency. One example is the System-on-Chip (SoC) platforms that typically incorporate CPUs, programmable logic (or other specialized accelerators), and sometimes embedded GPUs. These SoC systems provide many different system level configuration options, including accelerator configuration options as well as interconnect configuration options. This configurability greatly improves the flexibility of these SoC systems.

Recently, more and more systems incorporate various types of accelerators and processors. For example, Xilinx Versal Adaptive Compute Acceleration Platform (ACAP) platforms contain embedded ARM cores, adaptable logic engines, specialized AI engines, DSP engines, and Network-on-Chip (NoC) components. NVIDIA Jetson AGX Xavier is an SoC platform designed for autonomous machines. It contains embedded ARM core, Volta GPU, deep learning accelerator (DLA), and vision acclerator (VA).

The growing heterogeneity also poses severe challenges in designing, programming, and optimizing these complicated systems. High usability becomes especially important for these systems. More specifically, how to leverage the compute capability of each component in the system and how to program and optimize the overall performance of the heterogeneous sys-

tems are the critical problems. In Chapter 4, we will discuss how to enable collaboration between different computing units in these systems and further improve the overall system performance.

2.3 FPGA Design Methodology

As we discussed above, FPGAs are special chips that can be configured into specialized digital circuits. In the previous FPGA design methodology, there are mainly two levels of approaches, RTL-based design flow and HLS-based design flow.

The most widely adopted FPGA development flow today starts with programming FPGA at the register transfer level (RTL) in hardware description languages (HDL) such as Verilog and VHDL. Then designers use FPGA synthesis tools from FPGA vendors to synthesize RTL designs into FPGA bitstreams, which are used to configure FPGA. Programming FPGA at this level requires rich expertise in digital circuit design and the FPGA architecture. Besides, programming at this level is non-intuitive, error-prone, and hard to reuse code compared with modern programming languages, leading to long development, optimization, and verification cycles.

High-level synthesis (HLS) aims to simplify FPGA programming. Elevating the abstraction level of FPGA programming to that of C/C++/OpenCL [1, 2, 3, 4], HLS tools enable FPGA designers to express their algorithms in more familiar high-level languages. Developers are expected to use HLS pragmas or directives to guide the HLS tools to optimize and generate desired RTL design. Compared with RTL design flow, HLS allows FPGA developers to develop, optimize, verify, and reuse their design at a higher level, thereby greatly improving productivity. However, as C/C++/OpenCL are initially designed for general-purpose processors and start with an inherent sequential execution model inside each kernel/function definition of these languages, they are essentially different from the FPGA's fine-grained parallel processing nature (the OpenCL model can describe parallel work-items but it is at thread-granularity and not very well supported in current HLS tools). The existing HLS tools are also designed in a way that accommodates the sequential execution model of input languages.

As we will discuss later in the dissertation, we propose to move the ab-

straction level of FPGA programming to an even higher level, such as Python level, where there are high-level operations and data objects. Moving up the abstraction level simplifies FPGA programming, and allows compiler to do more aggressive optimizations. This will be elaborated in Chapter 5.

CHAPTER 3

CASE STUDY: SPARSE DNN ACCELERATOR

In this chapter, we will use sparse DNN inference accelerator as an example to demonstrate the design considerations, methodology, and optimizations when designing FPGA accelerators with HLS tools.

Deep neural networks (DNNs) have been widely adopted in many domains, including computer vision, natural language processing, and medical care. Recent research reveals that sparsity in DNN parameters can be exploited to reduce inference computational complexity and improve network quality. However, sparsity also introduces irregularity and extra complexity in data processing, which make the accelerator design challenging. This work presents the design and implementation of a highly flexible sparse DNN inference accelerator on FPGA. Our proposed inference engine can be easily configured to be used in both mobile computing and high-performance computing scenarios. Evaluation shows our proposed inference engine effectively accelerates sparse DNNs and outperforms the CPU baseline solution by up to 4.7× in terms of energy efficiency.

3.1 Introduction

Recent years have witnessed the success of deep learning in many domains, including computer vision, natural language processing, medical care, autonomous driving and so on [5], [6]. The extraordinary high accuracy of deep learning based approaches is made possible by performing inference using pretrained DNN models, which have very high computation and memory space demands. This complexity presents a significant challenge to adopting DNNs for many real-world applications, especially edge-computing scenarios, which have stringent power and latency requirements for computation. Researchers have invested significant effort in making efficient low-computational-cost

DNN based systems possible. The research efforts mainly fall into three aspects: designing light-weight DNNs, reducing the amount of computation in DNN without sacrificing accuracy, and accelerating DNNs with customized hardware.

Recent research reveals that many parameters in deep neural networks are redundant and can be pruned away. Parameter pruning reduces the number of parameters and the amount of computation; after parameter pruning, the parameters in DNN layers become sparse. However, sparsity in DNN layers also introduces irregularity and extra complexity from sparse data formats and scheduling computation workload. It is challenging for the processors and accelerators to handle the irregularity and extra complexity which may lead to non-negligible overhead in execution time. The overhead diminishes the benefits from sparsity and may even result in worse performance compared to non-sparse approaches if not handled properly.

As deep learning conquers more and more complicated cognitive computing tasks, the size and complexity of DNN architectures explodes. Practitioners realize that it is getting harder and harder to achieve performance and power efficiency targets for deep learning systems with CPUs and GPUs, and specialized deep learning accelerators are needed. Many deep learning accelerators have been proposed and they have all kinds of optimization objectives [7], [8]. Adoption of specialized deep learning accelerators has made many challenging application scenarios possible, including intelligent wearable devices, real-time high-definition video processing systems, etc. FPGAs have been one of the ideal platforms for DNN acceleration as FPGAs provide the combination of low latency, high energy efficiency, and high reconfigurability, which make FPGAs adaptable to many application scenarios.

In this work, we design and build a configurable sparse DNN inference engine on an FPGA that accelerates the inference of sparse DNNs. We target very deep sparse DNNs that can be used as the backbone network for future complex cognitive tasks. These DNNs have many layers and there are many neurons inside a layer. Our proposed inference engine can be reconfigured and adopted in different FPGA platforms, depending on the available hardware resources and application requirements.

The contribution of this work can be summarized as follows:

- We design and build a configurable sparse DNN inference engine that is

highly flexible and capable of processing different sizes of sparse DNNs.

- We propose several design optimization techniques for sparse DNN inference that are general enough to potentially benefit future works on sparse DNN acceleration.
- We model and analyze the computation of sparse DNNs, and we show how the accelerator design can be parameterized and what the accelerator design space looks like.

The rest of this chapter is organized as follows. Section 3.2 provides background information on sparse DNNs and FPGA accelerators. Section 3.3 discusses our proposed accelerator design optimization techniques. Section 3.4 presents the details of our proposed sparse DNN inference engine. Section 3.5 shows our experiment setups and results. Section 3.6 reviews the recent works on similar areas. Finally, Section 3.7 concludes the whole chapter.

3.2 Background

3.2.1 Sparse Deep Neural Networks

In this work, we focus on feedforward deep neural networks that consist of fully connected layers. Note that our formulation for fully connected layers can also be extended for sparse convolution neural networks (CNNs), since convolution can be reduced to matrix multiplication operations. Motivated by the Sparse DNN Graph Challenge [9], we consider a DNN with L layers. Assume each layer in the DNN has M neurons, i.e. the dimension of the input and output feature vectors of each layer is M.

Let $y_{l-1} = (y_{l-1,1}, y_{l-1,2}, \ldots, y_{l-1,M})$ be a single input sample to the l-th layer, and y_l be the corresponding output from the l-th layer. y_0 is the input feature vector to the neural network, e.g. y_0 can be one input image. There can be N input samples, and these N input samples can be stacked as input matrix $\mathbf{Y}_0 = (y_0^{(1)}, y_0^{(2)}, \ldots, y_0^{(N)})^\top$, where $y_0^{(i)}$ is a row vector and is the i-th input sample to the network. Similarly, the input and output of the l-th layer with multiple samples can be represented as matrices $\mathbf{Y}_{l-1}$ and $\mathbf{Y}_l$ respectively. Feature matrices $\mathbf{Y}$'s are $N \times M$ matrices.

The computation of the l-th layer can be formulated as

$$\mathbf{Y}_l = h(\mathbf{Y}_{l-1}\mathbf{W}_l + b_l), 1 \leq l \leq L \tag{3.1}$$

where $h(\cdot)$ is the ReLU function $h(x) = \max(0, x)$. $\mathbf{W}_l$ is an $M \times M$ matrix whose element $\mathbf{W}_l(i, j)$ at the i-th row and the j-th column represents the weight of the connection from the i-th input neuron to the j-th output neuron. b_l is the bias vector of dimension M.

$\mathbf{W}_l$ and b_l $(1 \leq l \leq L)$ are the parameters of the DNN which are determined by DNN training. y_0 is the input feature vector to the neural network. After DNN weight pruning, $\mathbf{W}$'s become sparse matrices. In some application, y_0 is also sparse. For example, in the handwritten digit recognition task (MNIST dataset), only a small subset of image pixels are black ("1") and the rest are white ("0"). In the problem setting of this work, all $\mathbf{W}$'s and y_0 are sparse.

3.2.2 FPGA Accelerators

FPGAs have been used in many application scenarios such as Internet-of-Things (IoT), wearable devices, autonomous driving, cloud computing, and scientific computing. Many different applications benefit from FPGA's low processing latency and high energy efficiency. Programming FPGAs has been a big challenge for decades, which prevents FPGA from being rapidly deployed and adopted in more domains. High-level synthesis (HLS) tools have greatly improved the productivity of FPGA designers. The C/C++/OpenCL to hardware description language (HDL) design flow enabled by HLS tools makes FPGA more accessible for designers.

3.3 Design Optimizations

This section presents the optimizations used in our sparse DNN accelerator. In this work, we use the test sparse networks provided by Graph Challenge [9]. The sparse DNN has 120 layers, each of which contains 1024 neurons. The optimization techniques presented here can be easily generalized to accelerate any sparse DNN.

3.3.1 Dense Feature Vectors and Sparse Parameters

As mentioned in Section 3.2.1, both input image and DNN parameters are sparse. The input files of images and parameters are also in sparse format. The input files contains all the edges between neurons that have weights larger than 0. Therefore, the multiplication in Equation 3.1 is the multiplication of two sparse matrices, and one straightforward design would be implementing multiplication of two sparse matrices directly. In this way, the dot product of a sparse row vector and sparse column vector requires computing the set intersection of their indices. Since these sets are typically small (less than 1024 elements), and the resource complexity of parallelizing the intersection is not trivial, the sequential comparison algorithm will be the most straightforward algorithm. Sequential comparison requires $\mathcal{O}(m + n)$ comparisons, where m and n are the number of non-zero elements in two vectors respectively.

However, we observe that treating both $\mathbf{Y}$ and $\mathbf{W}$ matrices as sparse may not be optimal. In practice, the number of non-zero elements in the column vectors in the weight matrices are typically constrained. In the test data used in this work, the number of non-zero elements in each column of weight matrices $\mathbf{W}$'s is less than or equal to 32 (in 1024 neurons case). However, the sparsity of feature maps varies a lot. The number of non-zero elements in the row vectors in feature maps can vary from 0 to 1024 (in 1024 neurons case).

In this work, we treat the feature maps as dense matrices and the DNN parameters as sparse ones. In this way, the storage of DNN parameters is compact while access to parameter and feature maps is more efficient. With this dense-feature/sparse-parameter scheme, the multiplication of the input feature vector and a column (weights of incoming edges to a neuron) in the parameter matrix can be done in a way shown in Listing 3.1. In Listing 3.1, `param` is the sparse representation of a column in the parameter matrix, an array of pairs of indices and weights. The code iterates through the `param` array, uses indices to retrieve the feature vector `fvec` element and multiplies it with the corresponding weight. With this approach, there are `param.size()` random accesses into the feature vector array `fvec`. Note that the difference from regular dense matrix multiplication is that here the feature vector is randomly accessed as parameter arrays are sparse. This

random access pattern is not friendly to memory access efficiency.

```
float sum = 0.0;
for(int i = 0; i < param.size(); i++) {
  sum += fvec[param[i].idx] * param[i].weight;
}
```

Listing 3.1: Feature vector `fvec` multiplied with a column `param` of the parameter matrix

3.3.2 Grid Representation and Data Dependencies

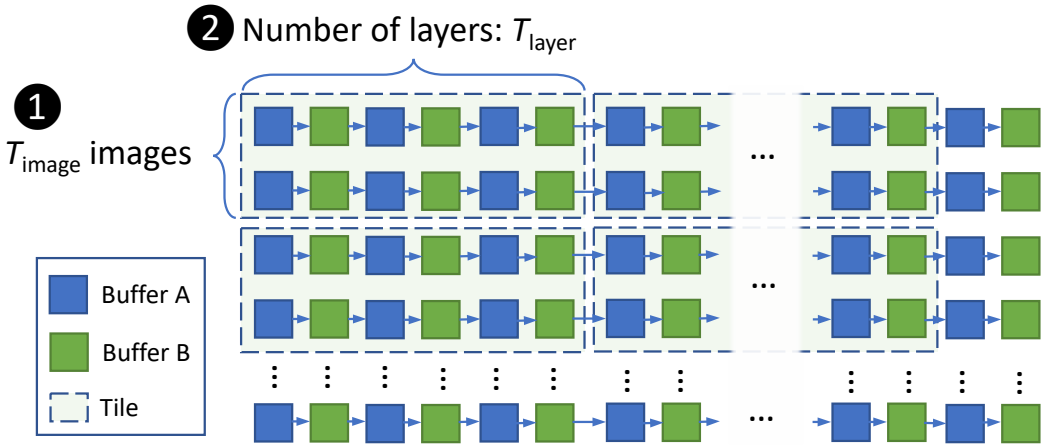


Figure 3.1: Tiling Scheme

In order to better illustrate the data dependencies in the sparse DNN computation, we represent the computation of sparse DNN as a 2D grid, as shown in Figure 3.1. Each row in the grid represents the computation on one input sample (one image). The i-th row represents the processing of the i-th input sample. The number of rows equals the number of input test images. Each column of the 2D grid represents the computation of a specific layer in the sparse DNN on all input samples' feature maps. The j-th column in the 2D grid represents the computation of all input samples' feature maps going through the j-th layer. The grid point at the i-th row and the j-th column represents the computation of the i-th input samples' feature maps and goes through the j-th layer in the DNN. Each grid point includes sparse vector-matrix multiplication (Figure 3.2), bias, and non-linear operations. Using

the previous notation of M neurons, inside each grid point, the dimensions of vector and matrix in the sparse vector-matrix multiplication are M and $M \times M$ respectively.

The computation of rows of the 2D grid is independent from each other, since there is no data dependency between input samples. The computation of columns has dependency on the computation of the previous column. This is because the input to the next layer is the output from the previous layer. Within each grid point, the computation of inner products of the vector and M columns of matrix is independent of each other.

3.3.3 Ping-Pong Buffering

In this work, we use very deep DNNs to test the performance of our system. The number of layers in the test DNNs varies from 120 to 1920. In order to keep track of the activations in this many layers, we use ping-pong buffers to store the input and the output feature maps. For example, when computing layer $2i$, buffer `buf_a` is used to store the input feature maps, while buffer `buf_b` is used to store the output feature maps. When processing layer $2i+1$, the roles of buffers `buf_a` and `buf_b` switch, the output feature map from layer $2i$ in buffer `buf_b` is read and processed and the output is stored back into buffer `buf_a`.

Figure 3.1 uses blue and green color to mark the usage of ping-pong buffers. The feature maps in neighbor layers are stored in two buffers. With this design, we only need two buffers to store the intermediate feature maps no matter how many layers there are. Note that we are not sacrificing performance here as there are intrinsic data dependencies between layers and layers need to be processed sequentially.

3.3.4 Multi-Level Tiling

The on-chip memory resource in FPGA is limited. The size of input samples and DNN parameters are much larger than the capacity of FPGA on-chip memory. Tiling is necessary to reuse on-chip memory space and improve the processing efficiency. In this work, we use tiling along multiple dimensions at multiple levels. The combination of tiling along multiple dimensions at

multiple levels enables high flexibility of the design. Given input sizes and the number of resources on the target FPGA platform, this tiled design can be easily configured for best performance by changing tile sizes.

In our design, tiling happens along three different dimensions and levels: ❶ across input samples (input batch); ❷ across layers (inter-layer); ❸ within a layer, across neurons (intra-layer).

The first type of tiling happens across input samples. Figure 3.1 and Figure 3.2 illustrate this type of tiling (see ❶ in the figures). Multiple input samples (T_{image} samples) are grouped into a tile and processed together. The input samples within a tile share the same copy of DNN parameters. Each load of DNN parameters is reused for T_{image} times. Therefore, the larger tile size T_{image} is, the more times DNN parameters are reused. At the same time, larger T_{image} requires more on-chip memory to store images and parameters.

The second type of tiling is done across DNN layers. ❷ in Figure 3.1 shows this type of tiling. T_{layer} layers form a tile. The parameters in a tile are loaded into on-chip BRAM all at once, and the input samples go through each layer in the tile. The processing within a tile is fully pipelined. That means larger T_{layer} requires more intermediate buffers. The output of the tile is the output feature vector from the last layer in the tile. Depending on the tile execution order, the output of the tile may need to be written back to the global DRAM on the FPGA board if image tiles are iterated first. Let L be the number of layers in DNN, then with tile size of T_{layer}, there will be $\lfloor L/T_{\text{layer}} \rfloor$ feature vectors being written back to the on-board DRAM in that case. The larger T_{layer} is, the fewer write-backs there are, while more on-chip memory is required to store parameters.

The third type of tiling happens at a different level than the previous two types of tiling. This type of tiling happens across neurons within a layer. ❸ in Figure 3.2 illustrates this type of tiling. Multiple columns (neurons) in a DNN layer are grouped into a tile and multiply with the input sample to get the partial sums of corresponding columns. As discussed before, these partial sums are independent from each other and can be done in parallel. The feature vector is duplicated and stored in T_{neuron} different BRAMs so that they can be accessed in parallel. In our design, each of these partial sums is calculated by a separate sparse vector dot product unit.

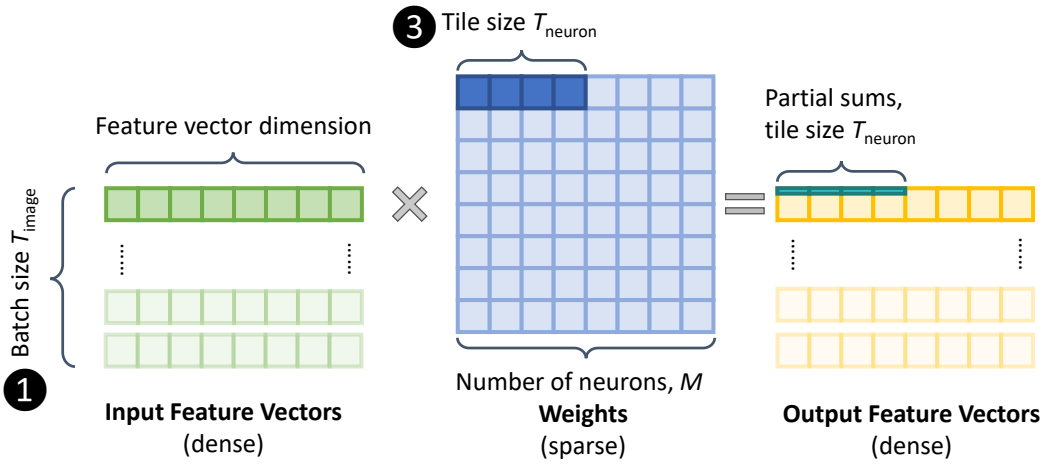


Figure 3.2: Sparse Vector-Matrix Multiplication

3.3.5 Dynamic Workload Balancing

In our sparse DNN inference engine, there are multiple accelerator instances. Each accelerator can be controlled independently. The host program assigns packs of images to these accelerators. We use a dynamic workload assignment algorithm (Algorithm 1) in the host CPU program to balance the workloads of the accelerators [10]. The input samples are partitioned into small packs and used as the minimal assignment unit. In Algorithm 1, one pack contains S input samples, e.g. $S = 32$. The high-level idea of dynamic workload balancing is that the host program checks the status of each accelerator and assigns a pack of input samples to the idle accelerator. There are two cases where the accelerator can accept new workload assignment. The first case is that the accelerator has finished the previous assignments, has results ready, and is ready to accept new ones. Line 7 in Algorithm 1 deals with this case. In this case, the host program collects the results returned from the accelerator, and assigns the current pack to this accelerator. The second case is that the accelerator is idle and does not have results to report (Line 4). In this case, the host program simply assigns a pack of input samples to this accelerator. In practice, we choose $S = 32$, and achieve nearly perfect workload balancing of accelerators.

Algorithm 1 Dynamic Workload Assignment

Input: Number of input samples N, pack size S, accelerator pool `P` = {`P[0]`, ..., `P[m-1]`}.

```
curr_img ← 0, acc_ptr ← 0
size ← MIN(S, N − curr_img)
while curr_img < N do
    if P[acc_ptr].ISIDLE() then
        ASSIGN(curr_img, size, P[acc_ptr])
        curr_img ← curr_img + size
    else if P[acc_ptr].ISDONE() then
        COLLECTRESULTS(P[acc_ptr])
        ASSIGN(curr_img, size, P[acc_ptr])
        curr_img ← curr_img+size
    end if
    acc_ptr ← (acc_ptr+1)%m
end while
```

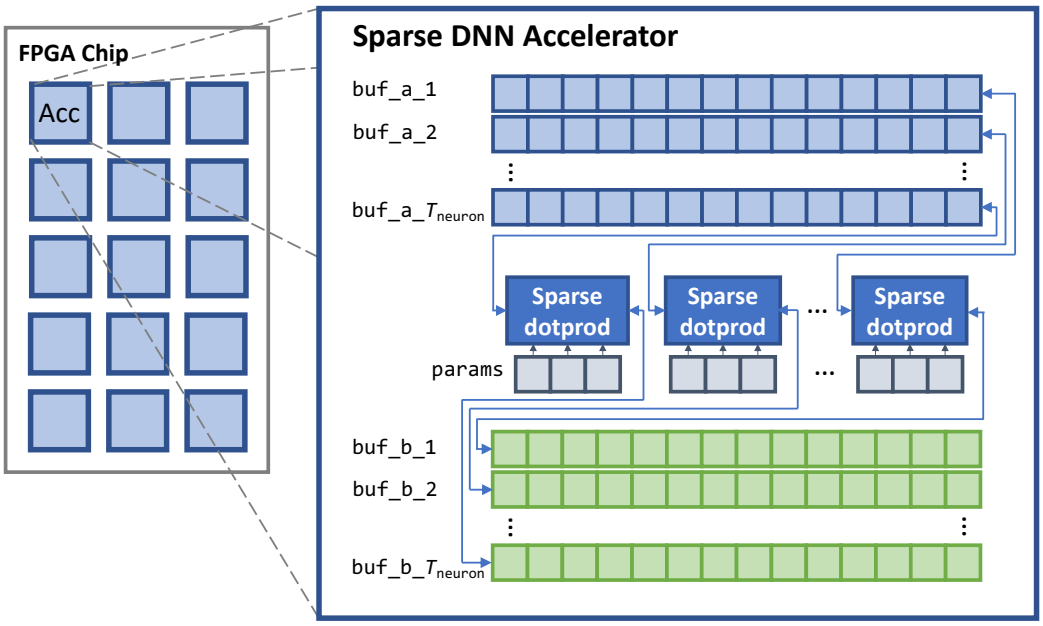


Figure 3.3: Sparse DNN Accelerator Architecture

3.4 Sparse DNN Accelerator Architecture

In this section, we present the hardware architecture of our sparse DNN inference engine, which incorporates all the optimizations discussed in Section 3.3.

Figure 3.3 depicts the high-level view of sparse DNN inference engine in an

FPGA chip and the structure of a sparse DNN accelerator. Our sparse DNN inference engine consists of a pool of accelerators. Each of these accelerators can be controlled independently by the host CPU and they do not require synchronization during the processing. Each accelerator can process any number of input samples and any number of DNN layers.

Our inference engine design can be adopted in both power-constrained edge computing scenarios as well as high-performance cloud computing scenarios. The number of accelerators in the engine is determined by the hardware resource in the FPGA chip. Each accelerator is lightweight but fully capable of running sparse DNN inference. In low-power FPGAs, we can instantiate one single accelerator and achieve low-power high-efficiency processing, while on high-performance FPGAs, many accelerators can be instantiated to achieve high processing throughput.

As illustrated in Figure 3.3, inside each accelerator, multiple pairs of ping-pong buffers (group A `buf_a_`i's and group B `buf_b_`i's in Figure 3.3) and sparse vector dot product processing elements (PEs) are instantiated.

Each PE processes the vector dot product of the same input feature vector with one different column in the parameter matrix. These PEs calculate the partial sums synchronously in a single instruction multiple data (SIMD) manner. The buffers in the accelerator are instantiated with block RAMs (BRAMs) in FPGA. Note that each of these BRAMs has two read ports and can provide two data point per clock cycle. In order to fully utilize the parallelism between columns in the parameter matrix, one input feature vector is actually replicated into T_{neuron} separate buffers, together comprising buffer group A. In this way, T_{neuron} buffers can all be accessed and processed at the same time. The outputs from sparse vector dot product engines are all stored into the same buffer, `buf_b_1`. After processing of one layer, the values in `buf_b_1` are copied into all the other buffers in group B. Then, the weights of the next layer are loaded into parameter buffers (`params` in Figure 3.3) and group B buffers are used as inputs to sparse vector dot product engines. The outputs are stored into the same buffer `buf_a_1`. Again, before processing the next layer, the values in `buf_a_1` are copied to the other buffers in group A.

Table 3.1: Test Platform Information

FPGA Board	Xilinx Virtex-7 FPGA VC709 Board
FPGA Chip	Xilinx XC7VX690TFFG1761-2 FPGA
On-Board Memory	2×4GB DDR3 (up to 933MHz)
On-Chip Memory (Kb)	52,920
On-Chip DSP Slices	3,600
On-Chip Logic Cells	693,120
Host-FPGA Interconnect	PCIe Gen3 up to 8 lanes
Host CPU	Intel(R) Core(TM) i7 CPU 950 @ 3.07GHz
Host Memory	24 GB DDR3 (800MHz)
Operating System	Ubuntu 14.04 LTS
Host Compiler	g++ 4.8.4

3.5 Experiments

3.5.1 Test Platform and Dataset

In this work, we use the Xilinx VC709 board [11] as the target FPGA platform. The basic information about our test system is listed in Table 3.1. We use the synthetic sparse DNN dataset from Graph Challenge [12] to evaluate our solution. The information on the synthetic dataset we used in this work is the sparse DNN with 120 layers. Each layer has 1024 neurons. The input sample dimension is 1024 as well. The total parameter size is 176MB.

The input data is stored in text files. Each line in the text file follows the graph edge representation of `(node_a, node_b, weight)`. Our host code reads the input text files and stored the DNN parameters in Compressed Column Storage (CCS) format. The input images are stored in the dense format.

3.5.2 Design Parameters

In our final design targeting Xilinx VC709 board, we choose the following design parameters to maximize the performance of the system. Please note that the choice of these design parameters highly depends on the target FPGA platform and the design goals.

- Number of accelerators in FPGA $P = 15$. We put as many accelerators as possible onto the FPGA chip and $P = 15$ is the maximum possible

number of accelerators to be integrated into the target Virtex-7 FPGA. Placing more accelerators will lead to severe place and route congestion and timing problem.

- Tiling across input samples $T_{\text{image}} = 1$. We choose $T_{\text{image}} = 1$ and optimize the design for $T_{\text{image}} = 1$ case so that our design can have optimal latency processing one single image. Although choosing $T_{\text{image}} > 1$ can increase the parameter reuse, $T_{\text{image}} > 1$ also requires larger buffer sizes to store intermediate results. Choosing $T_{\text{image}} = 1$ minimizes the pressure on the local on-chip memory.

- Tiling across layers $T_{\text{layer}} = 2$. We load the parameters for two layers at a time to accommodate the processing with ping-pong buffer. We choose to iterate through the layers and not reusing the parameters for images. This way, only the final classification result (one single integer per input sample) needs to be written back, which minimizes the number of intermediate results being written back. This reduces the pressure on DRAM bandwidth and improves the efficiency.

- Tiling across neurons $T_{\text{neuron}} = 16$. We create a script to automatically generate synthesizable C code with various T_{neuron} values and test the latency of the design. It turns out that $T_{\text{neuron}} = 16$ is the optimal setting under the timing constraint of 4 ns per clock cycle (250 MHz). Smaller T_{neuron} does not fully exploits the parallelism within a DNN layer, while larger T_{neuron} introduces larger overhead in extra buffering space.

- Workload assignment pack size $S = 32$. We evaluate the accelerator performance with different S values, such as 32, 64, and 256. The differences in performance with these sizes are not significant for the current setting.

3.5.3 Evaluation

With the design parameters listed in Section 3.5.2, the resource utilization of the inference engine on VC709 board is listed in Table 3.2. Note that different design parameters will lead to different FPGA resource utilization

Table 3.2: FPGA Resource Utilization

Look-Up Tables	209,814 / 433,200 (48.43%)
Flip-Flop	232,720 / 866,400 (26.86%)
BRAM	815 / 1,470 (55.44%)
DSP	150 / 3,600 (4.17%)

ratios. As we explained in Section 3.5.2, we conservatively use around 50% of FPGA resources so that the frequency and timing quality of the synthesized circuit can be guaranteed. The power consumption of this design is around 12 W, which is estimated by the synthesis flow in Xilinx Vivado.

To fully evaluate the benefits of our proposed techniques, we measure the performance of two FPGA designs, one ("Optimized") is with all optimizations described in Section 3.3, the other ("Basic") is a basic FPGA design without tiling. Figure 3.4 and Table 3.3 show the performance of two designs with various numbers of accelerators, as well as the efficiency improvement from the optimized FPGA solution compared to the CPU solution. As shown in the figure, the optimized design can achieve more than five times speedup compared to the basic design. In our evaluation, the best number of accelerators for the optimized design is around 7. For a smaller number of accelerators, adding more accelerators exploits parallelism in processing images and therefore improves performance. However, when there are enough accelerators, FPGA on-board memory bandwidth becomes the bottleneck of the whole system. Even though adding more accelerators increases computational capabilities, it also increases memory access pressure. At some point, memory bandwidth saturates and adding more accelerators no longer improves system performance.

We evaluated the baseline MATLAB code provided by Graph Challenge on a high-performance server with four AMD Opteron 6272 Processors (4×16 cores). The execution time of the MATLAB code is 124.07 seconds, and its power consumption is estimated to be 114 W [13]. Although the multi-core CPU performance is around two times faster than the FPGA solution, the power efficiency of our design is up to 4.7× higher than the multi-core solution. Here are a few notes to help understand the difference in performance between our solution and the MATLAB based multiple CPU solution. First, the MATLAB implementation uses the sparse BLAS libraries in MATLAB, which is highly optimized for sparse matrix operations. Therefore the

Table 3.3: Sparse DNN Accelerator Performance

P	**Basic** (s)	**Optimized** (s)	**Speedup**	**Efficiency over CPU**
1	4618.45	906.81	5.09	1.30
2	2313.04	474.29	4.88	2.49
4	1159.95	310.14	3.74	3.80
6	773.26	254.76	3.03	4.63
7	662.16	251.31	2.63	4.69
8	579.26	269.89	2.14	4.37
12	386.85	489.37	0.79	2.41
15	310.53	484.23	0.64	2.43

MATLAB version is not a trivial reference solution. Second, the target platform has many more hardware resources and higher computation capability. We are comparing our single FPGA solution against a solution based on 64-core server grade high-performance CPUs here. Third, the memory access path from host memory to FPGA device memory has lower peak data transfer bandwidth compared the CPU. In our design, in order to reduce the complexity of FPGA place and route, and create designs with good timing properties, we only uses one single PCIe lane (up to eight lanes are allowed in hardware) and only one host-device channel (up to four channels are allowed in hardware). Based on our experiment results, using more PCIe lanes or host-device channels will result in bad circuit timing and the design frequency will be low. This narrow CPU-FPGA interface becomes a bottleneck. The performance of our accelerator solution can be further improved with better optimized CPU-FPGA inference. This will be done as a future work.

Given enough DRAM bandwidth, our solution can be easily scaled to larger FPGAs with more accelerators or even multiple FPGAs, as our accelerators can operate independently on different set of input samples. This way the parallelism in input samples and neurons in the layers can be further exploited, as we discussed in Section 3.3. With more accelerator instances and multiple FPGAs, our solution should be able to outperform the multi-core CPU solution even in terms of execution time.

3.6 Related Works

In this work, we focus on hardware acceleration of sparse deep neural networks (DNNs). Converting dense deep neural networks into sparse ones is

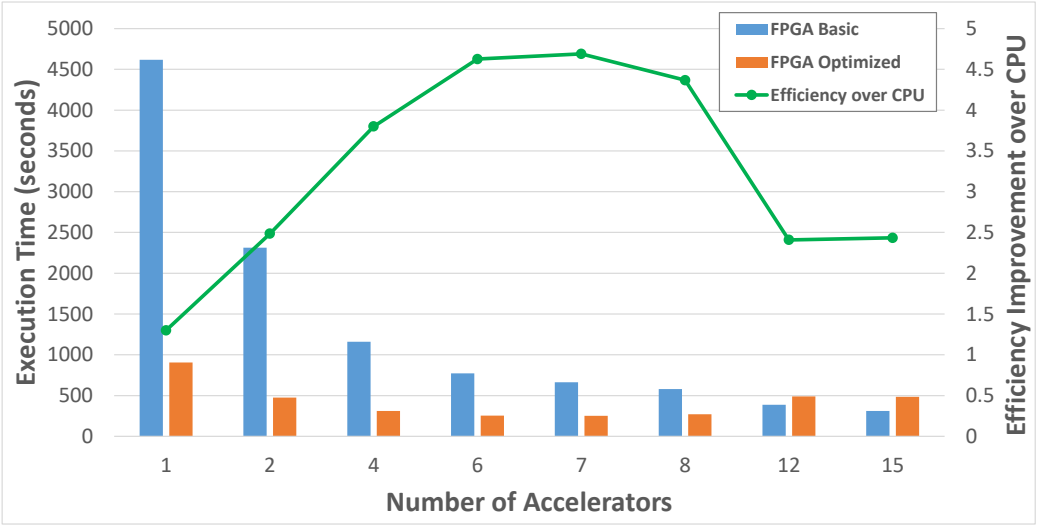


Figure 3.4: Sparse DNN Accelerator Performance

out of the scope of this work. The computation in sparse DNNs we focus on is essentially sparse vector-matrix multiplication with non-linear activation functions. There are several works on acceleration of sparse matrix vector multiplication. Fowers et al. [14] proposed an FPGA design for sparse matrix-vector multiplication. The accelerator is designed in RTL code. The design consumes 25 W and achieve 2.6× and 2.3× higher energy efficiencies than CPU and GPU. The performance of the design is around two thirds of CPU performance and one third of GPU performance. This work uses Compressed Interleaved Sparse Row (CISR) matrix encoding which enables simultaneous multiply-accumulate operations on multiple rows of the matrix. The problem solved here is similar to our work, the major difference is that our work focuses on sparse DNNs specifically instead of sparse matrix vector multiplication. Also, our sparse DNN inference engine is parameterized and is capable of exploiting various types of parallelism and data reuse opportunities.

Giefers et al. [15] did a thorough comparison of the energy efficiency of sparse matrix multiplication on CPU, Xeon Phi and FPGAs, in the context of heterogeneous systems. The FPGA platform in this work is Nallatech 385N FPGA board, which contains an Altera Stratix V FPGA. The design is done with OpenCL SDK for FPGA. The evaluation results show that FPGA is remarkably efficient. This work focuses on energy efficiency comparison across platforms, and the FPGA design uses the general OpenCL code which

may not be best optimized for FPGA and the design flow.

Besides, there are several recent works that focus on FPGA acceleration of sparse convolutional neural networks (CNN) [16], [17] and sparse long short-term memory (LSTM) [18]. These works accelerate the CNNs and LSTMs while this work focuses on very deep fully connected networks. The computation inside sparse CNN and LSTM have similar memory random access patterns as sparse matrix vector multiplication. However, sparse CNNs and LSTMs have unique data dependency patterns, therefore the high-level data access and computation patterns in these works are different from this work. Besides, this work targets very deep and wide networks which are generally larger than the networks used in current applications.

3.7 Conclusion

In this work, we proposed and built a configurable sparse DNN inference engine. The proposed inference engine is parameterized and it can be configured to have different sizes and different processing capabilities. The inference engine can be adopted in both edge computing and high-performance computing scenarios. We also modeled and analyzed the computation of sparse DNN inference, parameterized sparse DNN hardware design, and presented the design space of the sparse DNN accelerators. The proposed design was evaluated on Xilinx VC709 FPGA board. Evaluation results show that the proposed design achieve up to 4.7× better energy efficiency compared to CPU.

CHAPTER 4

DESIGN AND OPTIMIZATION OF HETEROGENEOUS SYSTEMS

Heterogeneous CPU-FPGA systems are evolving toward tighter integration between CPUs and FPGAs for improved performance and energy efficiency. At the same time, programmability is also improving with high level synthesis (HLS) tools (e.g., OpenCL Software Development Kits), which allow programmers to express their designs with high-level programming languages, and avoid time-consuming and error-prone register-transfer level (RTL) programming. In the traditional loosely coupled accelerator mode, FPGAs work as offload accelerators, where an entire kernel runs on the FPGA while the CPU thread waits for the result. However, tighter integration of the CPUs and the FPGAs enables the possibility of fine-grained collaborative execution, i.e., having both devices working concurrently on the same workload. Such collaborative execution makes better use of the overall system resources by employing both CPU threads and FPGA concurrency, thereby achieving higher performance. In this chapter, we explore the potential of collaborative execution between CPUs and FPGAs using OpenCL HLS tools. First, we compare various collaborative techniques (namely, data partitioning and task partitioning), and evaluate the tradeoffs between them. We observe that choosing the most suitable partitioning strategy can improve performance by up to 2×. Second, we study the impact of a common optimization technique, kernel duplication, in a collaborative CPU-FPGA context. We show that the general trend is that kernel duplication improves performance until the memory bandwidth saturates. Third, we provide new insights that application developers can use when designing CPU-FPGA collaborative applications to choose between different partitioning strategies. We find that different partitioning strategies pose different tradeoffs (e.g., task partitioning enables more kernel duplication, while data partitioning has lower communication overhead and better load balance), but they generally outperform execution on conventional CPU-FPGA systems where no collaborative exe-

cution strategies are used. Therefore, we advocate even more integration in future heterogeneous CPU-FPGA systems (e.g., OpenCL 2.0 features, such as fine-grained shared virtual memory).

4.1 Introduction

The demand for processing larger amounts of data with higher performance under constrained power and energy budgets makes *heterogeneity* a fundamental feature of computing systems. Therefore, heterogeneous architectures (e.g., CPU-GPU, CPU-FPGA) are now ubiquitous in modern data centers and supercomputers [19], [20], [21]. In addition to powerful CPUs, current computing systems typically employ various types of specialized devices, such as GPUs, FPGAs, tensor processing units (TPUs), and other ASICs. FPGAs are particularly interesting because they provide a tradeoff between performance and programmability, when programmed with High Level Synthesis (HLS) frameworks, like Intel FPGA SDK [22] for OpenCL and Xilinx SDAccel [23]. FPGAs are not only suitable for accelerating applications under stringent energy efficiency requirements [24], [25], [26], but they are also being increasingly adopted in cloud servers and data centers [27], [28], [29], [30], [31], [32].

For example, Microsoft has built an Earth-scale FPGA-based network (the Catapult V2 [28], [29]), which enables network flows to be programmably transformed at line rate in the cloud, thereby accelerating both network functions and applications. Other major efforts using FPGAs in the cloud include IBM SuperVesselCloud [30], Amazon EC2 [27], Microsoft Brainwave [33], and the Intel CPU-FPGA deep learning inference accelerator card (DLIA) [31]. Intel estimates that FPGAs will run in 30% of data center servers in 2020 [32].

Traditionally, accelerators (including FPGAs) have been used as *offload engines*, where an entire kernel runs on the accelerator while the CPU remains idle, waiting for the result [34], [35], [36], [37]. More recently, vendors provide interconnect technologies such as Intel QuickPath Interconnect (QPI) [38], Hyper Transport [39], Front Side Bus (FSB) [40], Accelerator Coherency Port (ACP) [41], AXI Coherency Extension (ACE) [42], ARM CoreLink Interconnect [43], IBM Coherent Accelerator Processor Interface (CAPI) [44], and Cache Coherent Interconnect for Accelerators (CCIX) [45]. In terms

of functionality, these interconnects operate in a similar manner, but their details vary across CPU architectures, processor implementations, and silicon fabrication. These interconnects enable tighter integration between CPUs and FPGAs in SoC chips [46], [47], [48] and server-grade systems [49], [50].

The trend toward tighter integration of CPUs and FPGAs enables more *collaborative execution* between devices. Rather than executing an entire kernel on the FPGA while the CPU is idle, collaborative execution makes better use of the overall system resources by involving *both* CPU threads and FPGA in the execution. One of the key challenges of collaborative execution between CPUs and FPGAs is the identification of the best strategy for partitioning work between the CPU and the FPGA. There are two major approaches to partitioning of work. The first approach, called *data partitioning*, is to have the CPU and the FPGA perform the same task on different subsets of the data. The second approach, called *task partitioning*, is to have each device perform a different sub-task and communicate intermediate results between them. Each partitioning strategy entails its own tradeoffs, and different applications may benefit from different strategies. The factors that impact the suitability of each partitioning strategy encompass (1) the latency and bandwidth of inter-device communication, (2) the disparity in the workload's performance on the CPU versus the FPGA, (3) the diversity of computation phases within a task, and (4) the hardware resource constraints. Each strategy poses its own challenges, such as how much data to assign to each device or which sub-tasks to assign to which device. Our goals in this work are (1) to evaluate different collaborative execution strategies for CPU-FPGA systems by analyzing their effectiveness and their tradeoffs, and (2) to provide insights for designing future CPU-FPGA collaborative applications. Though our work focuses on integrated CPU-FPGA systems, it could be extended to collaborative execution using other types of accelerators in heterogeneous systems.

We make the following contributions:

- We carry out the first quantitative evaluation of collaborative execution strategies with OpenCL HLS on CPU-FPGA systems using benchmarks from diverse fields (e.g., image processing, graph processing, producer-consumer computing, computer graphics, etc.).

- We propose new analytical models for different collaborative execution

strategies that assist us in estimating their performance of different strategies.

- We rigorously analyze the tradeoffs of different partitioning strategies for collaborative execution, and provide insights to help developers make informed decisions when designing collaborative programs for CPU-FPGA systems.

4.2 Collaborative Execution Strategies

We first define collaborative execution and two main strategies for it, namely data partitioning and task partitioning. Then, we propose analytical models to estimate the performance of data partitioning and task partitioning.

Collaborative execution refers to an application execution structure where the CPU and the FPGA (or another accelerator) both participate in performing the computations required by the application, as opposed to the traditional offload accelerator model where the entire kernel is executed on the FPGA while the CPU thread waits for the result. The strategies for collaboratively executing a program on different types of devices can be classified into two main categories: data partitioning and task partitioning. We discuss these in Sections 4.2.1 and 4.2.2 respectively. Many programs are amenable to both data partitioning and task partitioning, and thus programmers need to choose between them. In this chapter, we aim to provide insights to assist programmers in making the right decisions when writing collaborative programs for integrated CPU-FPGA systems.

Throughout this section, we illustrate the collaborative execution strategies using the simple example shown in Figure 4.1(a). In this example, a program consists of many data-parallel tasks ❶ that are applied to different data elements. Each data-parallel task consists of multiple types of sub-tasks (two in this case), and the result of the first sub-task ❷ is required for the execution of the second sub-task ❸. In some cases, a program may consist of multiple phases where there is a global synchronization point ❹ across all data-parallel tasks. In OpenCL, these phases are typically expressed as separate *kernels*, since global synchronization across the entire device is not supported in the programming model.

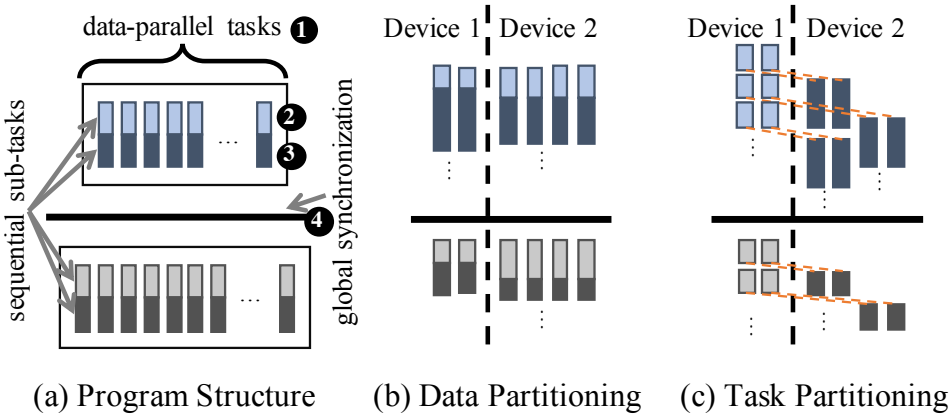


Figure 4.1: Program with Many Data-Parallel Tasks (a) and Two Collaborative Execution Strategies: Data Partitioning (b) and Task Partitioning (c)

4.2.1 Data Partitioning

Data partitioning is a collaborative execution strategy wherein different devices perform the *same task on a different subset of the data*, i.e., the data-parallel tasks are distributed across devices. Figure 4.1(b) illustrates this strategy. The main challenge with data partitioning is determining the optimal partitioning, i.e., the distribution of data-parallel tasks across devices that results in the highest performance. One possibility is *static partitioning* where a fixed fraction of the data-parallel tasks is statically assigned to each device prior to execution. Another possibility is *dynamic partitioning* where the data-parallel tasks are dynamically assigned to different devices from a task pool during execution until all tasks are exhausted.

To better understand and analyze the collaborative execution patterns, we establish analytical models. We use the abstraction of *workers* to represent the processing units on a device. A thread on a CPU is a CPU worker. A processing element on an FPGA is considered an FPGA worker. We use the

following notation to describe the application and system properties:

N – Number of data parallel tasks in the application

$t_{i,C}$ – Execution time of sub-task i by a CPU worker

$t_{i,F}$ – Execution time of sub-task i by an FPGA worker

w_C – Number of available CPU workers

w_F – Number of available FPGA workers

To define an analytical model for data partitioning, let α be the fraction of data-parallel tasks executed by the CPU, and let β_{data} be the factor of increase in the total execution time (i.e., overhead) due to distributing tasks and merging partial results. The total execution time of the application is expressed as:

$$t_{data,total} = \beta_{\text{data}} \cdot max\left(\frac{\alpha N \sum_i t_{i,C}}{w_C}, \frac{(1-\alpha) N \sum_i t_{i,F}}{w_F}\right) \tag{4.1}$$

The overall execution time is the maximum of the execution times on the CPU and the FPGA since the device that finishes first needs to wait for the other device.

To minimize the total execution time when performing data partitioning, α must be tuned such that the two terms in $max(\cdot, \cdot)$ are equal. This ensures load balance, and thus minimizes device idleness and the overall execution time.

The optimal α^* can be obtained as:

$$\alpha^* = \frac{\sum_i t_{i,F}}{w_F} \Big/ \left(\frac{\sum_i t_{i,C}}{w_C} + \frac{\sum_i t_{i,F}}{w_F}\right) \tag{4.2}$$

The optimal α^* is therefore determined by $t_{i,C}$, $t_{i,F}$, w_C, and w_F, which are specific to the application and the system. Statically determining the optimal α^* requires profiling the program and a performance model of the system.

Alternatively, to maximize the performance of data partitioning without the need for program profiling and a system performance model, *dynamic* data partitioning can be used. As opposed to static data partitioning where α is determined and fixed before execution, dynamic data partitioning does not partition the data using a fixed ratio. Instead, data is partitioned into

fine-grained data blocks and those fine-grained data blocks are dynamically assigned to devices (and workers) from a task pool until all tasks are exhausted. The fraction of total data blocks that each device ends up processing is largely dependent on the relative performance difference of the different devices involved. With dynamic data partitioning, load balance between devices can be achieved. However, dynamic data partitioning might have a higher β_{data} due to the additional overheads caused by contention on the task queue.

4.2.2 Task Partitioning

Task partitioning is a collaborative execution strategy wherein different devices execute *different types of sub-tasks on the entire set of the data*, i.e, within each data-parallel task, different types of devices perform different types of sub-tasks. Figure 4.1(c) illustrates this strategy. The main challenge with task partitioning is to determine which type of sub-tasks within the data-parallel tasks is more suitable for each device. Even if one device is better at all types of sub-tasks, task partitioning may still be beneficial, if it makes better utilization of the devices that might be otherwise idle, thus improving parallelism. Since the sub-tasks within a data-parallel task are sequential, task partitioning creates a dependency between devices such that one device must wait for intermediate results from another device before executing its sub-task. However, with multiple tasks available, significant parallelism can still be achieved across devices via pipeline-style (i.e., pipeline-parallel) execution, as illustrated in Figure 4.1(c).

To define an analytical model for task partitioning, we use the same notation defined in Section 4.2.1. In addition, let S_C and S_F be the sets of indices of sub-tasks to be executed on the CPU and the FPGA respectively. Note that $S_C \cap S_F = \emptyset$ and $S_C \cup S_F$ is the set of indices of all sub-tasks. Also let β_{task} be the percentage of total execution time increase (i.e., overhead) due to communication overhead in task partitioning. In task partitioning, the execution of sub-tasks on the CPU and the FPGA may or may not be overlapped, depending on the granularity of task partitioning and pipelining. If the target platform supports fine-grained task partitioning, which allows the CPU processing and the FPGA processing to perfectly overlap with each

other, the total execution time depends on the execution time of the device that takes longer to finish. The total execution time of the application in this case can be expressed as:

$$t_{task,total} = \beta_{\text{task}} N \cdot max \left(\frac{\sum_{i \in S_C} t_{i,C}}{w_C}, \frac{\sum_{i \in S_F} t_{i,F}}{w_F} \right) \tag{4.3}$$

If the target platform only supports coarse-grained task partitioning, where no overlap between CPU processing and accelerator processing is possible, the total execution time is the sum of the execution time of the CPU and the FPGA:

$$t_{task,total} = \beta_{\text{task}} N \cdot \left(\frac{\sum_{i \in S_C} t_{i,C}}{w_C} + \frac{\sum_{i \in S_F} t_{i,F}}{w_F} \right) \tag{4.4}$$

More generally, in cases where the CPU and the FPGA processing have some level of overlap, the total execution time must fall into the range given by Equations 4.3 and 4.4. Therefore, Equation 4.3 and Equation 4.4 give the lower bound and upper bound, respectively, of task partitioning execution time.

Optimizing the performance of task partitioning increases in difficulty as the number of sub-tasks increases, since the number of combinations of S_C and S_F grows exponentially with respect to the number of sub-tasks. Thus, minimizing Equations 4.3 and 4.4 is not straightforward. It requires a performance model of the underlying hardware or manual effort, which are out of the scope of this work. This work mainly focuses on how collaborative execution patterns can be better used, and any automatic or manual optimization of Equations 4.3 and 4.4 is orthogonal to our work.

4.3 Methodology

We use OpenCL programs from the Chai benchmark suite [51], which is developed to evaluate collaborative execution. Table 4.1 shows the benchmarks we evaluate, along with the collaborative execution strategy used by each one. The benchmarks are compiled with the Intel FPGA SDK for OpenCL 16.0 [22]. For the comparative evaluation of the two collaborative execution strategies in Section 4.4, we use Canny Edge Detection and Random Sample

Consensus, since these two benchmarks support both partitioning schemes.

Table 4.1: Evaluated Chai Benchmarks [51]

Benchmark	**Description**	**Strategy**
CED-D	Canny Edge Detection	Data Partitioning
CED-T	Canny Edge Detection	Task Partitioning
RSC-D	Random Sample Consensus	Data Partitioning
RSC-T	Random Sample Consensus	Task Partitioning
BS	Bézier Surface	Data Partitioning
HSTO	Image Histogram	Data Partitioning
SSSP	Single-Source Shortest Path	Task Partitioning
TQ	Task Queue System (Synthetic)	Task Partitioning
TQH	Task Queue System (Histogram)	Task Partitioning

We perform our evaluation on the two systems shown in Table 4.2. Note that the Nallatech 510T data center acceleration card hosts two Arria 10 1150 GX FPGAs, but the current OpenCL Board Support Package (BSP) from the vendor is a beta version that is limited to one FPGA and two DDR4 slots, so only one FPGA and 8GB device memory are used for the evaluation. The Arria 10 FPGA has more logic and DSP resources than the Stratix V FPGA does, and also features hard floating-point DSP blocks, which the Stratix V FPGA does not.

Table 4.2: System Specifications

FPGA Board	Terasic DE5-Net [52]	Nallatech 510T [53]
FPGA	Stratix V GX [54]	Arria 10 GX [55]
Device Memory	4GB (DDR3)	8GB (DDR4)
Host CPU	Xeon E3-1240 v3 [56]	Xeon E5-2650 v3 [57]
Host Memory	8GB (DDR3)	96GB (DDR4)
Interface	PCIe gen3.0 ×8	PCIe gen3.0 ×8

In the experiments, we repeat the execution and measurement five times for each test. The reported execution time is the averaged execution time of five runs.

In Figures 4.3 and 4.4, we show results for both FPGAs, which summarize the comparison between data and task partitioning for CED and RSC. For the remaining results, we only show results for the Stratix V FPGA for brevity, but the trends are similar on both systems we evaluate. Unless otherwise specified, we report the results for the best performing CPU thread count

(from among one, two, and four threads) and the best performing duplication factor. Besides, we use the built-in Intel FPGA dynamic profiler for OpenCL provided by the Intel OpenCL SDK to profile the execution of FPGA kernels and identify performance bottlenecks.

4.4 Evaluation of Collaborative Execution Strategies

In this section, we evaluate the performance of CPU-FPGA collaborative execution and analyze the sources of the performance improvements and bottlenecks.

4.4.1 Canny Edge Detection

Canny Edge Detection (CED) [58] is an edge detection algorithm that is commonly used for image processing. It consists of four stages: (1) a Gaussian filter, (2) a Sobel filter, (3) non-maximum suppression, and (4) hysteresis. We apply these four stages of the algorithm to a stream of video frames. In the data partitioning version of the benchmark, each device processes a different set of frames. In the task partitioning version, the first two stages are executed on the FPGA while the remaining two stages are executed on the CPU for all frames. We choose this style of task partitioning because Gaussian and Sobel filters are more regular whereas non-maximum suppression and hysteresis contain more control flow for which CPUs are well-optimized.
Data Partitioning (CED-D). Figure 4.2 shows the execution time of the CED-D benchmark for different data partitioning distributions. The values on the horizontal axis indicate the fraction of frames processed by the CPU in static partitioning swept in increments of 0.1, with the last pair of bars showing the results for dynamic partitioning. Here, static partitioning statically assigns a subset of frames to process to each device, while dynamic partitioning uses one CPU control (proxy) thread for each device (CPU or FPGA), fetching frames and sending them to the corresponding device, as soon as the device is available. In this benchmark, the overhead due to the control threads is negligible, since the granularity of partitioning is coarse (an entire frame). The execution time is broken down into compute time, copy time (for the FPGA only), and idle time (the time a device waits for

the other device to finish).

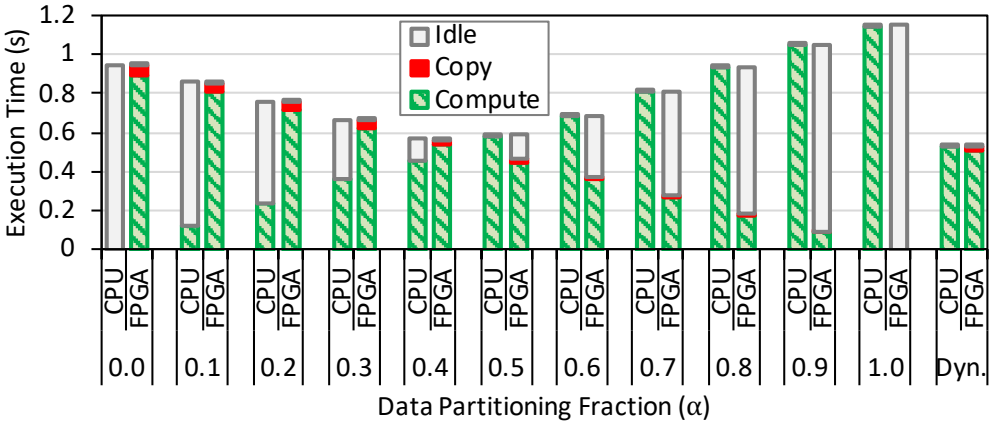


Figure 4.2: Execution Time of Canny Edge Detection (CED) with Different Data Partitioning Fractions (α) and with Dynamic Data Partitioning. α is the Fraction of Data-Parallel Tasks Assigned to the CPU

From the results, we make three major observations. First, processing all frames on the FPGA (shown by the $\alpha = 0.0$ bars) achieves shorter execution time than processing all frames on the CPU (shown by the $\alpha = 1.0$ bars). Second, the data partitioning strategy outperforms both the CPU and the FPGA. For this particular workload, the sweet spot for static partitioning (among the tested distributions) is $\alpha = 0.4$ where the CPU processes 40% and the FPGA processes 60% of the video frames. Third, dynamic partitioning eliminates the idle time completely, thus outperforming the best static partitioning, and providing the lowest execution times.

Task Partitioning (CED-T). Figure 4.3 compares the execution time of collaborative execution with task partitioning to data partitioning and to no collaboration. The execution time is broken down into compute time, copy time (for the FPGA only), and idle time (the time the device waits for the other device to finish).

We make three major observations. First, the overall best performance of task partitioning is comparable to the overall best performance of data partitioning. Second, task partitioning incurs more communication overhead, with copy time accounting for 11.4% of overall execution time in task partitioning and only 4.4% in data partitioning. One reason is that with task partitioning, data from all data-parallel tasks must be copied to the FPGA, while with data partitioning, only a subset of the data needs to be copied. It

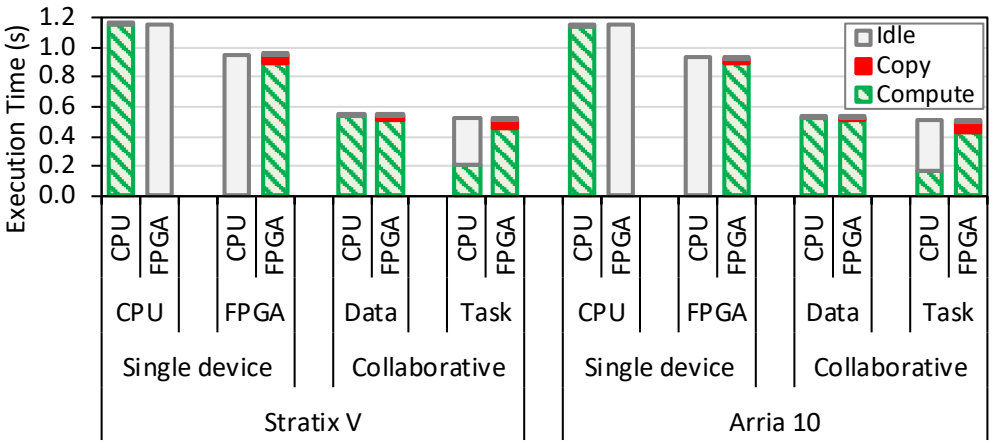


Figure 4.3: Execution Time of Canny Edge Detection (CED) across CPU-FPGA Systems and Collaborative Execution Strategies

is expected that the availability of coherent memory will make this communication overhead less of an issue. This is particularly important for workloads that are less compute-intensive, making copy-time a larger fraction of the total execution time. Third, in task partitioning, there is still some idle CPU time, whereas in data partitioning, this idle time is negligible due to dynamic data partitioning. Despite that, the performance of both strategies is comparable, as pointed out in our first observation. This fact represents a potential advantage of task partitioning over data partitioning, where different devices are more suitable or specialized for different workloads. If the sub-tasks assigned to the CPU in task partitioning were more time-consuming, the CPU could still use that fraction of idle time. However, the overall execution time of data partitioning would increase, since in data partitioning all sub-tasks run on *all* devices.

From the comparison of CED-D and CED-T, we derive three major conclusions. First, in data partitioning, finding the best load balance across devices is possible with static or dynamic partitioning. Second, task partitioning can greatly benefit from device specialization. Third, communication overhead in task partitioning is larger than that in data partitioning.

4.4.2 Random Sample Consensus

Random Sample Consensus (RSC) [59] is an algorithm for estimating the parameters of a model by taking random samples of an input iteratively until

a successful model is found. A single iteration of RSC consists of two stages: (1) model fitting using random samples and (2) evaluating the model accuracy by computing outliers and error values. The iterations are independent and can be done in parallel [60]. In data partitioning, each device processes a different set of iterations. In task partitioning, the first stage is executed on the CPU while the second stage is executed on the FPGA. We do so because the first stage is inherently sequential while the second stage is massively parallel, and thus better suited for the FPGA.

Data Partitioning (RSC-D). Figure 4.4 shows the execution time of RSC for different static data partitioning fractions. We make two observations. First, the static partitioning sweet spot for RSC-D is at 50% for each device, which is different from CED, highlighting the need to optimize data partitioning strategies individually for different applications. Second, the performance of RSC-D is lower on the Arria 10 than on the Stratix V, mainly because the clock frequency of the FPGA implementation of this kernel is lower on the Arria 10. The reason for the lower clock frequency could come from either the beta BSP of Nallatech 510T (see Section 4.3) or better optimization on the BSP of Terasic DE5-Net.

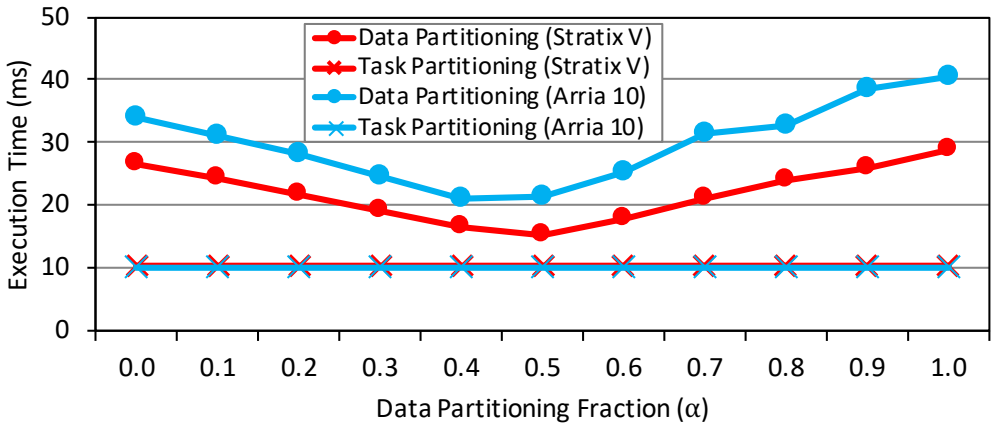


Figure 4.4: Execution Time of Random Sample Consensus (RSC) across CPU-FPGA Systems and Collaborative Execution Strategies. α is the Fraction of Data-Parallel Tasks Assigned to the CPU in Data Partitioning

The RSC implementation *cannot* perform dynamic data partitioning because the granularity of partitioning is smaller than that in CED. In CED, each data-parallel task is the processing of an independent frame with four OpenCL kernels launched by a CPU proxy thread. Thus, the CPU proxy

threads perform dynamic partitioning by accessing a pool of data-parallel tasks via a shared atomic variable. However, in RSC, a single OpenCL kernel is launched to compute *all* data-parallel tasks. Within this kernel, an FPGA worker (an OpenCL *work-group*) computes each data-parallel task. Since current CPU-FPGA systems do *not* support OpenCL 2.0 shared virtual memory and system-wide atomic instructions, it is not possible for FPGA workers to access the same atomic variable as CPU threads. Thus, dynamic data partitioning is not possible for RSC. OpenCL 2.0 shared virtual memory and system-wide atomic instructions are desirable architectural features in future CPU-FPGA systems for more effective collaborative execution.

Task Partitioning (RSC-T). Figure 4.4 shows that RSC with task partitioning noticeably outperforms RSC with the best data partitioning. The reason is that data partitioning exhausts the DSP blocks in the FPGA, since the first stage of RSC employs intensive floating-point computations. As task partitioning assigns the first stage to the CPU, the FPGA can utilize more resources for the second stage. This enables a higher degree of kernel duplication, a common optimization technique that duplicates the number of processing elements, which potentially translates into a significant performance improvement. We discuss kernel duplication in detail in Section 4.5.

4.4.3 Other Data Partitioning Benchmarks

Bézier Surface (BS). Bézier surfaces are parametric structures widely used in computer graphics and finite element modeling. This benchmark uses non-rational formulation of Bézier surfaces on a regular 2D surface. BS performs data partitioning on the output data by dividing the output surface into square tiles, which are assigned to different CPU threads and different OpenCL work-groups [61].

Figure 4.5 shows the execution time breakdown of BS running on the Arria 10 FPGA, with the data partitioning fraction α ranging from 0 to 1. We make two major observations. First, the kernel time varies as data partitioning fraction α changes, with $\alpha = 0.7$ minimizing the execution time. Second, for all the α's, most of the total execution time is spent on the kernel computation. The other parts of execution (e.g., data copy, allocation, and deallocation) account for a very small fraction of the total execution time.

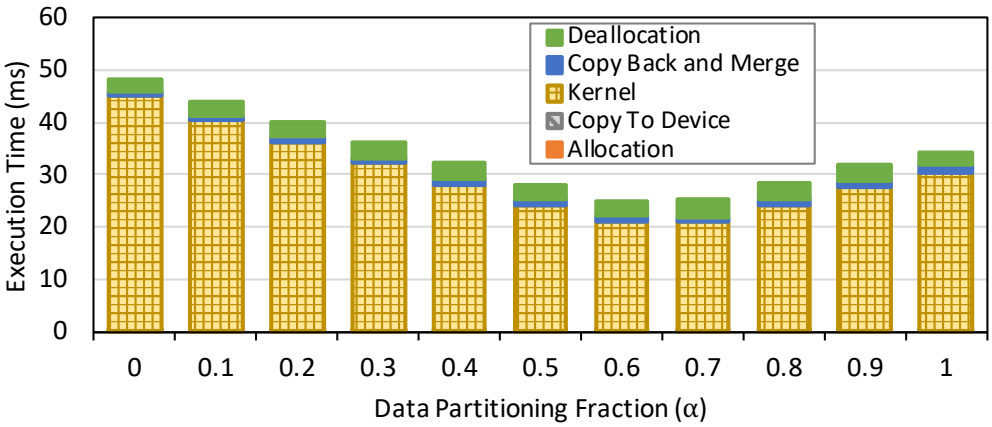


Figure 4.5: Execution Time of Bézier Surface (BS) with Different Data Partitioning Fractions (α). α is the Fraction of Data-parallel Tasks Assigned to the CPU

Image Histogram (HSTO). A histogram describes the frequency of data falling into each of some predefined bins. Histogram computation is a frequently used routine in many applications [62, 63], for example, image processing and pattern recognition. This benchmark implements the histogram computation of pixels of an image by binning pixels based on their value ranges. It uses data partitioning on the output bins, i.e., part of the bins are assigned to the CPU and the other part to the FPGA. Both the CPU and the FPGA process the entire set of input data, but they increase a bin counter only if an input data value falls into their assigned set of bins. This way, the CPU and the FPGA do not update the same bin counters.

Figure 4.6 shows the execution time breakdown of HSTO on the Arria 10 FPGA with various α values. A major observation is that the overall execution time of HSTO is almost *independent* of the data partitioning factor α. The reason is that in HSTO, both the CPU and the FPGA workers need to traverse through the large input data, the overhead of which overwhelms the benefit from data partitioning, leading to no performance improvement from any level of data partitioning.

4.4.4 Other Task Partitioning Benchmarks

Single-Source Shortest Path (SSSP). SSSP is a commonly used graph algorithm that identifies the path between two vertices in a graph that has

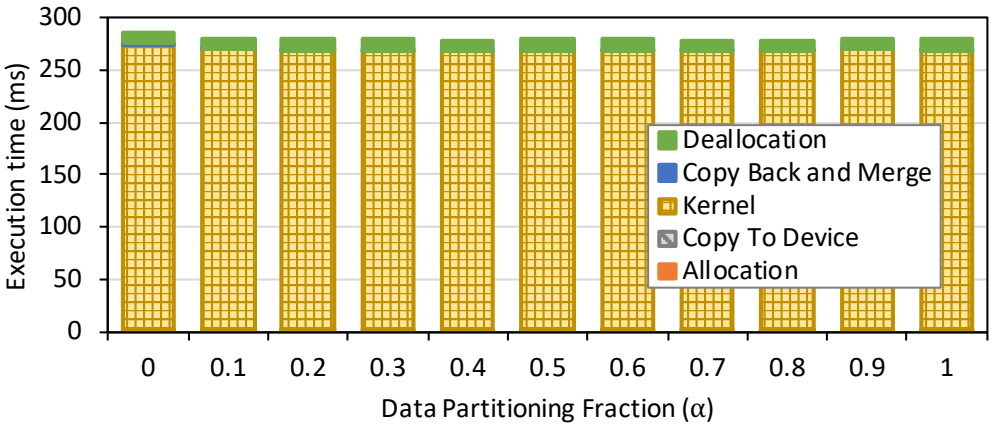


Figure 4.6: Execution Time of Histogram (HSTO) with Different Data Partitioning Fractions (α); α is the Fraction of Data-Parallel Tasks Assigned to the CPU

the minimal sum of weights of edges on the path. This benchmark has irregular memory access patterns and requires atomic instructions. SSSP performs coarse-grained task partitioning across a series of tasks, each of which is an iteration of the algorithm which constructs a frontier of vertices (i.e., the list of vertices to visit in the next iteration) and updates the shortest distance. The frontier of vertices is a queue data structure for communicating among tasks. Since graph structures are irregular, the size of vertex frontiers varies, which causes imbalance in processing times of different tasks. In SSSP, iterations are assigned to the CPU or the FPGA to process the frontier according to the *frontier size*. Small frontiers are assigned to the CPU while the large ones are assigned to the FPGA. This is based on the observation that the FPGA performs better for larger frontiers, where more parallelism can be exploited and the kernel launch overhead is less significant. On the CPU, CPU threads dequeue vertices from a vertex queue and process them sequentially. On the FPGA, different OpenCL *work-items* process different vertices and aggregate results with atomic instructions.

Task Queue Systems (TQ and TQH). Task queue systems (TQ and TQH) exemplify a type of the producer-consumer computing pattern, where the host enqueues tasks into some task queues in the device memory, while the device dequeues and processes the tasks. TQ works with synthetic data, while TQH generates the histograms of video frames (i.e., the input of each task is a video frame, and the output is its histogram). In both benchmarks,

the CPU threads generate and enqueue tasks, while the FPGA processes the tasks. As soon as the FPGA finishes with one task, it dequeues a new task. This way, the workload across work-groups is balanced.

We evaluate the effect of kernel duplication for SSSP, TQ and TQH in Section 4.5.

4.5 Evaluation of Kernel Duplication

In this section, we evaluate the effect of a common optimization technique, kernel duplication [22], on the performance of the benchmarks described in Section 4.4. With kernel duplication, multiple identical hardware instances (processing elements) are instantiated on the FPGA from the same OpenCL kernel. The Intel OpenCL SDK for FPGAs provides a programming attribute to specify the *duplication factor* of OpenCL kernels, i.e., the number of identical processing elements on the FPGA. The OpenCL work-items are executed on these hardware processing elements in a SIMD manner, which can potentially improve performance. With kernel duplication, the utilization of the available configurable logic on the FPGA increases, since there are more processing elements. However, the higher resource utilization increases the complexity of place and route on the FPGA circuit, and therefore could potentially lower the operating frequency and increase the execution time. A higher number of processing elements can also lead to memory access contention in the memory system [64, 65, 66, 67, 68, 69, 70].

With kernel duplication, the number of FPGA workers w_F in the analytical model presented in Section 4.2 increases, while the execution time of each individual sub-task $t_{i,F}$ may increase due to changes in frequency and resources. Therefore, the tuning of the kernel duplication factor is dependent on the tradeoff between the the ability to exploit more parallelism and the overhead of having multiple processing elements on the FPGA.

4.5.1 Performance Effect of Kernel Duplication

Figure 4.7 shows the impact of possible kernel duplication factors on the overall performance of four data-partitioning benchmarks (CED-D, RSC-D, BS, and HSTO) with various data partitioning factors α. As shown in the

figure, in the FPGA-only case ($\alpha = 0$), RSC-D, BS and HSTO benefit from duplicating kernels. For CED-D, kernel duplication does not change performance significantly. As discussed in Section 4.4, HSTO has a different behavior from the other benchmarks in terms of data partitioning – HSTO's performance is almost independent of the α value. The reason is that, in HSTO, partitioning happens only on output bins, and both CPU and FPGA need to traverse through a large amount of input data, the overhead of which hides the benefit of partitioning. In HSTO, kernel duplication is still beneficial, since the computation capacity of the FPGA and, thus, the whole system increases with kernel duplication.

We make three observations from Figure 4.7. First, a higher duplication factor does not necessarily lead to higher performance because of the tradeoffs we discuss above. Second, when kernel duplication is actually beneficial, the best α values tend to be smaller (i.e., more workload assigned to the FPGA) for higher duplication factors, since the computation capacity of the FPGA increases. Third, for larger values of α, kernel duplication has almost no impact on overall execution time, since most of the workload of the applications is assigned to the CPU.

Figure 4.8 shows the speedups from kernel duplication on three task-partitioning benchmarks (SSSP, TQH, and TQ). As shown in the figure, SSSP does not benefit much from kernel duplication because its irregular memory access pattern quickly saturates the memory bandwidth. On the other hand, in TQH and TQ, performance scales well with the duplication factor due to regularity of their memory access patterns and load balancing across work-groups.

Figures 4.9, 4.10 and 4.11 show the execution time breakdowns for SSSP, TQ, and TQH, respectively, on the Arria 10 FPGA. As the figures show, across all three benchmarks, kernel duplication reduces the kernel execution time without affecting other portions of the total execution time.

4.5.2 Analysis of Resource Utilization

We further analyze how kernel duplication changes FPGA resource utilization and how it impacts performance. For this analysis, we focus on canny edge detection (CED) and random sample consensus (RSC).

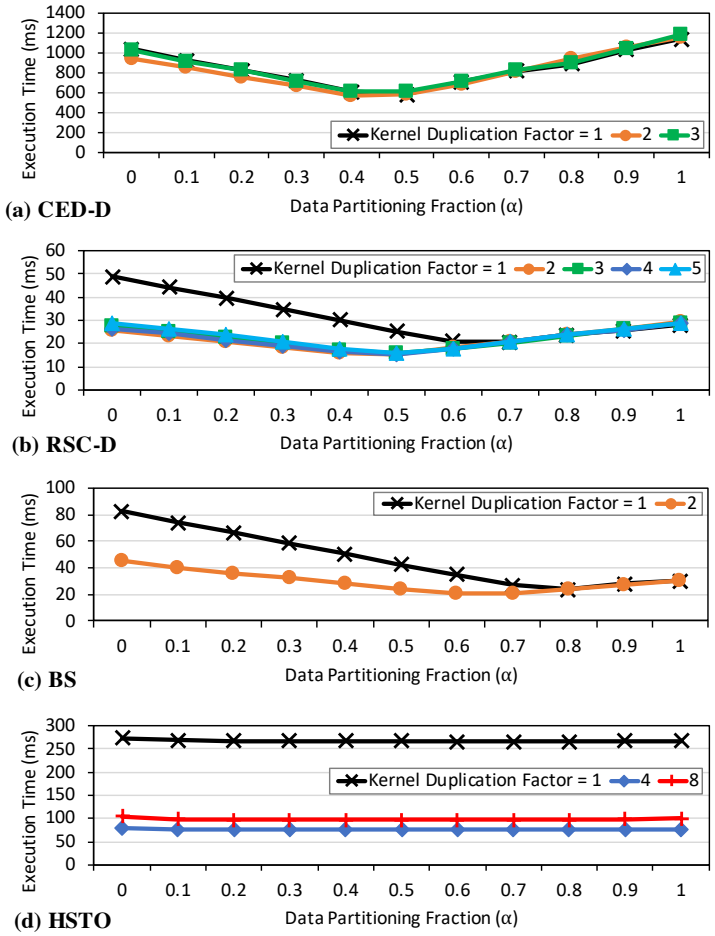


Figure 4.7: Execution Time of Data Partitioning Benchmarks (CED-D, RSC-D, BS, and HSTO) for Different Kernel Duplication Factors and Data Partitioning Fractions (α); α is the Fraction of Data-parallel Tasks Assigned to the CPU

Canny Edge Detection (CED-D/CED-T). Figure 4.12 shows the effect of the kernel duplication factor on performance, as well as other utilization metrics. Mapping to the axis on the left, the light-blue bar represents performance (higher is better) in terms of the speedup over data partitioning with a duplication factor of 1, and the black line represents the frequency of the FPGA processing elements also normalized to data partitioning with a duplication factor of 1. Mapping to the axis on the right, the different lines represent the utilization in terms of percentage of the logic, DSP blocks, and RAM blocks on the FPGA.

We make two major observations. First, the duplication factor has little impact on the performance of CED, and too much duplication may

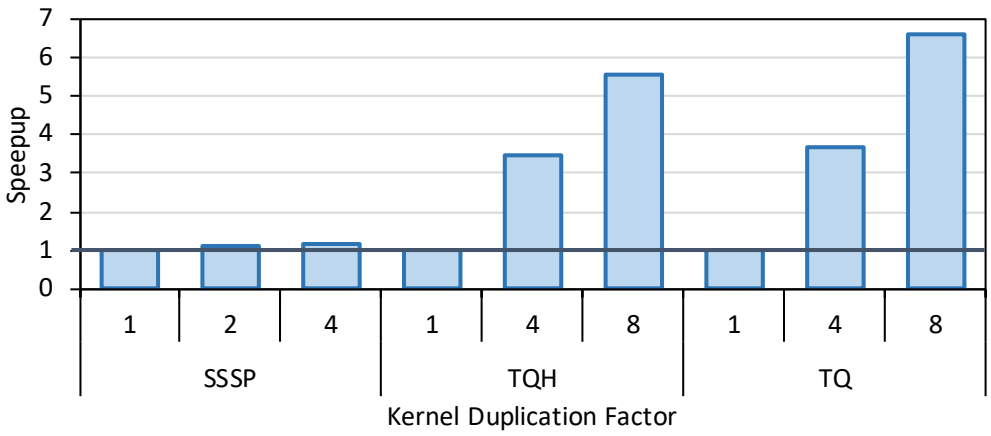


Figure 4.8: Speedup (Normalized to Kernel Duplication Factor 1) of Task Partitioning Benchmarks (SSSP, TQ, and TQH) for Different Kernel Duplication Factors

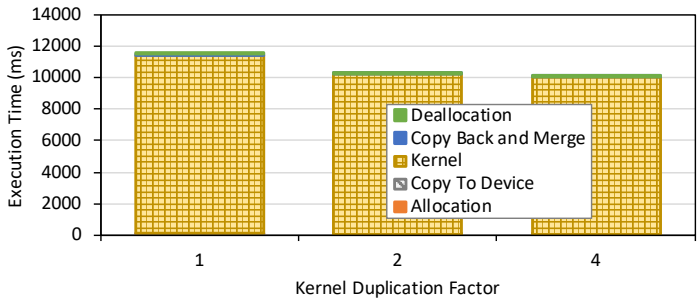


Figure 4.9: SSSP: Execution Time Breakdown for Different Duplication Factors

even slightly hurt its performance. Further profiling using the Intel FPGA OpenCL Profiler reveals that the main reason behind the performance effect of the duplication factor on CED is the saturation of the memory bandwidth. More processing elements on the FPGA exhaust the available bandwidth. Second, task partitioning tends to lead to less resource pressure and higher FPGA frequency for the same duplication factor. The reason is that, in task partitioning, only the sub-tasks that run on the FPGA need to be synthesized on the FPGA, while in data partitioning all sub-tasks need to be synthesized. As a result, the maximum duplication factor for task partitioning is higher than data partitioning.

Random Sample Consensus (RSC-D/RSC-T). Figure 4.13 shows the impact of the kernel duplication factor on the performance of RSC-D and

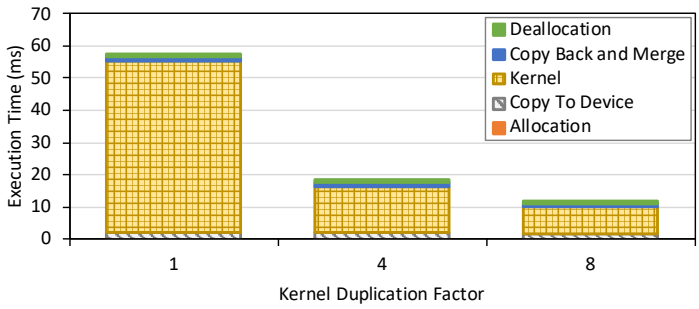


Figure 4.10: TQ: Execution Time Breakdown for Different Duplication Factors

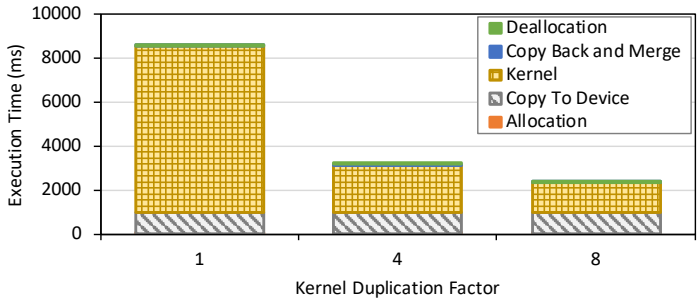


Figure 4.11: TQH: Execution Time Breakdown for Different Duplication Factors

RSC-T and FPGA utilization metrics. The bars, lines, and axes are set up in the same way as in Figure 4.12.

In data partitioning, we observe reasonable performance improvement with a duplication factor of 2, but there is little improvement beyond that. Unlike all the other cases where the bounding resources are the RAM blocks, the bounding resources for RSC-D are the DSP blocks. This is due to the fact that the first stage of RSC-D performs a large number of floating-point computations.

In the task partitioning strategy for RSC, because the first stage is offloaded to the CPU, the DSP block utilization drops significantly, enabling the kernel duplication factor to continue to increase, resulting in much better performance for task partitioning than for data partitioning. The performance improvement saturates around a kernel duplication factor value of 8, which results in a 1.6× speedup for task partitioning over data partitioning. Similar to CED, the profiler shows that the saturation of the memory bandwidth is the major reason why performance saturates at higher values of the

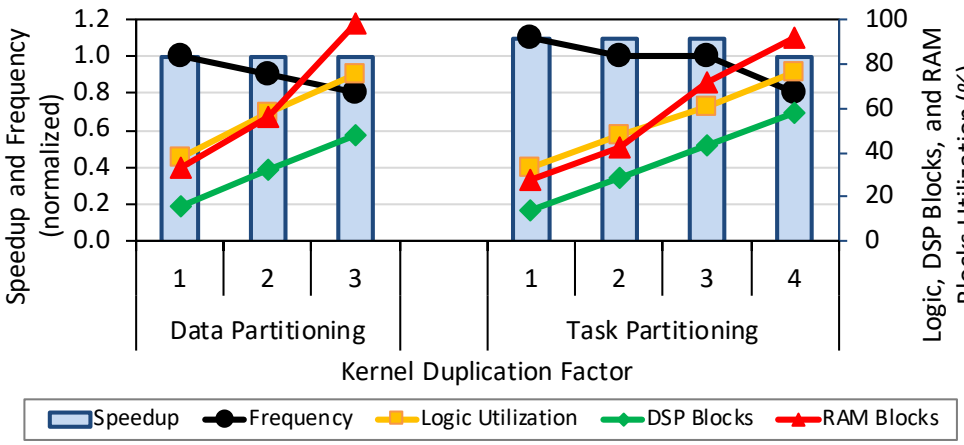


Figure 4.12: Canny Edge Detection: Speedup and Frequency (Normalized to Data Partitioning with Duplication Factor 1) and Resource Utilization for Different Duplication Factors.

kernel duplication factor.

4.6 Key Insights

Based on the performance evaluations we present in Sections 4.4 and 4.5, we extract five key insights for developers who choose to write collaborative programs for CPU-FPGA architectures. These insights cover generic collaborative computing techniques for heterogeneous systems, as well as CPU-FPGA specific collaboration schemes.

The first insight is that collaborative execution is actually beneficial. We observe that with both data and task partitioning strategies, collaborative execution effectively reduces the execution time of almost all benchmarks we examine.

Second, data partitioning requires careful choice of partitions to provide the highest performance. We observe that the different data partitioning benchmarks (i.e., BS, CED-D, HSTO, and RSC-D) prefer different data partitioning fractions α that result in the best performance (Section 4.4). This observation emphasizes the need for application-specific heuristics or offline tuning for finding the best static data partitioning, or the use of dynamic data partitioning. We show that dynamic data partitioning is effective in minimizing idle time in CED-D (Section 4.4.1). Unfortunately, CED-D is the only evaluated benchmark that supports dynamic data partitioning. Dy-

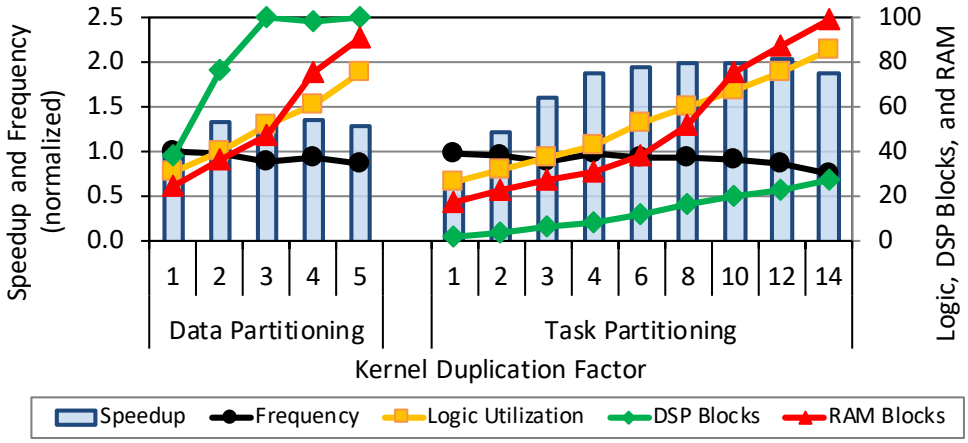


Figure 4.13: Random Sample Consensus: Speedup and Frequency (Normalized to Data Partitioning with Duplication Factor 1) and Resource Utilization for Different Duplication Factors.

namic data partitioning requires the use of shared memory variables to implement a task pool. CED-D uses two proxy CPU threads to control the execution on the CPU and the FPGA (i.e., launch CPU threads and FPGA OpenCL kernels for every task). The proxy threads can access the same task pool in the CPU memory and assign tasks to the CPU and the FPGA. In other data partitioning benchmarks, the CPU threads and the OpenCL kernel are launched only once at the beginning of the execution. Thus, in order to implement a shared task pool, CPU and FPGA workers would need to access the same shared variables, as CPU-GPU systems do [51]. However, shared virtual memory is not available in current CPU-FPGA systems. Future integration of shared coherent memory features and system-wide atomic instructions in FPGAs [71] will make dynamic partitioning more feasible.

Third, task partitioning generally enables more kernel duplication on the FPGA than data partitioning does, because task partitioning does not need to dedicate FPGA resources to all types of sub-tasks in an application, as it runs some computation stages entirely on the CPU. In RSC, there is a large difference between different sub-tasks. The first sub-task is sequential and much more floating-point intensive than the second sub-task. Thus, task partitioning saves FPGA resources that can be used for a higher kernel duplication factor than data partitioning. As a result, RSC-T outperforms RSC-D (Section 4.4.2). However, more kernel duplication does not always imply better performance. The potential benefit of kernel duplication is

benchmark-specific (Section 4.5). In CED, the different sub-tasks are very similar to each other in terms of computation and resource requirements. They compete for the memory bandwidth. Hence, the higher kernel duplication factor of CED-T than CED-D does not provide performance benefits (Figure 4.12). Even if kernel duplication is effective, there can be diminishing returns from increasing the kernel duplication factor too much, if the memory bandwidth saturates, as we show for RSC in Figure 4.13. In summary, developers must carefully use kernel duplication, in order to make effective use of the FPGA resources and thus improve application performance.

Fourth, data partitioning inflicts less burden on programmers and has less communication overhead than task partitioning. In task partitioning, tasks can only be partitioned into sub-tasks at specific points in the code. Finding the partitioning points might be painful for the programmer and makes it more difficult to evenly balance the workload across devices. Moreover, task partitioning tends to require more communication and synchronization points between devices, because both devices participate in all tasks. Emerging shared coherent memory features [71] (e.g., fine-grained memory coherence and system-wide atomic instructions) are expected to be beneficial in making such communication and synchronization easier.

Fifth, the current OpenCL stack for FPGAs provides a convenient programming model for application programmers, but there is still room for better programmability and higher performance if new features are provided inside the OpenCL stack. We believe that incorporating more OpenCL 2.0 features, such as fine-grained shared virtual memory [71] and system-wide atomic instructions [72], to the OpenCL stack for FPGAs will greatly benefit programmability and performance.

4.7 Related Work

To our knowledge, this is the first work to perform a thorough analysis of collaborative execution strategies on CPU-FPGA systems programmed with High Level Synthesis tools, like OpenCL. In this section, we first review recent works on collaborative execution on CPU-FPGA systems programmed with register-transfer level (RTL) code. Second, we review works on OpenCL programming for FPGAs. Third, we discuss recent efforts on collaborative

execution for integrated CPU-GPU architectures.

4.7.1 CPU-FPGA Coherent Memory

CPU-FPGA platforms with shared coherent memory have recently captured great attention from both academia and industry. Choi et al. [73] conduct a quantitative study of modern CPU-FPGA platforms, including QPI-based and PCIe-based ones. This study focuses mainly on micro-benchmarking for memory systems and evaluates acceleration of *entire kernels* on FPGAs. Our work focuses on evaluating *collaborative* execution strategies on CPU-FPGA platforms.

Enabled by the tighter integration of the CPU and the FPGA in CPU-FPGA systems, collaborative execution has been analyzed in various studies that accelerate applications. Weisz et al. [74] present a task-partitioning collaborative strategy to accelerate linked-list traversals. Chang et al. [75] accelerate seeding in DNA sequence alignment through a data-partitioning collaborative strategy. István et al. [76] adopt task-partitioning collaborative execution for regular expression operators for databases. Zhang et al. [77] present a task-partitioning collaborative algorithm to accelerate merge sort. Qiao et al. [78] accelerate the Deflate lossless compression algorithm on an FPGA. They apply a task-partitioning-based collaborative execution strategy for an entire compression service on a CPU-FPGA system which takes advantage of pipeline parallelism. Sidler et al. [79] accelerate pattern matching queries using a task-partitioning collaborative strategy. Schmit et al. [80] present a use case of a CPU-FPGA system, where the FPGA serves as a smart network transmitter/receiver and the CPU runs applications, using a task-partitioning collaborative execution strategy. These studies focus on accelerating specific applications by writing RTL code, while our work focuses on evaluating multiple collaborative patterns comparatively for each selected application using OpenCL HLS.

Several studies [81, 82, 83, 84] focus on integration of different types of *accelerators* in heterogeneous systems with general-purpose CPUs. Our work mainly focuses on collaborative execution strategies for CPU-FPGA platforms, but could be further extended to other accelerators in heterogeneous systems.

4.7.2 High-Level Synthesis with OpenCL

High-level synthesis (HLS) with OpenCL has been widely adopted to accelerate FPGA design due to its programmability. Ndu et al. [85] present and evaluate a benchmark suite, CHO, for OpenCL FPGA accelerators. Verma et al. [86] evaluate OpenCL HLS using OpenDwarfs benchmarks and identify optimization techniques for OpenCL HLS. Ramanathan et al. [87] propose a work-stealing technique using OpenCL atomics on FPGAs. Wang et al. [88] present a performance analysis framework to identify bottlenecks of OpenCL kernels synthesized on FPGAs. Multiple recent studies accelerate applications using OpenCL HLS, including particle identification [89], relational queries [90], convolutional neural networks [24], etc. Most of these studies focus on accelerating or evaluating entire kernels on FPGAs. Our work evaluates collaborative execution patterns with OpenCL HLS on CPU-FPGA platforms.

4.7.3 Integrated CPU-GPU Architectures

Collaborative execution on heterogeneous PCIe-based CPU-GPU systems with discrete GPUs has been studied from various aspects. Shen et al. [91] propose a workload partitioning scheme for heterogeneous CPU-GPU systems. This work proposes modeling, profiling, and prediction techniques to predict the best workload partitioning. Luk et al. [92] propose adaptive mapping to automatically map computation to CPU-GPU systems. The proposed techniques adapt to changes in input problem size and system configuration. These works mainly focus on discrete CPU-GPU systems without much discussion on how tight integration of the CPU and the GPU on a single chip further enables acceleration opportunities.

Collaborative execution strategies have been studied for integrated CPU-GPU systems using benchmark suites such as Hetero-mark [93, 94, 95], Chai [51, 96], and HeteroSync [97]. We leverage benchmarks from Chai [51] to evaluate collaborative execution strategies on CPU-FPGA systems. Sun et al. [98] evaluate the Radeon Open Compute Platform using collaborative benchmarks. Gómez-Luna et al. [99] present three use cases of collaborative execution on a CPU-GPU system with the Heterogeneous System Architecture (HSA) [100]. Che et al. [101] study data partitioning between CPUs

and GPUs specifically for betweenness centrality. Tang et al. [102] propose EMRF, a policy for balancing between fairness and efficiency in integrated CPU-GPU architectures. FinePar [103] and Cho et al. [104] automatically partition workloads to use both CPUs and GPUs in integrated CPU-GPU architectures. Airavat [105, 106] is a power management framework that improves the energy efficiency of collaborative CPU-GPU applications. HAShCache [107] adds a stacked DRAM as a shared last-level cache for integrated CPU-GPU processors to address the problem of disparity between the two devices in their demands on the memory system. Garcia-Flores et al. [108] evaluate integrated CPU-GPU systems with a shared last-level cache using collaborative benchmarks. Staged Memory Scheduling [109] is a multi-level QoS-aware memory scheduler for integrated CPU-GPU systems. Kayiran et al. [110] propose a concurrency management mechanism for integrated CPU-GPU systems to control the usage of memory and network by the CPU and the GPU. Garcia-Flores et al. [111] analyze the inefficiencies of demand paging in CPU-GPU systems when running collaborative workloads, and explore data sharing between the CPU and the GPU at finer granularity than a page (e.g., a cache line). Spandex [112] is a memory coherence interface specifically targeting integrated architectures. Vesely et al. [113, 114] enable system calls from GPUs which benefits for having shared virtual memory across CPUs and GPUs. These works represent the numerous research efforts on software and hardware approaches to collaborative execution on integrated CPU-GPU systems. Our work is the first step toward similar research lines for integrated CPU-FPGA systems with OpenCL.

4.8 Conclusion

In this chapter, we present strategies for collaborative execution on CPU-FPGA architectures and evaluate these strategies using existing collaborative OpenCL applications with high-level synthesis. To our knowledge, this is the first work to carry out a comprehensive analysis of collaborative execution on CPU-FPGA systems using the OpenCL programming framework. We show that collaborative execution outperforms the execution on conventional CPU-FPGA systems where no collaborative execution strategies are used. We describe the challenges that each collaborative execution strategy faces, pro-

viding insights for developers on how to use them. We find that (1) task partitioning enables more kernel duplication, a common optimization technique for FPGAs, than data partitioning, yet (2) data partitioning has lower communication overhead and achieves better load balance than task partitioning. We provide suggestions for emerging CPU-FPGA systems, where support for fine-grained shared coherent memory and system-wide atomic instructions would be beneficial. We believe and hope that our study will inspire FPGA developers to further explore collaborative execution on CPU-FPGA architectures to achieve the highest performance and efficiency. Our study could also be extended to other types of accelerators in heterogeneous systems.

CHAPTER 5

LANGUAGES AND COMPILERS FOR ACCELERATOR DESIGN AUTOMATION

The exploding complexity and computation efficiency requirements of applications are stimulating a strong demand for hardware acceleration with heterogeneous platforms such as FPGAs. However, a high-quality FPGA design is very hard to create and optimize as it requires FPGA expertise and a long design iteration time. In contrast, software applications are typically developed in a short development cycle, with high-level languages like Python, which is at a much higher level of abstraction than all existing hardware design flows. To close this gap between hardware design flows and software applications, and simplify FPGA programming, we create PyLog, a high-level, algorithm-centric Python-based programming and synthesis flow for FPGA. PyLog is powered by a set of compiler optimization passes and a type inference system to generate high-quality design. It abstracts away the implementation details, and allows designers to focus on algorithm specification. PyLog takes in Python functions, generates PyLog intermediate representation (PyLog IR), performs several optimization passes, including pragma insertion, design space exploration, and memory customization, etc., and creates the complete FPGA system design. PyLog also has a runtime that allows users to run the PyLog code directly on the target FPGA platform without any extra code development. The whole design flow is automated. The evaluation shows that PyLog significantly improves FPGA design productivity and generates highly efficient FPGA designs that outperform highly optimized CPU implementation and state-of-the-art FPGA implementation by 3.17× and 1.24× on average.

5.1 Overview

The last decade has witnessed an explosive growth of new applications in terms of quantity, diversity, and demands for computing capability and energy efficiency. As an example, deep learning algorithms, which have been shown to be successful in many domains, are driving the revolutionary changes in computer system design. According to Dean et al. [115], the number of machine learning papers on arXiv [116] doubles in less than two years, which has outpaced Moore's law. The rapid growth of diverse applications poses immense challenges to many aspects of computing systems, including compiler, architecture, storage, etc.

These challenges have motivated new computing systems for the new decade. The FPGA-based computing platform is an emerging platform that provides reconfigurability, along with high performance, low latency, and high energy efficiency. FPGA's unique computing capability makes it a promising platform to tackle the rising computation challenges. FPGA accelerators have been deployed in both cloud servers and edge devices at scale. However, as FPGAs are getting used in increasing number of emerging applications and scenarios at a rapid pace, programming FPGA and optimizing FPGA design gradually become the main barriers in FPGA development.

The most widely adopted FPGA development flow today starts with programming FPGA at the register transfer level (RTL) in hardware description languages (HDL) such as Verilog and VHDL. Then designers use FPGA synthesis tools from FPGA vendors to synthesize RTL designs into FPGA bitstreams, which are used to configure FPGA. Programming FPGA at this level requires rich expertise in digital circuit design and the FPGA architecture. Besides, programming at this level is non-intuitive, error-prone, and hard to reuse code compared with modern programming languages, leading to long development, optimization, and verification cycles.

High-level synthesis (HLS) aims to simplify FPGA programming. Elevating the abstraction level of FPGA programming to that of C/C++/OpenCL [1], [2], [3], [4], HLS tools enable FPGA designers to express their algorithms in more familiar high-level languages. Developers are expected to use HLS pragmas or directives to guide the HLS tools to optimize and generate desired RTL design. Compared with RTL design flow, HLS allows FPGA developers to develop, optimize, verify, and reuse their design at a higher level, thereby

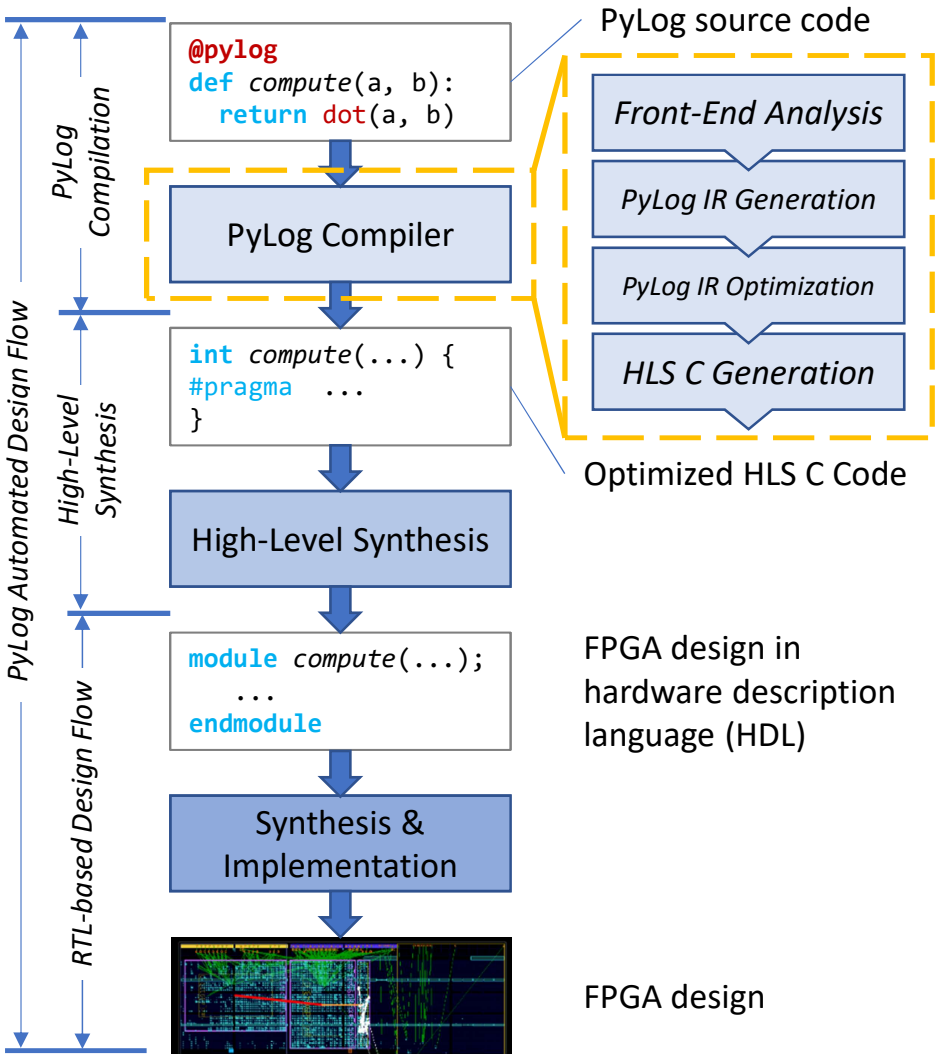


Figure 5.1: FPGA Design Flow with PyLog.

greatly improving productivity. However, as C/C++/OpenCL are initially designed for general-purpose processors and start with an inherent sequential execution model inside each kernel/function definition of these languages, they are essentially different from the FPGA's fine-grained parallel processing nature (the OpenCL model can describe parallel work-items but it is at thread-granularity and not very well supported in current HLS tools). The existing HLS tools are also designed in a way that accommodates the sequential execution model of input languages.

As a compromise between the HLS programming model and the FPGA execution model, HLS users often manually annotate their code with HLS pragmas or directives to give HLS compiler hints on parallelism and desirable synthesis approaches. These pragmas have a significant impact on the

performance and energy efficiency of the synthesis output. Oftentimes code transformation and rewriting are also needed to improve the outcome. In the end, the quality of synthesized design from HLS highly depends on how the code is written and how the HLS pragmas are added to the code. This requires a considerable amount of engineering time to iteratively adjust the source code and pragmas used. Complicated applications or thorough optimizations may lead to long source code that is difficult to read and maintain. For example, the HLS C code for convolution kernel from CHaiDNN [117] library has nearly 8,000 lines, which is much longer than the well-optimized convolution code for CPU or GPU.

Apart from the difficulties in creating and optimizing designs with current FPGA programming flow, the gap in the abstraction level between application programming and FPGA programming is another challenge in FPGA design. Applications are typically developed with languages at a much higher level of abstraction, where the programming models and styles focus more on describing the algorithm itself, instead of low-level implementation details. In the current HLS flow, when there is a need to accelerate the application, FPGA developers typically need to first lower the abstraction level of the application, re-implement the application in plain C code, and use it as the starting point for HLS. This lowering step is time-consuming and error-prone, and it also makes the FPGA design cycle longer.

These challenges in current FPGA design flows urge us to further elevate the abstraction level of FPGA programming. Among the existing programming languages, Python is one of the most popular and widely used languages. It has been well adopted in various domains such as machine learning, scientific computing, data analysis, education, etc. Python is also easy to learn. Compared to programming languages like C/C++ or Java, Python provides a concise syntax, a set of higher-level operators, as well as rich library support, which make programming in Python easier and more efficient.

We propose PyLog, an algorithm-centric Python-based programming and synthesis flow for FPGA. PyLog uses general Python compatible syntax, and it provides a set of handy low-level and high-level built-in operators that are capable of describing most of the common computation patterns in a natural way. The PyLog compiler takes Python code as input, and compiles the code into optimized HLS-synthesizable C code with HLS pragmas. Figure

5.1 shows the FPGA design flow with PyLog. We design PyLog in a way that the Python language allows the developers to focus on algorithm and computation flow description without much implementation details, while the PyLog compiler takes over the traditional FPGA developers' burden of exploring possible implementations and optimizations. The functional nature of the Python language also preserves some algorithm-level design information that is helpful for PyLog analysis and transformation. In the PyLog flow, algorithm and implementation are separated as much as possible. The goal of this design is to relieve FPGA programmers from manual design tuning while giving the PyLog compiler maximum information about computation patterns and therefore maximum freedom of design optimization. The whole PyLog flow is developed in Python. PyLog will be open-sourced to enable future research in this field.

The key contributions of this work are:

- We design and implement PyLog, a high-level Python-based programming and synthesis flow for FPGA, which greatly simplifies FPGA programming. The expressiveness of Python allows developers to achieve high design quality with much fewer lines of code compared with previous C/C++ based high-level synthesis flows.

- PyLog compiler is an ahead-of-time compiler and it is capable of doing various types of program analysis and optimizations. It features PyLog intermediate representation, type inference engines, and a set of compiler optimizations, which are all designed to create highly efficient FPGA systems.

- PyLog provides a set of high-level operators that ease algorithm description. These operators are general enough to describe computation patterns across different domains, and their implementation can be configured and optimized by PyLog to meet different design requirements.

- PyLog automatically generates optimized design from high-level algorithm specification, based on hardware resource constraints. Evaluation shows that PyLog generates highly efficient FPGA designs that outperform highly optimized CPU implementation and state-of-the-art FPGA implementation by 3.17× and 1.24× on average.

5.2 Related Works

Recently there are growing interests in the research community on high-level programming languages for FPGAs. HeteroCL [118] is one recent work that builds on the TVM framework [119]. HeteroCL is built as an API library of the Python Language, and its compiler is a runtime compiler. When HeteroCL code runs, it makes API calls to construct computation patterns and computing schedules from Python-syntax statements and generates synthesizable C code for Merlin HLS compiler [120]. For example, HeteroCL exposes APIs to perform code transformations such as loop pipelining, loop unrolling, quantization, etc. Although both HeteroCL and PyLog use Python syntax, their approaches are very different. HeteroCL is more like a Python library that is used to describe computation flow, instead of an implementation of a subset or extension of the Python language.

On the contrary, PyLog directly implements a subset of the Python language, PyLog code is compiled by ahead-of-time Python language compiler and the code that programmers write is what is the input to the compiler. PyLog's ahead-of-time compilation setting has several advantages. First of all, this enables maximum flexibility of the language, and makes it very easy to be extended. Second, PyLog is designed to be compatible with standard Python grammar, and Python programmers can immediately start to code in PyLog. PyLog supports a set of frequently used Python and NumPy operations and data containers. As long as a piece of Python code uses these supported operations, it can be synthesized into hardware. The Python language features that are synthesizable in PyLog is discussed in Section 5.3.7. Another difference between HeteroCL and PyLog is that, with HeteroCL, programmers still need to manually apply transformations and optimizations using HeteroCL APIs, while in PyLog, the compiler is responsible for implementation and optimization by default. Programmers can use pragmas to force the compiler to generate a specific implementation of the PyLog code, if they choose to do so.

Dahlia [121] is another high-level programming language that compiles to HLS C code. Dahlia uses Scala-like syntax and it uses a type system to enforce design constraints in the HLS code so that unrolling and memory partitioning factors match. Similar to HLS flow, Dahlia requires users to specify design pragmas. PyLog does not require manual annotation of HLS

pragmas. On the contrary, PyLog compiler automatically inserts pragma to optimize the design.

There are also several works that use Python and other high-level languages like Scala, and Haskell as hardware-description language (HDL) [122, 123, 124, 125]. The biggest difference between PyLog and these works is that PyLog flow is a high-level synthesis flow, and it does not require hardware knowledge to program in PyLog. These previous high-level HDLs elevate the syntax of the input languages, but users still need hardware expertise to use these HDLs.

5.3 PyLog Programming Model

5.3.1 Overview

PyLog presents a unified programming model for host and accelerator logic with consistent syntax and semantics. This seamless host-accelerator programming model enables agile system design, convenient functional simulation, and flexible design space exploration.

Listing 5.1 shows a high-level example of PyLog program that describes both host and accelerator. This example contains two functions, `preprocess` and `compute`. Function `compute` is decorated with a Python decorator `@pylog`, therefore it is a PyLog kernel function and will be synthesized into a hardware accelerator on FPGA by PyLog. With `@pylog` decorator, programmers can easily specify accelerator function in the existing Python code.

```
1  def preprocess(data):
2    ... # data pre-processing that runs on the host
3
4  @pylog
5  def compute(inputs): # top FPGA kernel function
6      def do_work(data): # user defined function
7          ...
8      for d in inputs:
9          do_work(d)
10   ...
11
12 inputs = preprocess(data) # data pre-processing
13 result = compute(inputs)  # call FPGA (or run synthesis)
```

Listing 5.1: A High-Level PyLog Example

As shown in the example, both host and accelerator are programmed with Python at the same abstraction level. The host and accelerator interact with

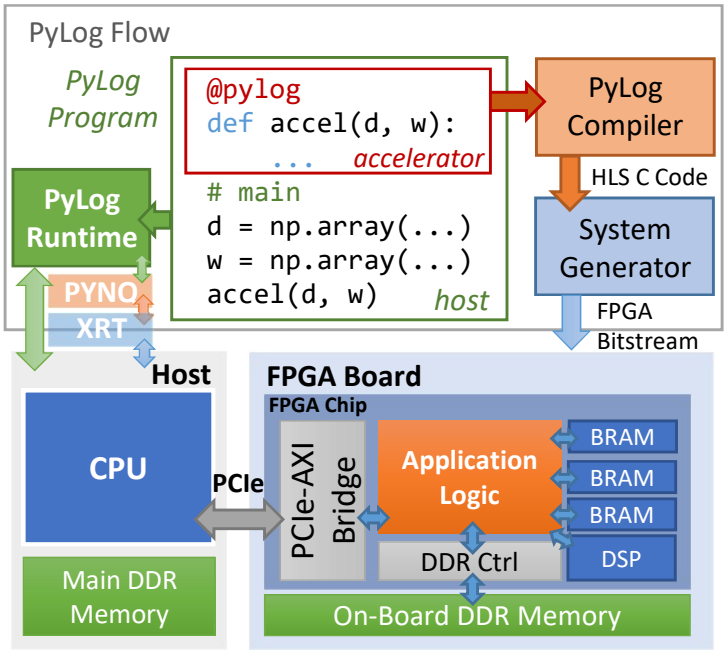


Figure 5.2: The PyLog Flow and Example System Architecture.

each other seamlessly in a natural way. PyLog closes the gap between the abstract level of host programming and FPGA accelerator programming, and enables efficient system-level host-accelerator co-design.

Figure 5.2 illustrates the overall PyLog flow and the target FPGA system architecture. When PyLog user runs PyLog program with a standard Python interpreter, the `@pylog` decorator calls PyLog compiler to compile the decorated PyLog kernel function into HLS C code. Then, the system generator syntesizes generated HLS C code and integrates all the system components to create a complete FPGA design. The generated FPGA bitstream is used to configure the resource and interconnects on FPGA. The rest of the PyLog program is interpreted by the standard Python interpreter, and this part is the host code that runs on the host CPU. When the decorated function is called, PyLog runtime is invoked to program FPGA, allocate and populate memory and invoke FPGA to accelerate the computation. The lower half of Figure 5.2 shows an example of PCIe-based FPGA platform. Note that PyLog can support both PCIe-based high-performance FPGAs and low-power SoCs and MPSoCs. CPU and FPGA interact through memory-mapped I/O ports as well as configuration registers on FPGA side.

PyLog requires the arguments of the PyLog functions to be NumPy arrays or NumPy scalars. From these NumPy object inputs, PyLog collects type

Table 5.1: PyLog's Built-in Modes

Mode	Description
`cgen`	Generates optimized high-level synthesis C code
`hwgen`	`cgen` then run FPGA synthesis
`deploy`	Program and call FPGA then collect results
`pysim`	Simulate PyLog code with Python interpreter

and shape information of the arguments to the top function, which is the initial information for the type inference engine in PyLog.

To run FPGA synthesis or run FPGA accelerator, users simply run the whole program with standard Python interpreter. When the decorated Python function `compute` is called (line 9), *PyLog compiler is invoked to look for the synthesized design for the decorated function.* If the function has not been synthesized before, PyLog will compile the decorated `compute` function, generate `compute` FPGA IP, integrate IPs into a complete FPGA design, and finally synthesize FPGA hardware design and get FPGA bitstream and configuration files. If there exists a synthesized design for the decorated function for the target FPGA platform but the design has not been deployed in the target FPGA, then PyLog will program the FPGA, allocate memory, populate memory space with input arguments, and call the FPGA accelerator and collect results. To run FPGA accelerated programs, users do not need to write any extra FPGA-specific code, and they will not notice any difference in the way of running the code on CPU or on FPGA. All the underlying CPU-FPGA interactions are taken care of by PyLog runtime. PyLog synthesis and execution share the same piece of code, which is also very similar to a regular Python code, except the decorator `@pylog`. Both synthesis flow and execution flow are fully automated.

PyLog users can also pass a "`mode`" string to `@pylog` decorator to configure PyLog mode. For example, `@pylog(mode='cgen')` runs only HLS C code generation flow. The list of possible PyLog modes is shown in Table 5.1.

PyLog uses Python syntax and it is friendly to both software and hardware developers. PyLog has a built-in type inference engine that can infer the types of objects in the program. PyLog supports basic Python operations and expressions as well as several high-level operators, which makes decription of computation flow intuitive, efficient, and natural (Section 5.3.2).

Besides, PyLog allows programmers to nest operators to express complicated computation patterns. Nested operators significantly simplify code and greatly increase the expressiveness of PyLog (Section 5.3.3). In addition to the expressive high-level PyLog operators, PyLog also supports data type customization and computation customization (Section 5.3.5). Besides, PyLog naturally allows users to simulate accelerator behavior in Python (Section 5.3.6). These capabilities make PyLog expressive and flexible enough to describe many different computation patterns across application domains. The rest of this section will describe all the features in detail.

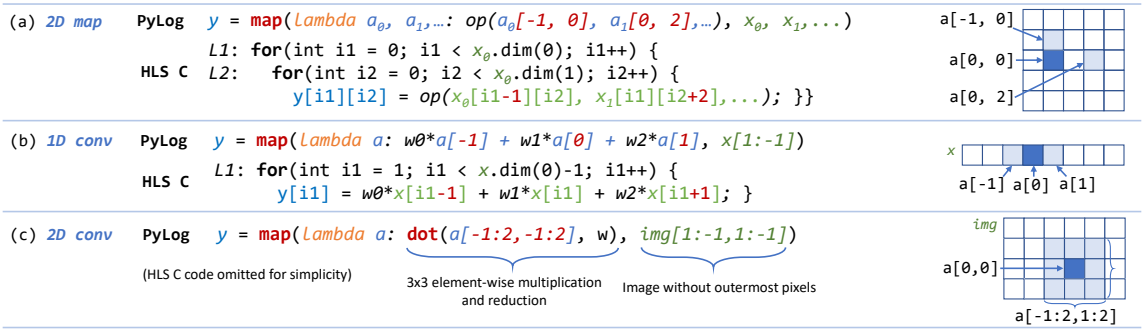


Figure 5.3: PyLog `map` Operator Examples. (a) 2D Stencil, (b) 1D Convolution, (c) 2D Convolution

5.3.2 High-Level Operators

In addition to the frequently used standard Python keywords and built-in functions, PyLog provides a set of high-level operators that describe common computation patterns. These high-level operators allow user to express computation at high-level without specifying implementation details. Each high-level operation can have multiple valid implementations. Figure 5.4 shows an example of generating multiple valid implementations from a PyLog `map` operation. PyLog can generate multiple versions of HLS C implementations in different styles, which corresponds to different hardware structures, e.g. shift registers, systolic arrays, etc.

Array operators. PyLog supports a set of commonly used NumPy-style binary array operators. Listing 5.2 shows two examples of array operators. The first example (line 2) assumes that `a` and `b` are multidimensional arrays with the same shape. PyLog can infer the types and shapes of operands (`a` and `b`) as well as target (`c`), and further infer the binary operations "`+`"

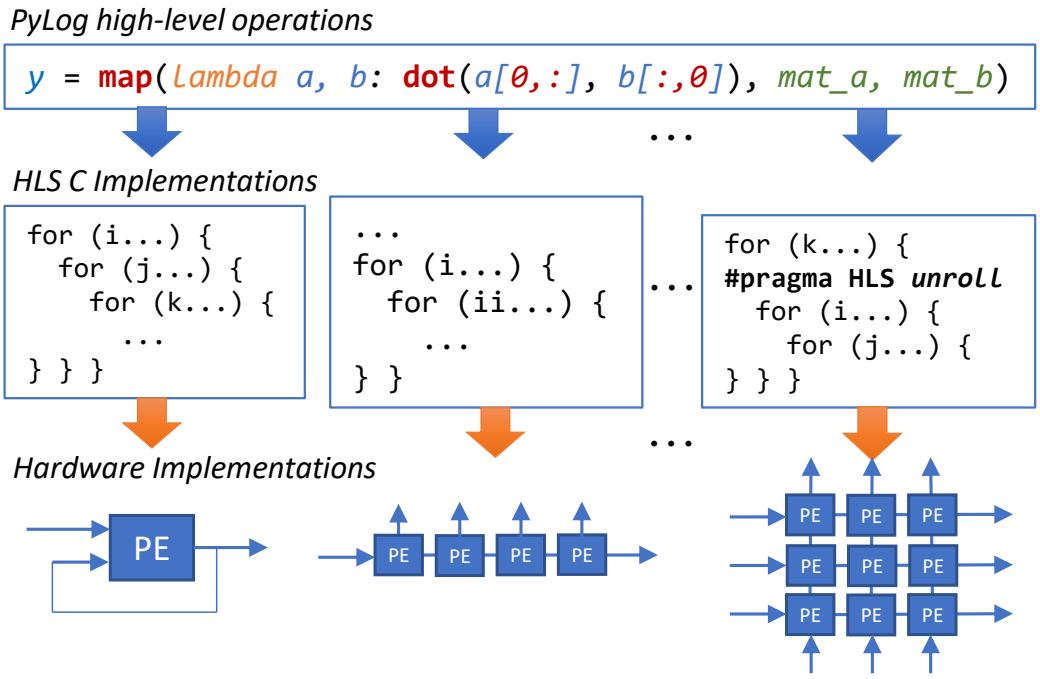


Figure 5.4: Different Implementations Generated from Same PyLog Code

and "=" (assignment) to be array operators. PyLog generates the optimized parallel for loops to implement this array operation. PyLog also supports indexing and slicing that adds flexibility to expressions as in NumPy and native Python.

PyLog slicing style is the same as that in Python. Slice expression "`start:end:step`" means a sequence of indices with "`start`" as the first index, "`start+step`" as the second index, etc. The index continues until it reaches `end` (but not including `end`). Note that this is consistent with Python slice's left-inclusive and right-exclusive definitions. Any of `start`, `end` or `step` can be left out. If `step` is left out, its value is understood to be 1. If `start` is left out, it means that the slice starts with 0. If `end` is missing, it means that the end value is the dimension. Using -1 for `end` means that the end is dimension-1. The PyLog compiler can infer the actual range of dimensions and indices according to the shape of the array under consideration. For example, `a[::2]` means a sub-array consisting of every other element in `a`.

```
1 # Array operation based on operators "+" and "="
2 c = a + b
3
4 # Array operation with slicing
5 z[2:66:2,:] = x * y[:,6,:-1]
```

Listing 5.2: Array Operation Examples.

Line 5 in the Listing 5.2 shows an example of using slicing and indexing to apply array operations to sub-arrays of the original arrays. In this example, elements in every other row from row 2 to row 64 of `z` (a total of 32 rows) receive the product of variable `x` and all elements in `y` that have index 6 in the second dimension and all the indices except the last one in the third dimension. The compatibility between the input and output arrays of this operation is checked by PyLog's type system. Also, the alignment between the input and output elements is automatically set by PyLog. For example, assume that `x` is a scalar variable. If `z` is a 64×8 array and `y` is a $32 \times 16 \times 9$ array, there will be $32 \times 8 = 256$ `z` elements and the same number of `y` elements involved. `z[2,0]` will receive `x * y[0,6,0]`, and `z[4,4]` will receive `x * y[1,6,4]`. In general, for all the 256 elements involved, `z[i,j]` will receive `x * y[m,6,n]` if $((i/2 - 1) \cdot 8) + j = m \cdot 8 + n)$.

Note also that `x` can be a scalar or a vector, which will make the meaning of `*` different (vector linear scaling and vector element-wise multiplication, respectively). Again, PyLog will be able to infer the types and shapes of all the arrays with indexing and slicing, and it will also check whether the operations are valid by comparing the shapes of sub-arrays. Array operators plus the slicing expressions support succinct and clear specification of linear algebra algorithms such as convolution and matrix multiplication.

```
1  # Vector add
2  out = map(lambda x, y: x + y, vec_a, vec_b)
3
4  # 1D convolution
5  out = map(lambda x:w0*x[-1]+w1*x[0]+w2*x[1], vec)
6
7  # Inner product
8  out_vec[i] = dot(matrix[i,:], in_vec)
9
10 # Square matrix multiplication
11 out = map(lambda x,y: dot(x[0,:],y[:,0]), ma, mb)
```

Listing 5.3: PyLog `map` and `dot` Examples.

The `map` operator. PyLog supports a built-in `map` operator, which is an extended version of the Python `map` function. Similar to the `map` function in Python, `map` operator in PyLog can be used as `map(f,`o_1`,...,`o_n`)` to repeatedly apply a function `f` to the n iterable objects o_1, o_2, ..., o_n where all objects must have the same shape. By default, the PyLog `map` operator behaves the same way as the Python `map` operator. In this case, f is defined with n formal parameters p_1, p_2, ... p_n, each of which refers to an element of the corresponding object that f is to be applied to in the map operator.

For example, in Line 2 of Listing 5.3, `x` (p_1) refers to an element of `vec_a` (o_1) and `y` (p_2) refers to an element `vec_b` (o_2). In the i^{th} iteration of the `map` implementation, `f` takes the i^{th} element of `vec_a` and the i^{th} element of `vec_b`, adds them together, and assigns to sum to the i^{th} element of `out`. Typically, function `f` is a `lambda` function (anonymous function), as shown in Listing 5.3. For example, in Line 2 of Listing 5.3, the `lambda` function is an addition function whose output is produced by adding the values of two formal parameters together.

Beyond the basic features, PyLog `map` further supports an extension to allow function `f` to access any number of elements in the iterable objects by specifying an offset or a offset slice expression that specifies a collection of offsets with each reference in the function body. The offsets are defined based on the iteration index. For example, in Line 5 of Listing 5.3, in the i^{th} iteration of `map`, the `lambda` function accesses `vec[i-1]` (specified as (`x[-1]`), `vec[i]` (`x[0]`), and `vec[i+1]` (`x[1]`) in the function body. Thus the `lambda` in this example is a 1D convolution function with a three-element filter of `w0`, `w1`, and `w2`. Figure 5.3 visualizes a few `map` examples. Note that this offset extension can naturally describe stencil operations. PyLog compiles stencil code described with the `map` operator and connects to SODA [126] to generate highly efficient stencil accelerators.

The `dot` operator. In PyLog, `dot` is defined as element-wise multiplication followed by a sum reduction. In other words, `dot(a, b)` is equivalent to `sum(a*b)`, where `a*b` is element-wise multiplication and the `sum` operator calculates the sum of all elements in the iterable object. For example, Line 8 of Listing 5.3 performs a dot product between row `i` of `matrix` and `in_vec` and assign the output value to the i^{th} element of `out_vect`. The `dot` operator is introduced in PyLog to simplify programming and expose more optimization opportunities to the compiler. This operation is frequently used in many different applications, e.g., image filtering, matrix multiplication, stencils, etc.

Custom operators. PyLog allows users to define custom operators as PyLog functions, inside a PyLog decorated top function. Similar to other operators, PyLog can infer the types and shapes of operands and output of custom functions by propagating type information from input to output through the whole function. These user-defined functions can be reused and simplify programming. These custom functions will be synthesized into HLS

C functions.

5.3.3 Offset-Slicing and Operator Chaining

The PyLog `map` offsets and offset slice expressions can be used for accessing higher dimensional formal parameter arrays. For each dimension, the offset can be a single number (e.g., 1, which means an input element in that dimension with offset 1 is accessed in an iteration), a slice (e.g., `-1:2`, which means three elements in that dimension with offsets -1, 0, and 1 are accessed in an iteration, or the entire dimension (`:`, which means all elements in that dimension are accessed in an iteration). For example, in Line 11 of Listing 5.3, the `map` operator will apply the `lambda` function to all positions of the 2D arrays involved. In iteration (`i`, `j`) of `map`, the offset expression `x[0,:]` means that the function accesses all elements of the $\texttt{i}^{\text{th}}$ row of `mat_a` and `y[:,0]` means that the function accesses all elements of the $\texttt{j}^{\text{th}}$ column of `mat_b`. That is, the map and lambda functions, chained together in Line 11, perform a dot product between the $\texttt{i}^{\text{th}}$ row of `mat_a` (`x[0,:]`) and the $\texttt{j}^{\text{th}}$ column of `mat_b` (`y[:, 0]`) and assigned the dot product value to `out[i,j]`, i.e., a matrix multiplication.

Note that the `map` offset slices for accessing formal parameter arrays should not be confused with the slicing of PyLog arrays. Assume that `x` is a formal parameter of a `lambda` and `vec` is a PyLog array, offset slice `x[-1:2]` specifies three accesses to the formal parameter with offsets -1, 0, 1 whereas `vec[1:-1]` produces a slice that consists of all elements of `vec` except the first and the last elements (i.e., all internal elements of `vec`).

The PyLog offset extension and operator chaining enables Python developers to intuitively and succinctly express common computation patterns in various application domains. For example, in Figure 5.3(c), `img` is the input to a 2D convolution, and `w` is a 3×3 convolution filter. The slicing `[1:-1,1:-1]` applied to `img` extracts out the sub-array excluding the outermost elements at the edges. The `lambda` function is applied to each element in the extracted array, `img[1:-1,1:-1]`. The parameter to the lambda function is `a`, which corresponds to each element in the array. The offset-slicing expression applied to `a`, `a[-1:2,-1:2]`, expresses a sub-array consisting of neighbor elements around the current element `a` as well as `a`.

The `dot` operator multiplies this square with convolution weight `w` (which is also a 3×3 square) element-wisely, and does a summation reduction to get the single convolution output at the current element `a`. This `dot` operation is repeatedly applied to each element position in `img[1:-1,1:-1]`, and this completes the whole 2D convolution. Note that PyLog can infer the shape of each object involved in the computation and users do not need to give any hints about object shapes. Details of PyLog type inference will be presented in Section 5.4.

```
1 # Dilated convolution where dilation = 2
2 map(lambda x:dot(x[-2:3:2,-2:3:2],w),img[2:-1,2:-1])
3
4 # Strided convolution where stride = 2
5 map(lambda x:dot(x[-1:2,-1:2],w),img[1:-1:2,1:-1:2])
```

Listing 5.4: Variants of 2D in PyLog.

In addition to basic 2D convolution, variants of more general convolution can also be expressed easily in PyLog, as shown in Listing 5.4. Dilated convolution with dilation equals to 2 can be described (line 2) by simply replacing `a[-1:2,-1:2]` with `a[-2:3:2,-2:3:2]` in the basic example above. Index "`-2:3:2`" means the sequence of indices of "`{-2, 0, 2}`", therefore `a[-2:3:2,-2:3:2]` represents a dilated convolution filter. Similarly, convolution with non-unit stride can also be described by specifying `step` in the index slices of `img` (line 5).

Note that these high-level operators only describe the arithmetic relationships between array objects without specifying any actual implementation details. For example, the `map` operator describes the repeated application of an operator to all the elements in an array, but it does not prescribe any ordering when iterating through these elements. In traditional HLS, the similar computation would be expressed with nested for loops, which actually implies an iteration order. The HLS quality heavily depends on how the computation is expressed, a key reason why it is hard to create optimal hardware design with HLS flow. In the PyLog flow, the actual order of iterating in the `map` operation is left for the PyLog compiler to decide. This approach not only simplifies programming, but also enables better design optimization and code generation. Given the high-level operators, PyLog compiler gets full information about the computation pattern. It knows exactly where data dependencies are in the code, which allows it to perform aggressive loop transformation and other code optimization. How PyLog performs code op-

Table 5.2: Examples of NumPy Operators in HLS C

Operator	**Tunable Performance Parameters**
`argmax`, `argmin`	p_i: number of parallel inputs of comparison tree
`max`, `min`	p_i: number of parallel inputs of comparison tree
`matmul`	p_o: number of output processed in parallel
`convolve`	p_o: number of output processed in parallel p_m: parallel multiplications for each output
`sort`	q_o: partition factor of the output buffer

timization will be discussed in Section 5.4.

5.3.4 HLS C Library Integration

In addition to the high-level operators and Python modules described above, PyLog also supports integration of external HLS C library functions. This allows users to leverage the existing highly optimized HLS libraries. We develop an extensible library of HLS C operators that implement widely used NumPy functions. The interface of these operators is compatible with NumPy functions. PyLog users can call these HLS C operators in the same way as NumPy function calls. A few examples are shown in Table 5.2.

In spite of the similar interface, the specific implementation of our operator library is very different from NumPy. These PyLog external operators are developed in HLS C language and highly optimized for hardware. These operators are implemented as HLS C code templates and are highly parameterized and configurable, and can be configured by PyLog according to data types and shapes, as well as design goals. Similar to other PyLog operators and function calls, the PyLog type inference engine will also do type inference and type checking for these external operators, to figure out the configurations of these operators and ensure the arguments and return of these operators to be valid. Type inference and checking is done based on the inference rules customized for each of these operators. Taking operator `matmul(A,B,C)` that performs matrix multiplication `C=A*B` as an example, the PyLog type engine checks if the shapes of `A`, `B`, and `C` has a pattern of (m, k), (k, n) and (m, n) or not. If yes, PyLog will configure the operator

based on the inferred type and shape and instantiate the operator template and generate HLS C implementation. Otherwise, PyLog will stop and output error messages accordingly.

Besides functionality parameters, these external operators also have configurable performance parameters, which configure the implementation of the operator. PyLog tunes these parameters to balance the performance and resource utilization of the entire design based on the design goals. The performance parameters are listed in the second column of Table 5.2. Each operator can also be configured as pipelined or non-pipelined according to design needs.

5.3.5 Bitwidth and Compute Customization

In FPGA designs, integers and fixed-point data types are widely used to improve computation efficiency. PyLog allows users to specify integer and fixed-point data types with arbitrary precision. Listing 5.5 shows a few examples. The PyLog type system supports the propagation and compatibility checking of user-defined data types.

```
1 a = pl_int8(0)     # 8-bit integer
2 b = pl_uint512(0) # 512-bit unsigned integer
3 c = pl_fixed(8,3)(0.0) # 8-bit fixed-point number,
4                        # 3 bits above decimal point
```

Listing 5.5: PyLog Arbitrary Precision Type Examples.

Aside from the internal optimization passes, PyLog also allows users who want to have more control to customize computation and memory in the code. Table 5.3 summarizes the computation customization types in PyLog. For loops can be customized with `unroll` or `pipeline`. Operator `map` can be customized with `reorder` or `tiling`, which will be applied to the for loops generated from `map` operation. Loop reordering and tiling are safe in `map` operation since there is no loop-carried dependence in `map` operation.

5.3.6 Functional Simulation Support

With the unified and seamless host-accelerator programming model provided by PyLog, programmers are not only able to program both host and accelerator efficiently, but also simulate the functionality of both host and accelerator easily. PyLog provides a `pysim` mode that allows the PyLog code to be

Table 5.3: Computation Customization in PyLog

Operator	Customization	Description
`for`	unroll	Unroll for loop
	pipeline	Pipeline for loop
`map`	reorder	Interchange for loops
	tiling	Tile for loops
generic	pragma	Insert HLS pragma

interpreted by the standard Python interpreter, and all the PyLog-specific operations and customizations will be removed or simulated. `pysim` can be used to simplify debugging and improve development efficiency.

5.3.7 Python Feature Support and Limitations

Inside the PyLog kernel (PyLog top function, to be synthesized), PyLog compiler supports commonly used Python features as well as PyLog-specific high-level operators. The host part in the PyLog code is interpreted by standard Python interpreter and it can contain any Python features and library calls.

Table 5.4 summmarizes the list of Python features that are supported in PyLog kernel. Please note that the Python features supported by PyLog compiler is a subset of complete Python features. This is because some Python features do not have well-defined and clear meaning for hardware synthesis, due to the language constraints in the backend HLS-C-based hardware design flow. For example, object-oriented programming, functions with variable length of input arguments, loops with variable bounds, dynamic array allocations, etc., are not synthesizable. For a piece of the existing Python function, as long as it uses the supported Python features, it can be synthesized into a hardware module by PyLog. If there are unsupported language features used in the PyLog kernel function, PyLog compiler will raise errors and warnings. We have some future plans in mind regarding Python feature support. First, we are working to support a wider set of Python features by defining their hardware behaviors. For example, dynamic array allocation, object-oriented programming, etc. There are several previous works that proposed the solutions to enable dynamic memory allocation in HLS [127], [128], [129], [130]. These works either provided libraries of dynamic data

Table 5.4: Supported Language Features

Category	**Operators**
PyLog high-level operators	`map`, `dot`, `user-defined ops`
NumPy operators	`argmax`, `argmin`, `max`, `min`, `matmul`, `convolve`, `sort`
Python features	`list`, `functions`, `calls`, `lambda`, `for`, `while`, `if...else...`, `slice`, `subscript`, `attribute`, `bin_op`, `unary_op`, `return`

Table 5.5: High-Level FPGA Design Flow Comparison

Features	Dahlia[121]	HeteroCL[118]	PyLog
Hardware customization	✓	✓	✓
Data type customization	✗	✓	✓
Ahead-of-Time compilation	✓	✗	✓
Host programming	✗	✗	✓
System generation	✗	✗	✓
HLS C library integration	✗	✗	✓

structures or dynamic memory allocation APIs for HLS users to use, or they provide a source-to-source compiler pass that translates code with dynamic memory allocation to static allocation code that can be synthesized by current HLS tool. Both approaches rely on some heuristic or learning-based models. Working with Python-based high-level operators, PyLog should be able to achieve similar dynamic allocation support at Python level. Second, PyLog compilation and synthesis are currently done at kernel level, therefore the unsupported operators will prevent the whole kernel from synthesis. Synthesis of individual operators is left as a part of future works. Operator-level synthesis can potentially enable JIT compilation for FPGA. We will discuss this future opportunity further in Section 5.6.2.

To summarize, Table 5.5 compares the features of existing high-level FPGA design languages and flows with PyLog.

5.4 Compilation and Synthesis Flow

PyLog flow is a fully automated FPGA programming and synthesis flow. It consists of three parts, PyLog compiler, PyLog system generator, and PyLog runtime.

PyLog compiler is a source-to-source compiler that translates PyLog source code to optimized HLS C code which can be synthesized by high-level synthesis (HLS) tools. The current supported HLS tool is Xilinx Vivado HLS [1] and Merlin compiler [120]. However, the analysis and optimization used in PyLog are not restricted to these HLS tools. The code generator of PyLog can be extended to support other HLS tools without much difficulty. The compilation steps of PyLog compiler can be categorized into four stages: (1) front-end analysis and PyLog intermediate representation (PLIR) generation, (2) type inference, (3) optimization, and (4) HLS C code generation.

PyLog system generator calls FPGA vendor's tools to synthesize generated HLS C code, integrates the FPGA application kernel with other system components, and generates FPGA configuration bitstream. PyLog runtime configures and invokes FPGA to accelerate computation in user's application.

5.4.1 Front-End Analysis and PLIR Generation

When a `@pylog` decorated function is called with NumPy arrays and scalars as parameters, PyLog compilation begins. In the first step, PyLog collects information about the data types and shapes of input arguments to the top function. This information is passed to PyLog sub-modules. The source code of the top function is parsed by Python built-in abstract syntax tree (AST) module `ast`, outputting the AST of PyLog code. AST is a low-level representation of the input code, which only represents the code syntax structure without semantics information. AST is the starting point for the following PyLog compilation steps. `ast` is the only module from standard Python interpreter that PyLog depends on. The following compilation steps do not depend on the existing Python interpreter.

In the first stage, the PyLog front-end traverses the PyLog AST, analyzes code structure, collects code information, and generates PyLog intermediate representation (PLIR). PLIR is a high-level tree-based data structure that describes the PyLog code structure, including high-level operations, low-level

Table 5.6: PLIR Node Categories

Category	**Example Node Types**
Low-Level Ops	`PLUnaryOp`, `PLBinOp`, `PLAssign`, etc.
PyLog High-Level Ops	`PLMap`, `PLDot`, `PLPragma`, etc.
Code Objects	`PLConst`, `PLVariable`, `PLArray`, etc.
Code Structures	`PLFuncDef`, `PLCall`, `PLFor`, etc.

controls, functions, external IP and configuration, etc. PLIR works as the hardware-agnostic IR for lowering high-level operations down to low-level controls, which is the implementation of high-level operations.

PLIR nodes include nodes representing high-level computation patterns and code structures, e.g., `PLMap`, `PLDot`, etc., as well as nodes representing lower-level generic statements and operations, e.g., `PLFor`, `PLBinOp`, etc. Compared with the nodes in the PyLog AST, each node in PLIR is coarser in granularity, and the attributes of PLIR nodes carry more information. Each PLIR node has multiple attributes that either point to the other PLIR nodes or store the related information about this node. Essentially, the PLIR generation can be considered as a process where the PyLog front-end analyzer aggregates the sub-trees and structure information in the AST to form PLIR nodes.

Table 5.6 lists the representative categories of PLIR nodes and examples. Low-level operations and expressions are the nodes that represent basic arithmetic operations, indices, basic expressions, etc. PLIR high-level operations are the nodes that represent high-level primitive operations in PyLog, e.g. `map`, `dot`, etc. These are PyLog-specific nodes. Code structures are the nodes that represent control flow and structure of the code, e.g., loops, branches, function definition, function calls, etc. The data fields of a PLIR node can point to another PLIR node; therefore, the whole PLIR is a tree that represents the code at high-level.

5.4.2 Type Inference and Type Checking

One of the biggest challenges in compiling Python code is that Python is a dynamically typed language and there is no explicit type declaration in the Python code, which makes it hard for the compiler to understand the actual

Table 5.7: Type Inference Rules Examples

Operations	Types
`out = UnaryOp(a)`	$\texttt{a} : T_t^n$, $\texttt{out} : T_t^n$
`out = BinOp(a, b)`	$\texttt{a} : T_{t_1}^n$, $\texttt{b} : T_{t_2}^n$, $\texttt{out} : T_{t_1}^n$
`out = dot(a, b)`	$\texttt{a} : T_{t_1}^n$, $\texttt{b} : T_{t_2}^n$, $\texttt{out} : T_{t_1}^0$
`out = map(f, a)`	$\texttt{a} : T_{t_1}^n$, $\texttt{f} : T_{t_1}^0 \rightarrow T_{t_2}^m$, $\texttt{out} : T_{t_2}^{m+n}$

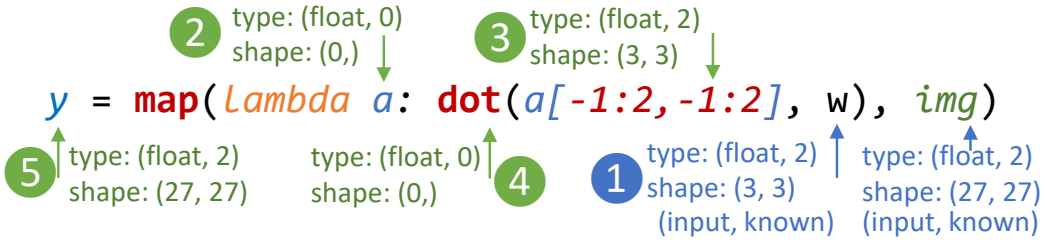


Figure 5.5: An Example of Type Inference on Chained High-Level Operators.

semantics of some operations. For example, consider a simple expression "`a + b`". Since `a` and `b` can be scalars or can be multidimensional arrays, this expression can mean scalar addition, or vector element-wise addition. The corresponding C code for this expression will be very different in these two cases. Without a context of types and shapes of `a` and `b`, it is not possible to know the actual meaning of this expression.

To solve this problem, we implement a type inference engine in PyLog that infers the types and shapes of each object in the PyLog code. With PyLog type inference support, PyLog users do not need to provide explicit type hints in PyLog code. The type checking engine will raise errors and warnings when it finds inconsistencies in the input and output types and shapes, according to the type checking rules. Since the backend of PyLog flow is C/C++-based HLS flows, which supports only a subset of C/C++ languages features and has several constraints in terms of code synthesizability, there are also constraints and assumptions on the PyLog kernel code. These constraints and assumptions simplify the type inference and type checking engine in the PyLog. Currently, we only support the inference and checking of the supported synthesizable operations in PyLog kernel code. Also, currently the type inference and checking in PyLog is done for data objects and expressions. Function types are not currently supported and are left as future work.

Algorithm 2 Type Inference and Checking

Input: Set of all variables S; set of input variables $S_{\text{in}} \subset S$; set of output variables $S_{\text{out}} \subset S$; Type mappings T defined on $S_{\text{in}} \cup S_{\text{out}}$, PLIR with root N_{root}.

Output: Type mappings T defined on S, or TypeError.

$S_{\text{typed}} \leftarrow S_{\text{in}} \cup S_{\text{out}}$
for each node $n \in \text{PostOrderTraversal}(N_{\text{root}})$ **do**
    if $n \in S_{\text{typed}}$ **then**
        $n_p \leftarrow \text{Parent}(n)$
        if $n_p \notin S_{\text{typed}}$ **then**
            $T(n_p) \leftarrow \text{TypeRule}(n \rightarrow n_p, T(n))$
            $S_{\text{typed}} \leftarrow S_{\text{typed}} \cup \{n_p\}$
        else if $T(n_p) \neq \text{TypeRule}(n \rightarrow n_p, T(n))$ **then**
            return TypeError
        end if
        for each $n_c \in \text{Children}(n)$ **do**
            if $n_c \notin S_{\text{typed}}$ **then**
                $T(n_c) \leftarrow \text{TypeRule}(n \rightarrow n_c, T(n))$
                $S_{\text{typed}} \leftarrow S_{\text{typed}} \cup \{n_c\}$
            else if $T(n_c) \neq \text{TypeRule}(n \rightarrow n_c, T(n))$ **then**
                return TypeError
            end if
        end for
    end if
end for
if $S \subset S_{\text{typed}}$ **then return** T
else return TypeError
end if

In the PyLog compiler, the type information of an object includes the type of data elements in the object as well as the number of dimensions of the object. PyLog compiler uses `PLType(ty, dim)` to denote types, where `ty` is the type of data elements and `dim` is the number of dimensions. For simplicity, we use notation T_t^d to represent `PLType(t, d)`. For example, the type of a three-dimensional array consisting of floating-point numbers can be represented as `PLType(float, 3)`, or T_{float}^3. Algorithm 2 shows the pseudo-code of the type inference algorithm in PyLog. Type inference starts with type and shape information of function arguments in the PyLog top function (which is carried by the input NumPy objects), and type information propagates across the whole PyLog code. Type inference is performed on PLIR and the resulting type and shape information is annotated to PLIR

nodes. Type information propagation happens by performing type inference at each PLIR nodes, which is done according to the type inference rules. Table 5.7 lists a few examples of type inference rules.

As an example, when inferring types of objects in a `map` operation, the type inference engine first retrieves the type of operands $T^n_{t_1}$ from current context, which stores the types and shapes of visible variables at current point in the code. Then, the type engine is able to tell that the type of the argument to function `f` is $T^0_{t_1}$, because `map` operator iterates through each element in the operand and `f` takes one element as input. Next, the type engine infers types of objects inside `f` and gets the return type of `f`, $T^m_{t_2}$. Finally, as a `map` operator, its return value is the aggregated results of `f`'s return values, so the type is $T^{m+n}_{t_2}$.

The shapes of objects are also inferred in the similar way, and happens at the same time as type inference. After type inference, the semantic of operations and expressions in the PyLog code is determined. When inconsistency is detected by the type system, a compilation error message is generated for the user. This makes debugging more user friendly than pure interpreter-based Python implementations. PLIR with type information is then ready for optimization and code generation.

5.4.3 Compiler Optimization

PLIR characterizes the high-level computation patterns in the input PyLog code, making it easier for the PyLog optimizer to optimize the computation flow. Before the optimization stage, all the compiler analysis and transformation are independent of actual implementation. In this optimization stage, the compiler starts to consider and optimize the implementation for better performance. PyLog optimization consists of three steps, high-level operators lowering, loop transformation, and HLS pragma insertion.

High-level operator lowering. In this first step, PyLog optimizer traverses through PLIR, and replaces high-level operators in PLIR with functional equivalent groups of generic low-level operators. For example, vector operators and `map` operators will be replaced with a group of for loop nodes that represent nested for loops. The information about the type of original high-level operators (e.g. map, dot, etc.) are kept and annotated to the

```
code: {
  loop_i: {},
  loop_map_1_tile_1: {
    loop_map_0: {
      loop_map_1_tile_0 : {}
    }
  },
  loop_dot_0: {
    loop_dot_1: {
    }
  }
}
```

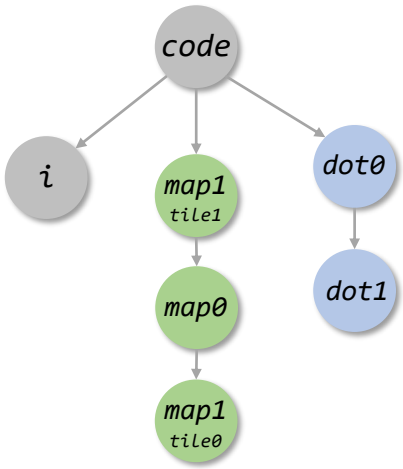


Figure 5.6: An Example of `for` Loop Structures and Its Tree Representation; The `for` Loops of Different Types are in Different Colors

generated for nodes. This information is useful in the following optimization.

Loop transformation. For each high-level operator, PyLog is capable of generating multiple styles of nested for loops, including plain sequential version, loop reordering version, loop tiling version, and mixed optimization version where loop reordering and tiling are combined. The type of loop transformation used depends on the operator type and the available hardware resources. For example, loop reordering and loop tiling are safe and possible in `map` operation since there is no loop-carried dependence in `map` operation. Note that each of these versions also has one or more tunable parameters.

HLS pragma insertion. After the PyLog optimizer replaces the high-level operators with nested for loops, the PyLog optimizer traverses the whole PLIR tree again and identifies the `for` loops in the code, then, it generates a loop structure tree that represents all the for loops in the code and their structure information. Each node in the tree is a `PLOptLoop` object that represents a for loop and its information. Each `PLOptLoop` object has actions of `unroll(n)` and `pipeline()`. Figure 5.6 shows an example of for loop structure and its tree representation. With this representation, the loop structure in the code becomes very clear. Then, PyLog optimizer starts to insert HLS pragmas according to the optimization algorithm. Algorithm 3 shows the pseudo-code for one of the optimization heuristics. In this algorithm, the algorithm traverses the loop structure in the post-order. That is, the optimizer starts with the leaf nodes in the loop structure, which corresponds to the in-

Algorithm 3 HLS Pragma Insertion Algorithm

Input: Original loop structure L, improvement threshold T, total available area A_{total}.

Output: Loop structure L with HLS pragmas configured.

```
1: latency, area ← EVALUATE(L)
2: for each loop L ∈ POSTORDERTRAVERSAL(L) do
3:     if L has attribute unroll or pipeline then
4:         continue
5:     else
6:         if L is from map then
7:             A ← {unroll, pipeline, unroll.pipeline}
8:         else A ← {pipeline}
9:         end if
10:        for each action ∈ A do
11:            L.action()
12:            latency', area' ←EVALUATE(L.action())
13:            if (latency − latency') / (area' − area) < T or area' > A_total then
14:                Undo L.action()
15:                continue
16:            else break
17:            end if
18:        end for
19:    end if
20: end for
```

nermost for loops in the code. Then it moves to the parents of the traversed nodes. For each node, determined by the type of for loop (whether it belongs to `map` nodes or `dot` nodes or regular for loops), the optimizer tries a set of candidate HLS pragmas (or actions). In each trial, it evaluates the area and latency changes after applying that pragma by running the area and latency estimators in PyLog. If the benefits are higher than a threshold, it accepts the change, otherwise it discards the change. Then it continues and moves on to next action or next node. Note that right now we are using a basic heuristic to guide the optimization. Other more sophisticated optimization/search algorithms can be used and plugged into the PyLog optimizer and guide the optimization. We leave this as a part of future work.

Performance and resource models. We use the performance and resource models proposed in [131] to estimate the latency and resource utilization of the design points of each source code version. Based on these estimates, the compiler identifies the optimal design points. These optimal

design points become the candidates of global design optimization. Note that here we are modeling the different implementation versions of high-level operations, which are much more structured and predictable compared to general loop nests. Besides, the current models are built based on C/C++ models, and there are other alternative ways of building these models. For example, since these high-level operators are regular and have well-defined computation patterns, we might be able to model the performance and resource of these operations at Python AST level, which will make the estimation even more efficient. We use an example in Section 5.5.3 to further demonstrate the optimization mechanism.

5.4.4 Global Design Optimization

The optimizations presented in Section 5.4.3 focus on fine-grained operation-level and loop-level optimization, while in this section, we discuss system-level optimization in PyLog flow. Operation-level and loop-level optimizations identify the top implementation candidates for each of the high-level operators in PyLog. At the system-level optimization stage, PyLog optimizer finalizes the low-level design choices based on the global constraints and optimization targets. We formulate this global optimization problem as an integer linear programming (ILP) problem, solve the problem with an ILP solver, and finally get the optimal design choices.

At the operation-level optimization stage, for each high-level operator p_i, the optimizer identifies the set of top m design candidates for this operator $\{p_i^{(1)}, p_i^{(2)}, \ldots, p_i^{(m)}\}$, as well as the latency and resource estimates of these candidates. We denote the latency and resource estimates with mappings $L : p_i^{(j)} \rightarrow \mathbb{Z}^+$ and $A : p_i^{(j)} \rightarrow \mathbb{Z}^+$ respectively. Note that for each type of FPGA resource, there will be one estimate function. To simplify the notations, here we only write down the formula for one type of resource. The same formula applies to the other types of FPGA resources. The goal of this global optimization stage is to identify the optimal choice of design candidates for each of the high-level operators so that the overall latency of the whole design can be minimized, while the resource usage meets the FPGA resource constraints. This optimization problem can be formulated as an ILP problem as follows.

We define a binary indicator variable $x_i^{(j)} \in \{0,1\}$ to denote whether or not we choose the j^{th} candidate of the i^{th} operator, i.e. $p_i^{(j)}$. Since only one candidate will be chosen for a specific operator, we have the constraint $\sum_j x_i^{(j)} = 1$ for each operator p_i. Given available resource $A_{\max}$ on FPGA, the optimization problem is:

$$\min_{x_i^{(j)} \in \{0,1\}} \sum_{i,j} x_i^{(j)} L(p_i^{(j)}) \tag{5.1}$$

$$\text{subject to} \sum_j x_i^{(j)} = 1, \forall i, \tag{5.2}$$

$$\sum_{i,j} x_i^{(j)} A(p_i^{(j)}) \leq A_{\max} \tag{5.3}$$

Please note that the summation here takes the control flows in the program into account. For example, if a high-level operator is called inside a sequential `for` loop, all the dynamic instances of this operator will be summed up. This formulated ILP problem is then solved by an external ILP solver, and the solution corresponds to the optimal choices of high-level operators design candidates.

5.4.5 C Code Generation and System Generation

After optimization, all the nodes in PLIR are low-level operators since the high-level operators have been replaced. The PyLog code generator traverses through the optimized PLIR and generates C AST, which is then translated into actual C code.

The PyLog system generator calls FPGA synthesis tools to synthesize generated HLS C code into FPGA IP block (Vivado HLS or Merlin compiler), and integrate the IP with all the other system components to create the complete FPGA design (Vitis or Merlin compiler). The whole system generation flow is fully automated.

5.4.6 PyLog Runtime

When the PyLog kernel function is called, PyLog automatically invokes FPGA to accelerate the program. First, it programs FPGA, then it allocates and populates arrays in CPU-FPGA shared memory space. Second,

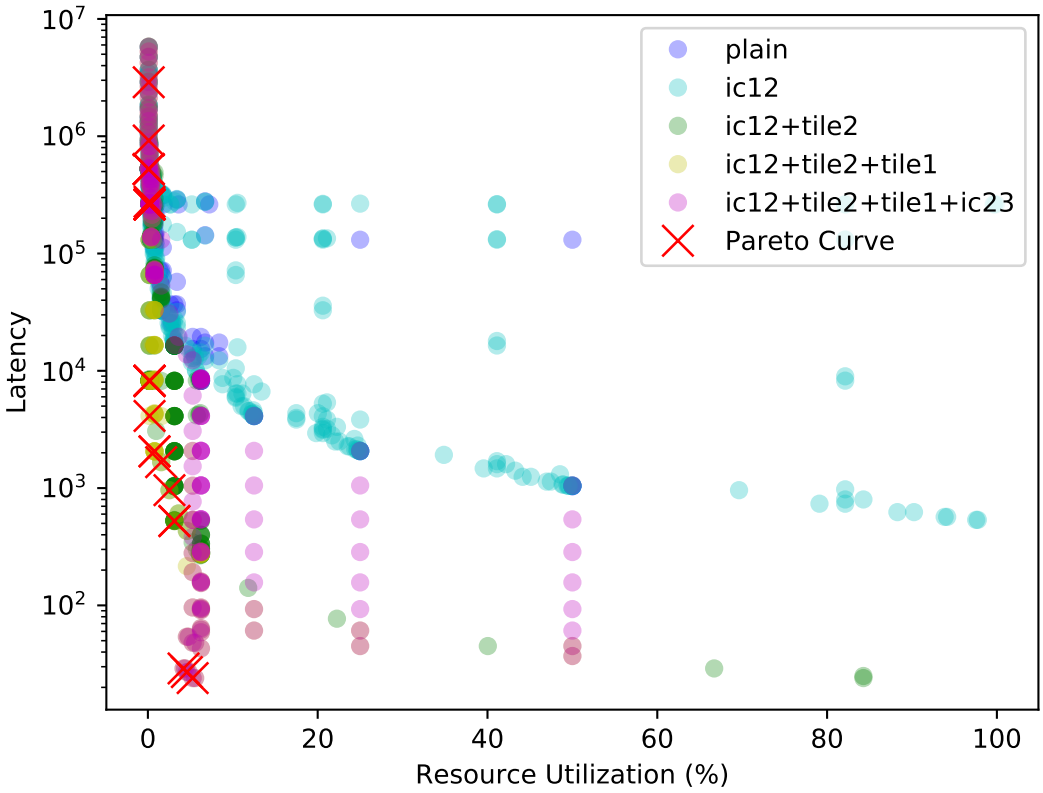


Figure 5.7: Design Space of Various Code Versions of 2D Array Addition ("ic12": interchange loop 1 and loop 2; "tile2": tile loop 2)

it invokes FPGA accelerator, and waits for FPGA to finish. Finally, PyLog collects computing results from FPGA and returns the results to the kernel function caller in the host program. This runtime is built on top of the Xilinx PYNQ library [132], which supports both low-power platforms and high-performance platforms.

5.5 Evaluation

This section evaluates PyLog flow in several different aspects, namely portability, language expressiveness, and performance.

5.5.1 Portability

PyLog flow is designed to be generic enough to be portable across different FPGA platforms. The whole PyLog flow, including code generation,

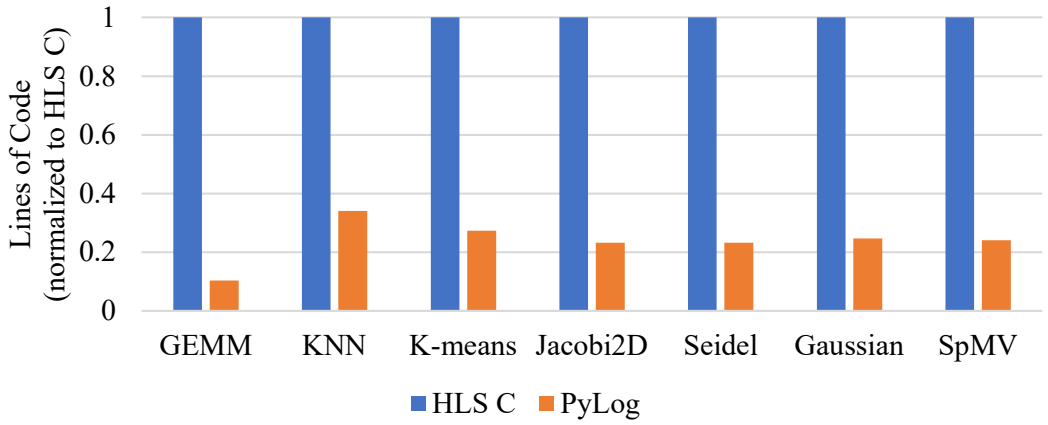


Figure 5.8: Length of HLS C Code and PyLog Code

hardware generation, and PyLog runtime, can be used with a wide range of FPGA platforms. Table 5.9 lists the FPGA platforms that are currently supported by PyLog flow. When targeting a specific FPGA platform, one simply passes the platform name to the `@pylog` decorator, and no other code change is needed. For example, `@pylog(mode='deploy', board='aws_f1')` will execute the PyLog code using Amazon AWS F1 instance FPGAs. Exactly the same PyLog code with `@pylog(mode='deploy', board='pynq')` applied instead will run the program with PYNQ FPGA, assuming FPGA bitstreams have been generated with `mode='hwgen'`.

5.5.2 Expressiveness

To evaluate the expressiveness of PyLog, we compare the number of lines of code between HLS C code and PyLog code. To make comparison fair, for PyLog versions, we only use PyLog built-in high-level operators to express the benchmarks but not using other pre-built functions or libraries. The HLS C versions are HLS C code generated by PyLog from the corresponding PyLog version. This guarantees the FPGA designs of two versions are equal. Figure 5.8 shows the results. With PyLog, on average only 30% length of code is needed to express computation, compared to the Vivado HLS flow.

5.5.3 Design Space Exploration and Search

We evaluate the effectiveness of our compiler optimizations by profiling the latency and resource utilization of real-world workloads. Figure 5.7 shows

all the valid design points of a 2D array addition example, as well as the optimal design points. These design points are identified by PyLog compiler automatically. First of all, PyLog identifies the four valid versions of implementation, that is, "ic12", "ic12+tile2", "ic12+tile2+tile1", and "ic12+tile2+tile1+ic23". Here "ic12" corresponds to the version that interchanges loop1 and loop2, while "tile2" is the version that tiles loop 2. Second, PyLog explores all the valid design points for each of these valid code versions. These design points correspond to different ways of inserting HLS pragmas. Figure 5.7 uses circles with different colors to mark the design points from different code versions. The optimal design points on the pareto curve are also marked in the figure. As we can see from Figure 5.7, the optimal design points come from different code versions. Our compiler is able to identify the optimal design points from all the code versions. These optimal design points become the design candidates for the global design optimization, which is then solved by the ILP solver.

5.5.4 Accelerator Performance

We evaluate PyLog performance with real-world benchmarks on Amazon EC2 F1 f1.2xlarge instance [138]. Amazon EC2 F1 f1.2xlarge instance is a cloud computing platform with 8-core Intel Xeon E5-2686 v4 CPU and a Xilinx Virtex UltraScale+ XCVU9P FPGA. The benchmarks are from different domains and have varied computation patterns, including linear algebra, data analytics, stencil, sparse operations, etc. Table 5.8 shows the evaluation results. The table lists FPGA resource utilization (look-up tables, registers, BRAMs and DSPs), design frequency (f (MHz)), design power (P (W)), and kernel execution time (T (ms)) of PyLog generated designs. Resource utilization, frequency, and power are collected from Vivado post-implementation reports. T_{CPU} is the execution time on AWS F1 CPU. The CPU baselines are optimized CPU implementations from [118] and other sources. We enable multi-threading whenever possible, as long as a NumPy operation has multi-thread implementation. T_{HCL} is the execution time on HeteroCL [118] generated accelerators. The HeteroCL accelerators are generated from optimized HeteroCL implementations that are available online. T_{PyLog} is the execution time on PyLog generated accelerators. SpMV and

histogram benchmarks do not have HeteroCL implementations available yet so we do not compare them with HeteroCL versions for these two benchmarks. The stencil benchmarks (Jacobi-2D, Seidel, and Gaussian Filter) are compiled to generate Vivado HLS C code with IPs from external HLS library SODA [126] as SODA generates IPs in Vivado HLS C code. Since HeteroCL uses the Merlin compiler as its general backend in the evaluation [118], we also compile the benchmarks to Merlin C code to allow further optimizations from Merlin so that we can fairly compare performance results from PyLog and HeteroCL. In terms of compilation time, PyLog HLS C code generation only takes seconds and therefore PyLog compilation time is negligible compared to Vitis synthesis that takes hours.

The last two columns in Table 5.8 show the speedup achieved by PyLog accelerators over CPU implementation and HeteroCL implementation respectively. On average, PyLog accelerators achieve around 3.17× and 1.24× speedup over CPU baseline and HeteroCL accelerators. PyLog can generate accelerators with better or similar performance compared with HeteroCL in most of the benchmarks. Note that we use PyLog generic backend to compile GEMM benchmark while HeteroCL uses special systolic array backend for GEMM. This is the main reason for the performance gap in the GEMM benchmark. After we add support for systolic array backends this gap will be filled. This is left as future work. The main sources of speedup achieved by PyLog are as follows. First, the high-level operators expose parallel processing opportunities and the compiler is able to insert HLS pragma with better insight. Second, PyLog compiles Python code directly and has native support for imperative programming. This enables users to have fine-grained control in hardware generation. Third, PyLog incorporates external highly optimized HLS libraries and it tunes the design parameters of these libraries to achieve good balance of performance and resource utilization.

5.6 PyLog Future Works

While we presented several features that PyLog currently supports, there remains several promising directions or features that we are exploring or will explore in the future. This section discusses the future features or directions that might be promising for the PyLog project.

5.6.1 Supporting Other Accelerators

Currently, PyLog generates FPGA designs from PyLog code. However, the PyLog frontend and intermediate representation are general enough to express computation patterns on other accelerators. For example, the high-level operators like NumPy operators can also be mapped to the parallel computation on GPUs or multi-core CPUs. Recently there are several Python-based libraries and frameworks that focus on parallel computing targeting GPUs or multi-core CPUs, e.g. Numba, CuPy, RAPIDS, Dask, etc. These frameworks either provide a set of APIs that connects to high-performance parallel code for target device, or compile Python code with optimized backend for the target platform.

In PyLog, the high-level operators aim to provide an abstraction of parallel and distributed computing patterns. For example, the `map` operator expresses the parallel applications of functions to individual elements in data container. This pattern is common in GPU computing and multi-core computing. Beside GPUs and multi-core CPUs, there are other customized accelerators that can potentially become the target devices for PyLog in the future. For example, the AI engine in the latest Xilinx Versal device is an 2D array of accelerators that accept instructions, and it is suitable at performing certain computation patterns like matrix multiplications and processing pipelines. Being able to compile PyLog high-level operators to these array of accelerators will simplify the programming of these customized accelerators.

5.6.2 Working with Existing Python Frameworks

Python has become very popular in many application domains, and many programming frameworks are built with Python. For example, several deep learning frameworks are very successful and with their help, development and deployment of deep learning algorithms to accelerators like GPU clusters are greatly simplified.

FPGAs are being deployed in more and more use scenarios, in both cloud computing and edge computing environments. The complexity in deployment environment motivates agile and efficient FPGA programming flows. With the Python interface and environment provided by PyLog flow, it becomes possible to plug PyLog synthesis flow into the existing Python-based

frameworks, like PyTorch and TensorFlow. The existing Python frameworks have several level of interfaces, which provide opportunities at different levels for PyLog. One possibility is that PyLog compiles users' Python kernel code and generates FPGA accelerators as well as Python/C++ glue logic that connect to the Python framework. Besides, the interaction between PyLog and surrounding Python environment might happen at a more fine-grained manner. For example, if PyLog can synthesize fine-grained operators instead of Python functions or kernels, and, with the lower-level hardware interface and driver supports from FPGA, we might be able to achieve JIT (Just-In-Time) compilation from Python code to FPGA accelerators, which will make application development for FPGAs much more flexible. Recently, researchers start to look into the JIT compilation for FPGA. [140] is a recent work that proposed a JIT compiler of Verilog. It reduces the time between compilation and running code to less than a second, and it enables printf-based debugging from hardware, which is an interesting step toward interactive debugging from FPGAs. In the future, if PyLog can support JIT compilation at the Python level, which is much higher level than Verilog level, it will greatly improve the hardware debugging efficiency.

5.6.3 Memory Access Optimizations

Memory access bandwidth and latency limitation remain one of the main challenges in creating high-performance FPGA designs. Although PyLog has some level of optimization on compute and data burst accesses, there is still significant room for improvement in memory access efficiency. There are several potential directions. First, a better model that captures memory latency and bandwidth constraint details will be beneficial for the compiler optimizer. In terms of memory bandwidth modeling and optimization, there are several recent FPGA-based DNN accelerator works that proposed memory bandwidth models and optimize for the optimal utilization of the memory access bandwidth. For example, Caffeine [141] proposed a hardware/software co-designed library that accelerates the entire CNNs on FPGAs. This work used a roofline model to predict and optimize the computing and memory access bandwidth of the generated FPGA designs. Besides, [142] is recent work that co-optimizes the DNN architecture and FPGA accelerator architecture.

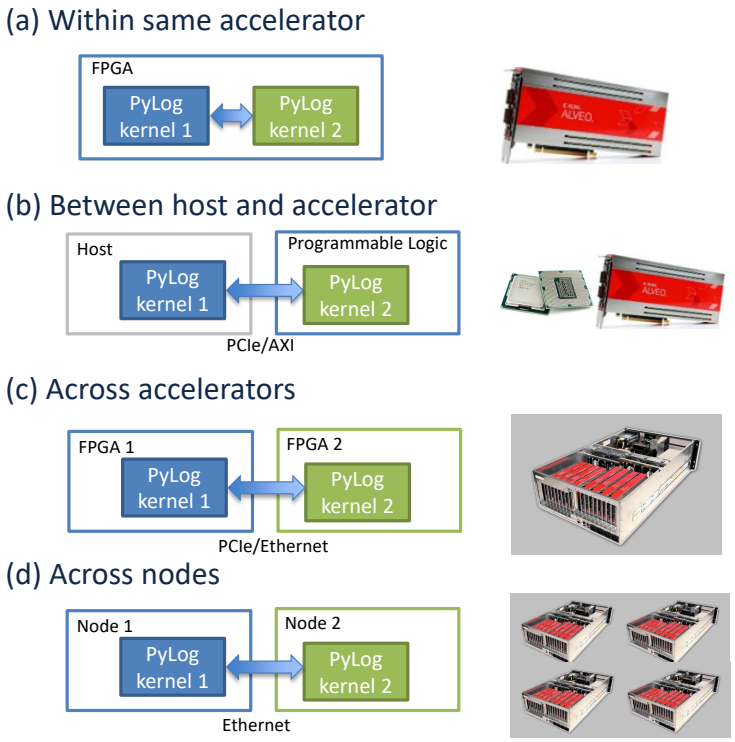


Figure 5.9: Multiple Kernels Mapped to Different Architectural Levels

In this work, a memory access bandwidth model is also proposed and used to predict hardware performance. Both of these two works, as well as most of the existing computing and memory bandwidth modeling works, work at C/C++ source code level. Since PyLog compiles and optimizes at Python level, there are several alternative ways to achieve this type of modeling and prediction. For example, using the input and output of high-level operations to predict memory access patterns, and, leveraging Python runtime to dynamically profile data access patterns, are both interesting options to explore. Second, depending on the data access patterns to the memory, PyLog may be able to choose the best buffer sizes and buffer layout, as well as the configuration of memory interfaces. Third, choosing the best memory or register type as well as data alignment and layout are also design considerations that may also become a part of future compiler optimizations.

5.6.4 Heterogeneous Computing Support

Besides the features presented in the sections above, PyLog also supports multiple kernels for synthesis. Currently, PyLog supports the mapping of

multiple kernels to logic in the same FPGA. However, the benefits of supporting multiple kernels are not limited to expressing multiple modules on the same FPGA. The major benefit of supporting multiple kernels is that it allows users to describe their system design in a more modular and flexible way. Also, it provides a high-level way of expressing interactions, parallelism, and pipelining between modules. The abstract kernels can be mapped and interpreted to different physical meanings when considered at different architectural levels. Figure 5.9 illustrates the physical implementations of multiple kernels when they are mapped to different architecture levels. This type of multiple kernel support at high abstraction level enables the possibilities to program heterogeneous computing systems with multiple accelerators from high level.

This multiple kernel support opens up many new opportunities for system-level optimization and automation. First of all, the mapping and scheduling of multiple kernels to physical devices are interesting problems to solve. Depending on the granularity of kernels and the data access patterns, the compiler may be able to help designers to choose the optimal mapping and scheduling of kernels. Second, the interfaces between multiple kernels and between architecture levels may have many choices, and depending on the application and data access patterns, compiler may be able to do some automatic tuning and choose the best configuration of the interface. Third, these are new dimensions of the overall design space exploration problem, and they may be optimized together with other dimensions using the unified compiler optimizer. This will help designers to identify the globally optimal design point.

5.7 Conclusion

In order to improve FPGA development efficiency and simplify FPGA programming, we built PyLog, an algorithm-centric Python-based FPGA programming and synthesis flow. PyLog flow compiles Python functions into optimized HLS C code, and generates a complete system including FPGA accelerator as well as host-side runtime environment. The built-in PyLog high-level operators and PyLog optimizer automate design implementation and optimization, which reduces the burden of FPGA developers. Evalu-

ation results show that PyLog is expressive to describe different types of applications with few lines of code and it can accelerates high-level software applications effectively and achieve significant speedup.

Table 5.8: Accelerator Performance Evaluation on AWS F1 Instance

Benchmark	**LUT**	**Registers**	**BRAM**	**DSP**	f (MHz)	P (W)	T_{CPU}	T_{HCL} [118]	T_{PyLog}	$\frac{T_{\text{CPU}}}{T_{\text{PyLog}}}$	$\frac{T_{\text{HCL}}}{T_{\text{PyLog}}}$
KNN	109276	74889	425	0	256.40	37.222	0.48	0.45	0.26	1.85	1.73
K-means	10829	17604	3	7	273.97	37.429	38.16	4.24	4.45	8.58	0.95
Jacobi-2D [133]	93925	111144	96	704	269.03	37.327	11.31	8.25	5.19	2.18	1.59
Seidel [133]	47304	57854	30	304	269.03	37.341	21.37	8.22	5.16	4.14	1.59
Gaussian Filter [133]	56580	75846	48	688	147.15	37.783	23.63	7.34	5.19	4.55	1.41
GEMM	12868	63759	655	1024	250.00	39.657	60.34	8.13	13.05	4.62	0.62
SpMV	8294	12787	25	21	273.97	37.225	0.29	-	0.24	1.21	-
Histogram [134]	4096	7647	13	0	273.97	37.327	5.85	-	2.07	2.83	-
									Geometric Mean	**3.17**	**1.24**

T_{CPU}: Execution time on CPU; T_{HCL}: Execution time on HeteroCL [118] generated accelerator; T_{PyLog}: Execution time on PyLog generated accelerator; All time values are in milliseconds (ms); '-' means the implementation is not publicly available.

Table 5.9: Current Supported FPGA Platforms in PyLog

Platform Type	**FPGA Platforms**
Low Power	ZedBoard [135], PYNQ [136], Ultra96 [137]
High Performance	Amazon EC2 F1 instance [138], Alveo series (U200, U250, U280) [139]

CHAPTER 6

DESIGN SPACE SEARCH AND OPTIMIZATION

In this chapter, we will use the problem of quantization for ReRAM-based DNN inference accelerator as an example to show a typical design space exploration (DSE) problem and one reinforcement-learning-based solution to this DSE problem.

ReRAM-based accelerators have shown great potential for accelerating DNN inference because ReRAM crossbars can perform analog matrix-vector multiplication operations with low latency and energy consumption. However, these crossbars require the use of ADCs which constitute a significant fraction of the cost of MVM operations. The overhead of ADCs can be mitigated via partial sum quantization. However, prior quantization flows for DNN inference accelerators do not consider partial sum quantization which is not highly relevant to traditional digital architectures. To address this issue, we propose a mixed precision quantization scheme for ReRAM-based DNN inference accelerators where weight quantization, input quantization, and partial sum quantization are jointly applied for each DNN layer. We also propose an automated quantization flow powered by deep reinforcement learning to search for the best quantization configuration in the large design space. Our evaluation shows that the proposed mixed precision quantization scheme and quantization flow reduce inference latency and energy consumption by up to 3.89× and 4.84×, respectively, while only losing 1.18% in DNN inference accuracy.

6.1 Introduction

Quantization is an important optimization for reducing DNN inference latency and energy consumption [143], [144], [145], [146], [147], [148], [149], [142], [150], [151]. Workflows that apply quantization to DNNs commonly

target the weights and inputs of the DNN layers. Reducing the number of bits used to represent weights and inputs saves memory space, reduces data movement overhead, and shortens the latency of arithmetic operations. When different quantization configurations are chosen for different layers, the quantization is considered to be *mixed precision*.

ReRAM-based accelerators have shown great potential for accelerating DNN inference because ReRAM crossbars can perform analog matrix-vector multiplication (MVM) operations with low latency and energy consumption [152, 153, 154, 155, 156]. However, ReRAM crossbars require ADCs to convert the partial sum computed by each crossbar from an analog to a digital value before it is combined with partial sums from other crossbars. These ADCs consume a large fraction of the total chip area and energy. Since the cost of ADCs scales exponentially with their precision, reducing the precision of ADCs is an important optimization. Hence, ReRAM-based accelerators provide the opportunity for a third important quantization target, the partial sums at the ADC output, which are not usually a concern for traditional digital architectures.

To take advantage of this opportunity, we propose an automated mixed precision quantization flow that jointly targets weights, inputs, and partial sums. Since the design space is large and prohibitive to search exhaustively, we use deep reinforcement learning (DRL) to search for the best configuration. Our evaluation shows that the proposed quantization flow reduces inference latency and energy consumption by up to 3.89× and 4.84×, respectively, while only losing 1.18% in accuracy.

We make the following contributions:

- A quantization scheme for ReRAM-based DNN inference accelerators that jointly targets weights, inputs, and partial sums, with a functional simulator that models the quantization scheme
- An automated mixed precision quantization flow powered by deep reinforcement learning that searches for the best quantization configuration for DNN inference on ReRAM-based accelerators
- An evaluation of the joint impact of weight, input, and partial sum quantization on the energy and latency of ReRAM-based DNN inference accelerators

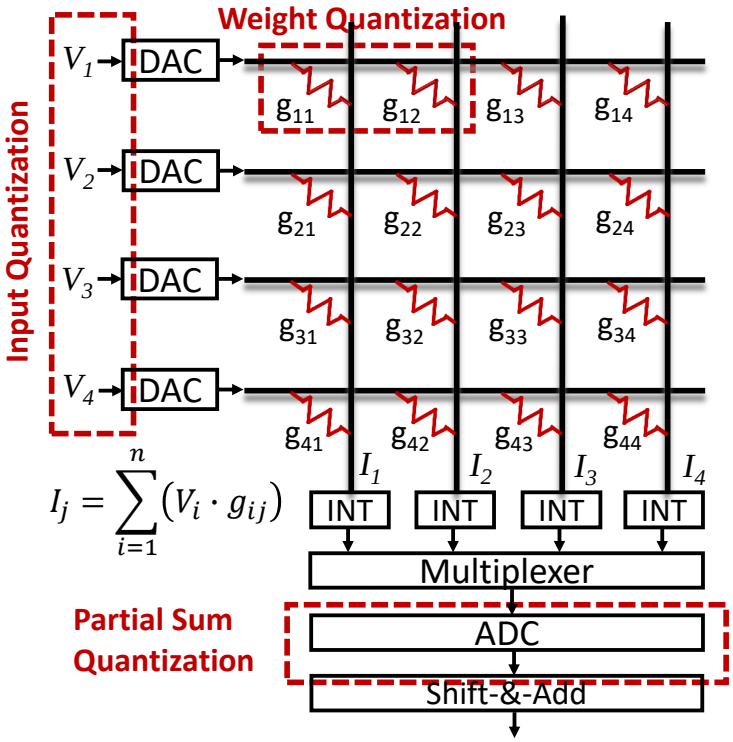


Figure 6.1: ReRAM Crossbar Architecture and Quantization

6.2 Quantization Scheme

6.2.1 Background

ReRAM crossbars are circuits capable of performing MVM operations with low latency and energy consumption by leveraging analog computing. Figure 6.1 shows a high-level diagram of a typical crossbar architecture. The weights of a matrix are stored in the resistive memory cells of the crossbar. In Figure 6.1, g_{ij} is the conductance of a memory cell. The input vector is applied as a voltage at the rows of the crossbar (V_i). The output vector is read as the current at the columns of the crossbar (I_j). The current is then converted to a digital value using an ADC.

Since practical crossbars are only capable of performing low precision MVM operations, higher precision MVM operations are realized by bit-slicing the weights and inputs. Weight slices are distributed across multiple crossbars, and the partial sums of each crossbar are then shifted and added together. Input slices are streamed sequentially into each crossbar, and the partial sums of each input slice are also shifted and added together. If the weight matrix dimensions are larger than the crossbar dimensions, the matrix is

divided into tiles and the partial sums of each tile are then added to produce the final MVM result.

6.2.2 Weight and Input Quantization

Recent research has revealed that weights (synapses) and inputs (activations) in a DNN typically do not need full precision to guarantee the DNN prediction accuracy [145, 147]. In general, using a low bit-width format to represent weights and inputs saves memory space, reduces data movement overhead, and shortens the computation latency. Weight and input quantization have been thoroughly studied for traditional architectures. In this work, we propose weight and input quantization schemes for ReRAM crossbars.

Weight quantization in the crossbar architecture can be achieved by either changing the number of crossbars or the number of bits per crossbar cell. However, the impact of device-circuit non-idealities in the crossbar (both linear and non-linear) increases with increasing bits per crossbar cell leading to significant losses in network accuracy [157]. Hence, this work assumes a fixed two bits of weights are stored in each crossbar cell [154] and implements weight quantization by varying the number of crossbars used to store the weights.

Input quantization in the crossbar architecture can be obtained by either changing the number of input slices streamed or the number of bits per slice. However, increasing the bits per input slice requires increasing the ADC precision which increases the overhead of the ADC non-linearly. Hence, this work assumes a fixed one bit per input slice [154] and implements input quantization by varying the number of input slices streamed.

6.2.3 Partial Sum Quantization

While weight and input quantization have received significant attention by quantization flows because of their relevance to digital architectures, partial sum quantization has not been thoroughly studied. Partial sum quantization in the crossbar architecture can be achieved by reducing the precision of ADCs at crossbar outputs. We model the impact of partial sum quantization combined with weight and input quantization by implementing a functional

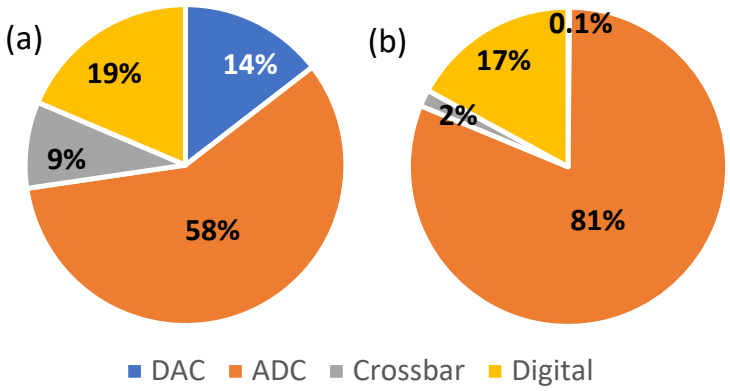


Figure 6.2: Distribution of MVM Cost: (a) Energy Distribution (b) Area Distribution

simulator described in Section 6.4.2. We show that partial sum quantization can substantially reduce the energy consumption and latency of DNN inference accelerators.

Energy. Figure 6.2(a) shows the distribution of energy consumption across four components (DAC, ADC, crossbar, and digital peripherals) for a 16-bit MVM operation [154]. It is clear that ADCs are the dominating component of the total energy consumption. Consequently, quantizing partial sums by operating ADCs at lower resolutions can yield significant energy reductions in the MVM operation. Table 6.1 shows how reducing ADC resolution translates to reductions on the total inference energy consumption. These results assume that ADC power decreases linearly with the ADC resolution, which is a conservative assumption.

Latency. The latency of MVM operations is limited by ADCs, but to understand why, it is important to first look at ADC area. Figure 6.2(b) shows that ADCs consume a substantial amount of the area in the crossbar architecture. For this reason, ADCs are time-multiplexed such that one ADC is reused across all the columns of the crossbar [154]. The typical size of a crossbar is 128×128. Consequently, even with an ADC working at very high sampling frequency such as 1 GHz, a crossbar read operation requires 128 ns, while typical crossbar reads without ADC require 5-20 ns [158]. Area consumption and sampling time for an ADC decrease with reducing ADC resolution. Consequently, partial sum quantization can achieve significant reductions in MVM latency.

Table 6.1: Energy Savings of Reduced ADC Resolution

ADC Res.	LSTM (24 tiles)		MLP (9 tiles)	
	Energy (μJ)	Diff.	Energy (μJ)	Diff.
8 (baseline)	59.0	-	16.6	-
4	46.3	-21.4%	13.2	-20.6%
2	41.0	-30.5%	11.6	-30.3%
1	38.3	-35.0%	10.8	-35.1%

6.3 Mixed Precison Quantization Flow

We model the mixed precision quantization problem as a reinforcement learning (RL) problem. In an RL problem, an *agent* (search engine) interacts with the *environment* and learns the best *policy* to take *actions* in certain states of the environment. The environment in an RL problem can be modeled with a Markov Decision Process (MDP) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{P}$ is the transition function that describle the dynamics of the MDP system, $\mathcal{R}$ is the reward function that maps state and action pair to a real-valued reward number, and γ is the discount factor that describes the degradation of future rewards.

6.3.1 MDP Modeling

In this quantization problem, we define the MDP as follows. State space $\mathcal{S}$ is defined as all possible configurations of the DNN. We can see this is a huge state space. $O(|\mathcal{S}|)$ is exponential to the number of layers in the DNN. Action space $\mathcal{A}$ is defined as all possible quantization configurations for a layer in the DNN. Transition function $\mathcal{P}$ is defined in a way such that we quantize the DNN layer by layer from the first layer. After an action is applied on the current layer, the environment moves to the next layer. Reward $\mathcal{R}$ is defined so that accuracy, power, and latency of the quantized DNN are all captured. Note that reward is a function of both current state and current action. We will discuss more about the reward definition in the following paragraphs. Discount factor γ is set to 1. This is based on the finite-horizon problem setting, reward setting, and the fact that we only care about the final accuracy of the fully quantized DNN. In the RL setting, a policy π is a

Table 6.2: Tunable Parameters

Parameters	**Values**
Weight bits	4, 8, 16, 32
Weight bits (fractional)	1, 2, ..., Weight bits - 1
Input bits	4, 8, 16, 32
Input bits (fractional)	1, 2, ..., Input bits - 1
ADC precision	1,2,3,4,5,6,7,8

function that maps state space to action space. In other words, a policy tells the agent the action a to take given the current state s: $a = \pi(s)$.

6.3.2 Action Definition

The actions are defined as quantization configurations for a layer in the DNN, which includes weight quantization, input quantization, and ADC precision. All the tunable parameters are listed in Table 6.2. There are two parameters to describe quantizaiton of each of weights and inputs, one is the total bit width, the other is the bit width for the fractional part. The possible values of total bits are powers of 2. This is a constraint that comes from bit slicing in the functional simulator. The bit width of the fractional parts can be any number less than the total bit width. ADC precision can be any integer between 1 and 8. Each action represents a configuration of these parameters for one DNN layer.

6.3.3 Reward Assignment

We define reward $R(s, a)$ at state s, action a as a function of accuracy cost and hardware cost (energy and latency) of the quantized DNN when running on ReRAM accelerators.

To capture the inference error of quantized DNNs, instead of using prediction accuracy of quantized DNN as in previous works [145, 147, 148], we use the following definition of the error of quantized model:

$$Cost_{\text{accuracy}} = Loss_{\text{quantization}} - Loss_{\text{original}} \tag{6.1}$$

where $loss_{\text{quantization}}$ and $loss_{\text{original}}$ are the cross-entropy losses of the quantized DNN model and the original model respectively. The intuition is to use the original model as the reference and any differences in the output are counted as errors introduced by quantization. The major reason of using loss instead of accuracy is to reduce the number of inferences during the evaluation of a quantization scheme and speed up the search progress. To model the hardware cost (energy and latency) of quantization schemes, we use the fractions of bitwidth over original bitwidth in each layer, weighted by the number of weights and inputs in DNN layers as well as ADC bitwidth. That is, the hardware cost of a quantization scheme can be approximated as

$$Cost_{\text{hardware}} = \sum_i \alpha^{B^i_{\text{ADC}}} \left(f^i_{\text{input}} \frac{B^i_{\text{input}}}{B_{\text{full}}} + f^i_{\text{weight}} \frac{B^i_{\text{weight}}}{B_{\text{full}}} \right) \tag{6.2}$$

where B^i_{ADC}, B^i_{input}, and B^i_{weight} are the ADC bitwidth, input bitwidth, and weight bitwidth of the i^{th} layer respectively. B_{full} is the full bitwidth without quantization for inputs and weights. f^i_{input} and f^i_{weight} are the fractions of number of inputs and number of weights over total inputs and total weights respectively.

Reward is calculated based on both costs as follows:

$$Reward = -T(Cost_{\text{accuracy}}) - Cost_{\text{hardware}} \tag{6.3}$$

where T is a threshold function: $T_t(x) = \infty \cdot \mathbb{1}_{x>t} + x$. Here $\mathbb{1}_{x>t}$ is the indicator function which equals to 1 when $x > t$ and 0 otherwise; and t is the threshold.

6.3.4 Learning Algorithm

With the definition of MDP, our goal is to find an optimal policy that gives us the largest expected reward for the starting state (the whole DNN for quantization). In order to find out the optimal policy, we need some estimation on the potential of each state and each action, with current policy π, i.e., the expected value of state s or state-action pair (s, a). The expected value of state is typically called state-value funciton $V^\pi(s)$, while the expected value of state-action pair is called Q-value function $Q^\pi(s, a)$.

One thing to notice is that both state space and action space are very large, and it is almost impossible to search for optimal policy with brute-force. We parameterize the policy as π_θ where θ is the parameter. With the parameterized policy, we are able to apply policy gradient algorithms to the problem, which is a family of RL algorithms widely used in practice.

The basic policy gradient theorem states that the gradient of the value function can be expressed as

$$\nabla v^\pi = \frac{1}{1-\gamma}\mathbb{E}_{s\sim\eta^\pi, a\sim\pi(s)}\left[Q^\pi(s,a)\nabla \log \pi(a|s)\right]] \tag{6.4}$$

where $Q^\pi(s,a)$ is the expected value of state s and action a, which can be estimated with value-based methods in RL. Value-based methods learn the value function, rather than the optimal policy itself. η^π is the estimated normalized state occupancy under policy π. This formula essentially states the weighted average of values over potential trajectories of the agent, and the weights are the function of policy gradient. The policy gradient algorithms work as follows. First of all, we start with some policy with random parameters θ. We use value-based algorithms to evaluate the expected values of state-action pairs (s,a). Then we calculate the gradient of θ using the formula, and we can update θ and get a new policy π'. With the new policy, we can again evaluate the expected value of each state-action pair (s,a) again. This completes one iteration of the algorithm. We can see the iteration consists of two parts: (1) "Actor": update θ with gradients of θ, and propose a new policy π'; (2) "Critic": evaluate the current policy π with value-based methods. Therefore, this type of combination of policy gradient and value-based method is often called "actor-critic" methods. In this work, the actor and critic are both implemented with fully connected DNNs.

6.3.5 Putting It All Together

With definition of all the details of MDP, we can put together everything to form a complete optimization flow. The complete flow is illustrated in Figure 6.3.

The optimization flow is a combination of reinforcement learining components and quantized model evaluation components.

RL components. The RL components consist of three parts: *actor*,

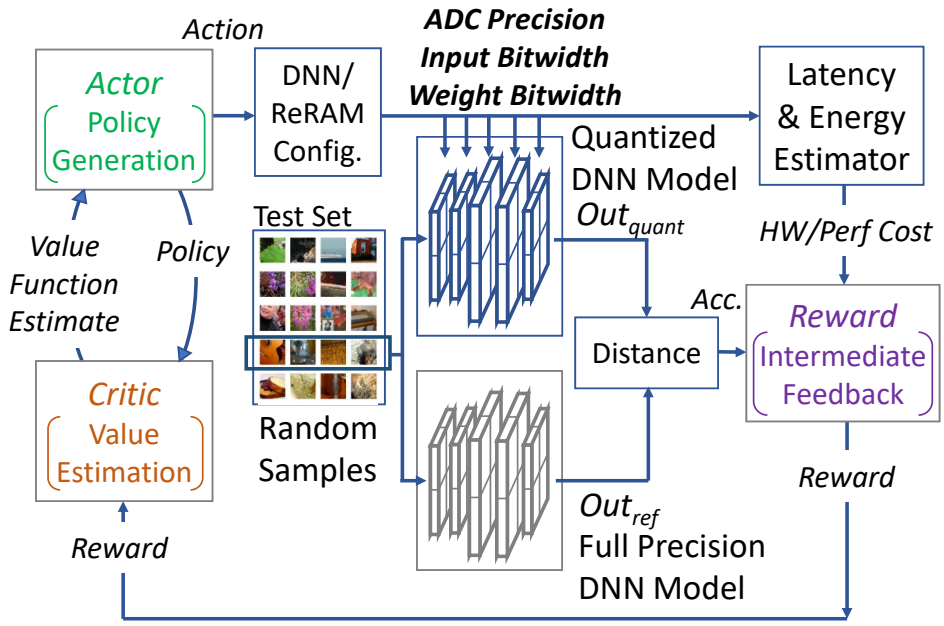


Figure 6.3: Reinforcement Learning based Mixed-Precision Quantization Flow

critic, and *reward*. The *actor* part generates new policies based on the value function estimates by calculating gradients of parameters. The generated policy is parsed by a DNN/ReRAM configurator which generates ADC precision and all the bitwidths for each layer in the DNN. The *critic* part reads reward from the environment and calcuates Q-value function of this MDP. The *reward* part takes in the loss measure collected from the quantized model and original model, as well as the DNN configuration, and calculates the reward for current state s and action a.

Quantized model evaluation components. The evaluation components consist of *test dataset*, *quantized DNN model*, *full precision DNN model*, and *loss measure*. *Test dataset* generates batches of random test samples. The batch size is configurable. *Quantized DNN model* and *full precision DNN model* take in the sample batch from the test dataset, and run model inference. Most likely the full precision DNN model will generate a smaller loss $loss_{\text{Ref}}$ than that of the quantized model $loss_{\text{Quant}}$. The flow calculates the differences $d = loss_{\text{Quant}} - loss_{\text{Ref}}$, and uses 4^d as the cost for the accuracy drop.

6.4 Methodology

6.4.1 Performance Simulation

To evaluate the impact of different quantization schemes on DNN inference energy consumption and latency, we use the PUMA [156] simulator, PUMAsim, which is a cycle-level architecture simulator for ReRAM-based accelerators. PUMAsim runs applications compiled with the PUMA compiler and provides detailed traces of execution. The simulator incorporates timing and power models of all system components. The simulator provides options for configuring various architectural parameters, including input and weight bit-width. We extend the simulator to also configure ADC resolution for evaluating partial sum quantization. We consider 2-bits per device for weight slicing and 1-bit per slice for input slicing, which are the typical parameters used in past crossbar-based accelerators [154, 156].

6.4.2 Functional Simulation

To evaluate the impact of different quantization schemes on DNN inference accuracy, we develop a functional simulator to simulate the arithmetic behavior of ReRAM-based accelerators which PUMAsim does not capture. Although several libraries, e.g., Distiller [159], Model Optimization Toolkit [160], etc., have been developed using TensorFlow and PyTorch to enable software and hardware co-design studies for quantization, such frameworks cannot emulate the precise implication of quantization on ReRAM-based accelerators because of the intrinsic differences between digital and ReRAM-based hardware. Digital accelerators typically express layer operations as general matrix-matrix multiplication, which use floating or fixed point computation units. On the other hand, ReRAM-based accelerators typically express layer operations as a tiled MVM which use bit-serial computation units operating in the analog domain (discussed in Section 6.2). For this reason, we develop our own functional simulator using PyTorch to analyze the impact of quantization of different aspects of crossbar hardware: weights, inputs, and partial sums. The functional simulator models the key phases of MVM computation in typical crossbar accelerators, namely iterative MVM, tiling, and bit-slicing, and ignores the memory and communication aspects which

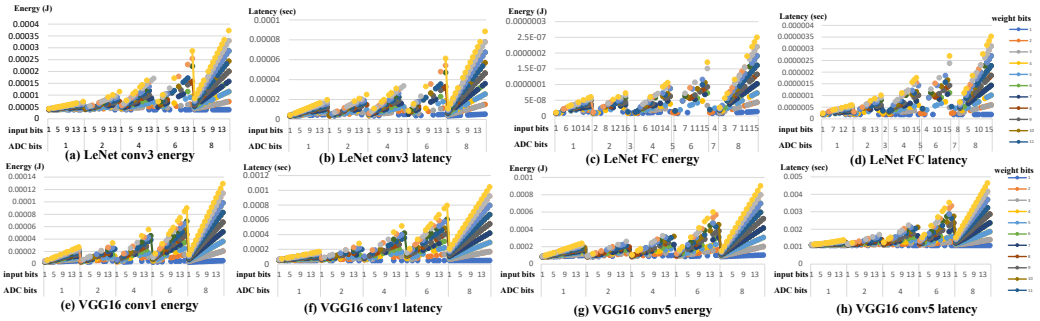
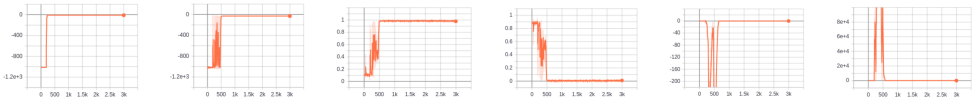
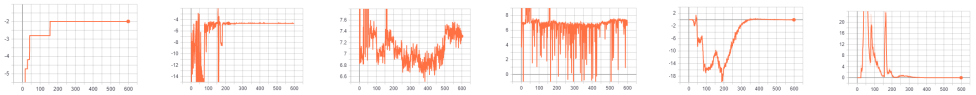


Figure 6.4: Energy and Latency under Quantization



(a) best reward | (b) last reward | (c) accuracy | (d) accuracy diff | (e) policy loss | (f) value loss



(g) best reward | (h) last reward | (i) quant error | (j) quant error diff | (k) policy loss | (l) value loss

Figure 6.5: Intermediate Values in Search (top: LeNet; buttom: VGG16)

are captured by PUMAsim.

6.4.3 Search Flow

We use the functional simulator to guide our search flow, and the performance simulator to calibrate our hardware cost model and evaluate the search result. We use prediction error instead of model accuracy as a metric to evaluate quantization accuracy. Prediction error requires fewer samples to estimate compared to model accuracy. We extract a small batch of random samples from the whole dataset, run the quantized DNN model with the functional simulator, and compare its outputs with outputs from the full-precision DNN model to get the predicion error.

6.5 Evaluation

6.5.1 Benefits of Mixed Precision Quantization

We use the performance simulator to evaluate the benefits of mixed precision quantization in ReRAM-based accelerators. We simulate all the combinations of quantization configurations of each layer in LeNet and VGG16, and collect energy and latency results of the simulated layer. Figures 6.4(a)-(d) show the energy and latency numbers of the third convolution layer and the fully connected layer in LeNet. Figures 6.4(e)-(h) show the energy and latency numbers of the first convolution layer and the fifth convolution layer in VGG16. As we can see from these figures, energy and latency numbers follow similar trends as bit-widths change. With the same ADC precision, energy and latency change linearly as input bitwidth and weight bitwidth change. With higher ADC precision, the energy and latency are more sensitive to changes in input bitwidth and weight bitwidth. Although Figure 6.4 shows regular patterns with a fixed ADC precision, the design space of weight bitwidth and input bitwidth combined with ADC precision is non-linear.

6.5.2 Quantization Search Flow

We use our proposed mixed precision quantization search flow to search for the optimal quantization schemes for LeNet and VGG16 that have the lowest energy and latency while not losing much accuracy. The intermediate reward and loss during the searches are shown in Figure 6.5. Figures 6.5(a)-(f) show the results for LeNet search, while Figures 6.5(g)-(l) show the results for VGG search. In our experiments, we ran LeNet search for 3,000 episodes, and VGG16 search for 600 episodes. However, note that LeNet search also converges within 600 episodes. As shown in the figure, search converges in both cases. At the beginning of the search, the RL agent takes random actions to explore the design space (*environment*). The agent learns about the environment at the same time. After warming up, the agent starts to apply the knowledge it learned into the search, and optimizes policy to improve the expected future rewards. At the end, the agent reports the best policy it discovered in the search, which translates to the best quantization configuration discovered. Table 6.3 lists the energy, latency, and accuracy of a

Table 6.3: LeNet Quantization Schemes

Quantization	$\boldsymbol{E}$ (μJ)	$\boldsymbol{T}$ (ms)	**Accuracy**
Q_{base}	850.99 (1.00×)	2.95 (1.00×)	97.27% (-0.00%)
Q_A	175.61 **(4.84×)**	0.76 **(3.89×)**	96.09% **(-1.18%)**
Q_B	229.82 (3.70×)	0.85 (3.48×)	96.29% (-0.98%)
Q_C	468.48 (1.82×)	1.69 (1.74×)	97.07% (-0.20%)

NOTE: Values in parentheses are savings compared to Q_{base}.
Q_A: (4, 16, 7), (4, 8, 8), (4, 8, 7), (4, 16, 8); Q_B: (16, 8, 8), (4, 8, 8), (8, 8, 8), (4, 8, 8);
Q_C: (16,16,6), (16,8,8), (4,8,7), (4,16,7); Q_{base}: (16,16,8),(16,16,8),(16,16,8),(16,16,8).

few quantization schemes for LeNet discovered by the search flow. Q_{base} is the baseline with full bit-widths. Each (i, w, a) tuple describes the bitwidths for inputs (i), weights (w), and ADC (a) for a layer. The four tuples correspond to quantization schemes for conv1, conv2, conv3, and fully connected layers in LeNet respectively. As the table shows, the discovered quantization scheme achieves up to 4.84× savings in energy and 3.89× savings in latency while only losing 1.18% accuracy, compared to Q_{base}. Schemes with even higher accuracy but less latency and energy savings are also discovered, e.g. Q_c. These design points reflect the tradeoffs between accuracy and performance (or resource), and provide different design options for different design requirements.

6.6 Related Work

Various works apply quantization to DNNs to improve the efficiency of their execution [143], [144], [145], [146], [147], [148], [149], [142], [150], [151]. ADMM-NN [161] is a framework that performs quantization and pruning jointly. Ares [162] is a framework for quantifying the resilience of DNNs to faults, and considers the impact of different quantization schemes on resilience. Choi et al. [163] reduce the mismatch between forward and backward passes when training networks that use quantization. Sakr et al. [164] propose quantization for back-propagation, not just inference. All these works focus on quantization of DNNs in general without architecture-specific considerations.

Bit fusion [165] and UNPU [166] provide architecture support for dynam-

ically reconfiguring bit width in DNN accelerators to support different levels of quantization. OLAccel [167] is an accelerator that enables better quality quantization by handling outliers separately. Various works that focus on quantization have also targeted FPGAs, such as REQ-YOLO [168] which focuses on FPGA resource awareness, and FINN [169] which specializes in binarized neural networks. All these works focus on digital accelerators, whereas our work focuses on ReRAM-based accelerators and the unique opportunities they provide.

Zhu et al. [170] provide a framework for quantizing CNNs on single-bit ReRAM crossbars. Zhang et al. [171] provide design guidelines for ReRAM-based DNN accelerators and include the impact of ADC quantization as part of their study. Our framework performs joint quantization of both weights and partial sums on two-bit crossbars based on deep reinforcement learning.

Various works propose ReRAM-based accelerators for DNN inference [152, 153, 154, 155, 156] and training [172, 173, 174, 175, 176]. Our work proposes a framework for quantization of DNNs to configure such accelerators. Other frameworks have also been proposed for transforming [177, 178] and pruning [179] DNNs for such accelerators.

6.7 Conclusion

We presented a mixed precision quantization scheme and an automated quantization flow for optimizing DNN inference on ReRAM-based accelerators. The flow uses deep reinforcement learning to find the best configuration of weight quantization, input quantization, and partial sum quantization across DNN layers. The evaluation shows that the quantization scheme enables more optimization opportunity and the automated quantization flow can effectively search for the best quantization configuration. The quantization configuration discovered by the search flow achieves up to 3.89× and 4.84× improvement over baseline without quantization in terms of inference latency and energy respectively, while only losing 1.18% in DNN inference accuracy.

CHAPTER 7

CONCLUSION

Modern computing faces new challenges arising from data complexity, application complexity, and hardware complexity. The overall compound complexity makes it harder and harder for traditional general-purpose processors to deliver desired computation performance and efficiency. These new challenges motivate people to create customized hardware acceleration systems that can deliver higher computation efficiency over traditional computing systems. However, computation efficiency is not the only metric to consider when designing hardware acceleration systems. To catch up with the fast growing complexity in applications and hardware platforms, people realize that agile hardware design and synthesis flows that can design and program hardware acceleration systems efficiently is in great need. Besides, depending on the computation needs of applications, these customized hardware acceleration systems might be highly heterogeneous platforms that contain CPUs, GPUs, FPGAs, ASICs, and other customized accelerators. How to program these highly heterogeneous platforms is another challenging problem to solve.

In this dissertation, we first looked into the key aspects of hardware accelerators, that is, efficiency, usability, and heterogeneity. The efficiency of hardware accelerator requires careful design and optimization. We show that the key to create high usability and programmability of accelerators is to use high-level programming abstractions and optimization flows. Then, we proposed a programming language and compiler based design flow, PyLog, that solves the agile hardware synthesis challenge mentioned above. This approach elevates the abstraction level of hardware programming and synthesis by providing a Python-based programming and synthesis environment for hardware acceleration systems. The high-level programming abstraction provided by PyLog not only simplifies users' hardware programming, but also creates additional opportunities for the compiler to perform code analysis and optimization. With PyLog high-level operators, users can express

computation pattern at a higher level and focus on algorithm specification. At the same time, these high-level operators open up more optimization opportunities for the compiler. Our evaluation showed that PyLog significantly improves FPGA design productivity with fewer lines of input source code, and it generates highly efficient FPGA designs that outperform highly optimized CPU implementation and state-of-the-art FPGA implementation by 3.17× and 1.24× on average. We also demonstrated how PyLog can be used to describe the collaborative computation patterns on heterogeneous computing platforms. Besides the current features supported by PyLog, there remain several promising directions for PyLog, including additional accelerator support, integration with current Python environment, memory optimizations, heterogeneous computing support, etc. We leave these as future works for PyLog.

REFERENCES

[1] Xilinx, "Vivado High-Level Synthesis," https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[2] Intel, "Intel High-Level Synthesis Compiler," https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[3] Xilinx, "Xilinx SDAccel Development Environment," https://www.xilinx.com/products/design-tools/software-zone/sdaccel.htm.

[4] Intel, "Intel FPGA SDK for OpenCL Software Technology," https://www.intel.com/content/www/us/en/software/programmable/ sdk-for-opencl/overview.html.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[7] X. Zhang, X. Liu, A. Ramachandran, C. Zhuge, S. Tang, P. Ouyang, Z. Cheng, K. Rupnow, and D. Chen, "High-performance video content recognition with long-term recurrent convolutional network for FPGA," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–4.

[8] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu, and D. Chen, "FPGA/DNN Co-Design: An efficient design methodology for IoT intelligence on the edge," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3316781.3317829 pp. 206:1–206:6.

[9] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi, "Sparse deep neural network graph challenge," *Graph Challenge*, 2019. [Online]. Available: https://graphchallenge.mit.edu/challenges

[10] S. Huang, L.-W. Chang, I. El Hajj, S. Garcia de Gonzalo, J. Gómez-Luna, S. R. Chalamalasetti, M. El-Hadedy, D. Milojicic, O. Mutlu, D. Chen, and W.-m. Hwu, "Analysis and modeling of collaborative execution strategies for heterogeneous CPU-FPGA architectures," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19. New York, NY, USA: ACM, 2019, pp. 79–90.

[11] "Xilinx Virtex-7 FPGA VC709 Connectivity Kit," https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html.

[12] J. Kepner, S. Alford, V. Gadepally, M. Jones, L. Milechin, R. Robinett, and S. Samsi, "Sparse deep neural network graph challenge," in *Graph Challenge*, 2019.

[13] "Bulldozer for servers: Testing AMD's "interlagos" opteron 6200 series," https://www.anandtech.com/show/5058/amds-opteron-interlagos-6200/8.

[14] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2014, pp. 36–43.

[15] H. Giefers, P. Staar, C. Bekas, and C. Hagleitner, "Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of GPU, Xeon Phi and FPGA," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 46–56.

[16] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang, "An efficient hardware accelerator for sparse convolutional neural networks on FPGAs," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2019, pp. 17–25.

[17] J. Chang, K. Kang, and S. Kang, "SDCNN: An efficient sparse deconvolutional neural network accelerator on FPGA," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019, pp. 968–971.

[18] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse LSTM on FPGA with bank-balanced sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3289602.3293898 pp. 63–72.

[19] E. Strohmaier, J. Dongarra, S. Horst, and M. Meuer, "Top500 List June 2018." [Online]. Available: https://www.top500.org/lists/2018/06/

[20] F. Wu and T. Scogland, "Green500 List June 2018." [Online]. Available: https://www.top500.org/green500/lists/2018/06/

[21] RightScale, "Rightscale 2018 state of the cloud report." [Online]. Available: https://assets.rightscale.com/uploads/pdfs/RightScale-2018-State-of-the-Cloud-Report.pdf

[22] Intel, "Intel FPGA SDK for OpenCL. Programming Guide," October 2016.

[23] Xilinx, "SDAccel Development Environment," https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[24] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *FPGA*, 2016.

[25] S. R. Chalamalasetti, M. Margala, W. Vanderbauwhede, M. Wright, and P. Ranganathan, "Evaluating FPGA-acceleration for real-time unstructured search," in *ISPASS*, 2012.

[26] D. Chen, J. Cong, Y. Fan, and L. Wan, "LOPASS: A low-power architectural synthesis system for FPGAs with interconnect estimation and optimization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2010.

[27] "Amazon EC2 F1 instances," https://aws.amazon.com/ec2/instance-types/f1/, 2018.

[28] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *MICRO*, 2016.

[29] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA*, 2014.

[30] "New OpenPOWER cloud boosts ecosystem for innovation and development," http://www-03.ibm.com/press/us/en/pressrelease/47082.wss, 2015.

[31] Intel, "Intel Deep Learning Inference Accelerator Product Specification and User's Guide," https://www.intel.com/content/dam/support/us/en/documents/server-products/server-accessories/Intel_DLIA_UserGuide_1.0.pdf, July 2017.

[32] "The first chip from Intel's Altera buy will be out in 2016," http://fortune.com/2015/11/18/intel-xeon-fpga-chips/, 2015.

[33] D. Burger, "Microsoft unveils Project Brainwave for real-time AI," *Microsoft Research*, 2017.

[34] K. Rupnow, Y. Liang, Y. Li, D. Min, M. Do, and D. Chen, "High level synthesis of stereo matching: Productivity, performance, and software constraints," in *FPT*, 2011.

[35] S. Liu, A. Papakonstantinou, H. Wang, and D. Chen, "Real-time object tracking system on FPGAs," in *SAAHPC*, 2011.

[36] S. Huang, G. J. Manikandan, A. Ramachandran, K. Rupnow, W.-m. W. Hwu, and D. Chen, "Hardware acceleration of the Pair-HMM algorithm for dna variant calling," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021749 pp. 275–284.

[37] X. Zhang, X. Liu, A. Ramachandran, C. Zhuge, S. Tang, P. Ouyang, Z. Cheng, K. Rupnow, and D. Chen, "High-performance video content recognition with long-term recurrent convolutional network for FPGA," in *FPL*, 2017.

[38] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek, "Intel® quickpath interconnect architectural features supporting scalable system architectures," in *HOTI*, 2010.

[39] H. T. Consortium et al., "Hypertransport i/o link specification," *Revision*, vol. 1, pp. 111–118, 2008.

[40] Altera, "Accelerating High-Performance Computing With FPGAs," https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01029.pdf.

[41] "Accelerator Coherency Port," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0434a/BABGHDHD.html.

[42] "AXI Coherency Extensions," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0438i/BABIAFAJ.html.

[43] "Arm CoreLink Interconnect," https://developer.arm.com/products/system-ip/corelink-interconnect.

[44] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel, "CAPI: A coherent accelerator processor interface," *IBM J. Res. Dev.*, vol. 59, no. 1, p. 7:1–7:7, Jan. 2015. [Online]. Available: https://doi.org/10.1147/JRD.2014.2380198

[45] "Cache Coherent Interconnect for Accelerators (CCIX)," http://www.ccixconsortium.com, 2016.

[46] Xilinx, "Zynq UltraScale+ MPSoCs. White Paper," June 2016.

[47] Altera, "Altera's User-Customizable ARM-Based SoC," 2015.

[48] M. Hummel, M. Krause, and D. O'Flaherty, "AMD and HP: Protocol enhancements for tightly coupled accelerators," *AMD-HP whitepaper*, 2007.

[49] W. Augustin, V. Heuveline, and J.-P. Weiss, "Convey HC-1 – the potential of FPGAs in numerical simulation," *Preprint Series of the Engineering Mathematics and Computing Lab*, no. 07, 2010.

[50] Convey Computer, "The Convey HC-2 computer. Architectural overview," 2012.

[51] J. Gómez-Luna, I. El Hajj, L.-W. Chang, V. Garcia-Flores, S. Garcia de Gonzalo, T. Jablin, A. J. Pena, and W.-m. Hwu, "Chai: Collaborative heterogeneous applications for integrated-architectures," in *ISPASS*, 2017.

[52] Terasic, *DE5-Net User Manual*, 2018.

[53] Nallatech, "510T FPGA Accelerator Card Datasheet."

[54] Intel, "Intel Stratix V FPGAs," https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-v.html.

[55] Intel, "Intel Arria 10 FPGAs," https://www.intel.com/content/www/us/en/products/programmable/fpga/arria-10.html.

[56] Intel, "Intel Xeon Processor E3-1240 v3," https://ark.intel.com/products/75055/Intel-Xeon-Processor-E3-1240-v3-8M-Cache-3-40-GHz-.

[57] Intel, "Intel Xeon Processor E5-2650 v3," https://ark.intel.com/products/81705/Intel-Xeon-Processor-E5-2650-v3-25M-Cache-2-30-GHz-.

[58] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1986.

[59] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, 1981.

[60] J. Gómez-Luna, H. Endt, W. Stechele, J. M. González-Linares, J. I. Benavides, and N. Guil, "Egomotion compensation and moving objects detection algorithm on GPU," in *PARCO*, 2011.

[61] R. Palomar, J. Gómez-Luna, F. A. Cheikh, J. Olivares-Bueno, and O. J. Elle, "High-performance computation of Bézier surfaces on parallel and heterogeneous platforms," *International Journal of Parallel Programming*, 2018.

[62] J. Gómez-Luna, J. González-Linares, J. Benavides, and N. Guil, "An optimized approach to histogram computation on GPU," *Machine Vision and Applications*, 2013.

[63] J. Gómez-Luna, J. Gónzalez-Linares, J. Benavides, and N. Guil, "Performance modeling of atomic additions on GPU scratchpad memory," *IEEE Transactions on Parallel and Distributed Systems*, 2013.

[64] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *USENIX SECURITY*, 2007.

[65] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory access scheduling for chip multiprocessors," in *MICRO*, 2007.

[66] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *HPCA*, 2013.

[67] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The Application Slowdown Model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *MICRO*, 2015.

[68] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory scheduling: Exploiting differences in memory access behavior," in *MICRO*, 2010.

[69] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA*, 2010.

[70] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing both performance and fairness of shared DRAM systems," in *ISCA*, 2008.

[71] Khronos group, "The OpenCL specification," *Version 2.0*, 2015.

[72] M. Gupta, D. Das, P. Raghavendra, T. Tye, L. Lobachev, A. Agarwal, and R. Hegde, "Implementing cross-device atomics in heterogeneous processors," in *IPDPS Workshops*, 2015.

[73] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A quantitative analysis on microarchitectures of modern CPU-FPGA platforms," in *DAC*, 2016.

[74] G. Weisz, J. Melber, Y. Wang, K. Fleming, E. Nurvitadhi, and J. C. Hoe, "A study of pointer-chasing performance on shared-memory processor-FPGA systems," in *FPGA*, 2016.

[75] M.-C. F. Chang, Y.-T. Chen, J. Cong, P.-T. Huang, C.-L. Kuo, and C. H. Yu, "The SMEM seeding acceleration for DNA sequence alignment," in *FCCM*, 2016.

[76] Z. István, D. Sidler, and G. Alonso, "Runtime parameterizable regular expression operators for databases," in *FCCM*, 2016.

[77] C. Zhang, R. Chen, and V. Prasanna, "High throughput large scale sorting on a CPU-FPGA heterogeneous platform," in *IPDPS*, 2016.

[78] W. Qiao, J. Du, Z. Fang, M. Lo, M.-C. F. Chang, and J. Cong, "High-throughput lossless compression on tightly coupled CPU-FPGA platforms," in *FPGA*, 2018.

[79] D. Sidler, Z. István, M. Owaida, and G. Alonso, "Accelerating pattern matching queries in hybrid CPU-FPGA architectures," in *SIGMOD*, 2017.

[80] H. Schmit and R. Huang, "Dissecting Xeon+FPGA: Why the integration of CPUs and FPGAs makes a power difference for the datacenter," in *ISLPED*, 2016.

[81] N. Chandramoorthy, G. Tagliavini, K. Irick, A. Pullini, S. Advani, S. A. Habsi, M. Cotter, J. Sampson, V. Narayanan, and L. Benini, "Exploring architectural heterogeneity in intelligent vision systems," in *HPCA*, 2015.

[82] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, H. Huang, and G. Reinman, "Composable accelerator-rich microprocessor enhanced for adaptivity and longevity," in *ISLPED*, 2013.

[83] E. G. Cota, P. Mantovani, G. D. Guglielmo, and L. P. Carloni, "An analysis of accelerator coupling in heterogeneous architectures," in *DAC*, 2015.

[84] H. Usui, L. Subramanian, K. Chang, and O. Mutlu, "DASH: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators," *ACM TACO*, 2016.

[85] G. Ndu, J. Navaridas, and M. Luján, "CHO: Towards a benchmark suite for OpenCL FPGA accelerators," in *IWOCL*, 2015.

[86] V. Anshuman, A. E. Helal, K. Krommydas, and W.-c. Feng, "Accelerating workloads on FPGAs via OpenCL: A case study with OpenDwarfs," *Virginia Tech CS Tech. Rep.*, 2016.

[87] N. Ramanathan, J. Wickerson, F. Winterstein, and G. A. Constantinides, "A case for work-stealing on FPGAs with OpenCL atomics," in *FPGA*, 2016.

[88] Z. Wang, B. He, W. Zhang, and S. Jiang, "A performance analysis framework for optimizing OpenCL applications on FPGAs," in *HPCA*, 2016.

[89] S. Sridharan, P. Durante, C. Faerber, and N. Neufeld, "Accelerating particle identification for high-speed data-filtering using OpenCL on FPGAs and other architectures," in *FPL*, 2016.

[90] Z. Wang, J. Paul, H. Y. Cheah, B. He, and W. Zhang, "Relational query processing on OpenCL-based FPGAs," in *FPL*, 2016.

[91] J. Shen, A. L. Varbanescu, Y. Lu, P. Zou, and H. Sips, "Workload partitioning for accelerating applications on heterogeneous platforms," *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[92] C. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO*, 2009.

[93] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. McCardwell, A. Villegas, and D. Kaeli, "Hetero-mark, a benchmark suite for CPU-GPU collaborative computing," in *IISWC*, 2016.

[94] S. Mukherjee, Y. Sun, P. Blinzer, A. K. Ziabari, and D. Kaeli, "A comprehensive performance analysis of HSA and OpenCL 2.0," in *ISPASS*, 2016.

[95] S. Mukherjee, X. Gong, L. Yu, C. McCardwell, Y. Ukidave, T. Dao, F. N. Paravecino, and D. Kaeli, "Exploring the features of OpenCL 2.0," in *IWOCL*, 2015.

[96] L.-W. Chang, J. Gómez-Luna, I. El Hajj, S. Huang, D. Chen, and W.-m. Hwu, "Collaborative computing for heterogeneous integrated systems," in *ICPE*, 2017.

[97] M. D. Sinclair, J. Alsop, and S. V. Adve, "HeteroSync: A benchmark suite for fine-grained synchronization on tightly coupled GPUs," in *IISWC*, 2017.

[98] Y. Sun, S. Mukherjee, T. Baruah, S. Dong, J. Gutierrez, P. Mohan, and D. Kaeli, "Evaluating performance tradeoffs on the radeon open compute platform," in *ISPASS*, 2018.

[99] J. Gómez-Luna, I.-J. Sung, A. Lázaro-Muñoz, W.-H. Chung, J. González-Linares, and N. Guil, "Chapter 8 - Application use cases: Platform atomics," in *Heterogeneous System Architecture*, 2016.

[100] W.-m. W. Hwu, *Heterogeneous System Architecture: A New Compute Platform Infrastructure*. Morgan Kaufman, 2015.

[101] S. Che, M. Orr, and J. Gallmeier, "Work stealing in a shared virtual-memory heterogeneous environment: A case study with betweenness centrality," in *CF*, 2017.

[102] S. Tang, B. He, S. Zhang, and Z. Niu, "Elastic multi-resource fairness: balancing fairness and efficiency in coupled CPU-GPU architectures," in *SC*, 2016.

[103] F. Zhang, B. Wu, J. Zhai, B. He, and W. Chen, "Finepar: Irregularity-aware fine-grained workload partitioning on integrated architectures," in *CGO*, 2017.

[104] Y. Cho, F. Negele, S. Park, B. Egger, and T. R. Gross, "On-the-fly workload partitioning for integrated CPU/GPU architectures," in *PACT*, 2018.

[105] T. Baruah, Y. Sun, S. Dong, D. Kaeli, and N. Rubin, "Airavat: Improving energy efficiency of heterogeneous applications," in *DATE*, 2018.

[106] T. Baruah, "Energy efficient execution of heterogeneous applications. Master thesis. Northeastern University," 2017.

[107] A. Patil and R. Govindarajan, "HAShCache: Heterogeneity-aware shared DRAMCache for integrated heterogeneous systems," *ACM TACO*, 2017.

[108] V. Garcia-Flores, J. Gómez-Luna, T. Grass, A. Rico, E. Ayguade, and A. J. Peña, "Evaluating the effect of last-level cache sharing on integrated GPU-CPU systems with heterogeneous applications," in *IISWC*, 2016.

[109] R. Ausavarungnirun, K. Chang, L. Subramanian, G. Loh, and O. Mutlu, "Staged Memory Scheduling: Achieving high performance and scalability in heterogeneous systems," in *ISCA*, 2012.

[110] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing GPU concurrency in heterogeneous architectures," in *MICRO*, 2014.

[111] V. Garcia-Flores, E. Ayguade, and A. J. Peña, "Efficient data sharing on heterogeneous systems," in *ICPP*, 2017.

[112] J. Alsop, M. D. Sinclair, and S. V. Adve, "Spandex: a flexible interface for efficient heterogeneous coherence," in *ISCA*, 2018.

[113] J. Veselỳ, A. Basu, A. Bhattacharjee, G. H. Loh, M. Oskin, and S. K. Reinhardt, "Generic system calls for GPUs," in *ISCA*, 2018.

[114] A. Basu, J. L. Greathouse, G. Venkataramani, and J. Veselỳ, "Interference from GPU system service requests," in *IISWC*, 2018.

[115] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, no. 2, pp. 21–29, Mar 2018.

[116] arXiv, "arXiv.org e-Print archive," https://arxiv.org/.

[117] Xilinx, "CHaiDNN: HLS based deep neural network accelerator library for Xilinx UltraScale+ MPSoCs," https://github.com/Xilinx/CHaiDNN.

[118] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA'19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3289602.3293910 pp. 242–251.

[119] TVM, "TVM stack," https://tvm.apache.org/.

[120] F. Computing, "Merlin compiler," https://www. falconcomputing. com/ merlin-fpga-compiler/.

[121] R. Nigam, S. Atapattu, S. Thomas, Z. Li, T. Bauer, Y. Ye, A. Koti, A. Sampson, and Z. Zhang, "Predictable accelerator design with time-sensitive affine types," 2020.

[122] Chisel, "Chisel/FIRRTL hardware compiler framework," https:// www.chisel-lang.org/.

[123] Clash, "Clash: A modern, functional, hardware description language," https://clash-lang.org/.

[124] PyMTL3, "PyMTL3 (Mamba), an open-source python-based hardware generation, simulation, and verification framework," https:// github.com/pymtl/pymtl3.

[125] PyRTL, "PyRTL," https://ucsbarchlab.github.io/PyRTL/.

[126] Y. Chi, J. Cong, P. Wei, and P. Zhou, "Soda: Stencil with optimized dataflow architecture," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[127] J. Lau, A. Sivaraman, Q. Zhang, M. A. Gulzar, J. Cong, and M. Kim, "Heterorefactor: Refactoring for heterogeneous computing with fpga," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3377811.3380340 p. 493–505.

[128] F. Winterstein, S. Bayliss, and G. A. Constantinides, "High-level synthesis of dynamic data structures: A case study using vivado hls," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 362–365.

[129] Z. Xue and D. B. Thomas, "Synadt: Dynamic data structures in high level synthesis," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 64–71.

[130] D. Diamantopoulos, S. Xydis, K. Siozios, and D. Soudris, "Dynamic memory management in vivado-hls for scalable many-accelerator architectures," in *Applied Reconfigurable Computing*, K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, Eds. Cham: Springer International Publishing, 2015, pp. 117–128.

[131] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "Performance modeling and directives optimization for high level synthesis on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[132] "Pynq," http://www.pynq.io/.

[133] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–10.

[134] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel programming for FPGAs," 2018.

[135] "Zedboard," http://zedboard.org/product/zedboard.

[136] "PYNQ-Z1: Python productivity for Zynq-7000 ARM/FPGA SoC," https://store.digilentinc.com/pynq-z1-python-productivity-for-zynq-7000-arm-fpga-soc/.

[137] "Ultra96 board," https://www.96boards.org/product/ultra96/.

[138] "Amazon EC2 F1 instances," https://aws.amazon.com/ec2/instance-types/f1/.

[139] "Xilinx Alveo boards," https://www.xilinx.com/products/boards-and-kits/alveo.html.

[140] E. Schkufza, M. Wei, and C. J. Rossbach, "Just-in-time compilation for verilog: A new technique for improving the fpga programming experience," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3297858.3304010 p. 271–286.

[141] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2072–2085, 2019.

[142] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu, and D. Chen, "FPGA/DNN co-design: An efficient design methodology for iot intelligence on the edge," in *DAC*, New York, NY, USA, 2019. [Online]. Available: https://doi.org/10.1145/3316781.3317829

[143] P. Judd, J. Albericio, T. Hetherington, T. Aamodt, N. E. Jerger, R. Urtasun, and A. Moshovos, "Reduced-precision strategies for bounded memory in deep neural nets," *arXiv preprint arXiv:1511.05236*, 2015.

[144] B. Moons, B. De Brabandere, L. Van Gool, and M. Verhelst, "Energy-efficient convnets through approximate computing," in *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 2016, pp. 1–8.

[145] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, 2016, pp. 2849–2858.

[146] C. Sakr, Y. Kim, and N. Shanbhag, "Analytical guarantees on numerical precision of deep neural networks," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 3007–3016.

[147] L. Hou and J. T. Kwok, "Loss-aware weight quantization of deep networks," *arXiv preprint arXiv:1802.08635*, 2018.

[148] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 8612–8620.

[149] J. Wang, Q. Lou, X. Zhang, C. Zhu, Y. Lin, and D. Chen, "Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA," in *FPL*, 2018. [Online]. Available: https://doi.org/10.1109/FPL.2018.00035 pp. 163–169.

[150] Y. Li, C. Hao, X. Zhang, X. Liu, Y. Chen, J. Xiong, W.-m. Hwu, and D. Chen, "EDD: Efficient differentiable DNN architecture and implementation co-search for embedded ai solutions," in *DAC*. IEEE Press, 2020.

[151] C. Gong, Y. Chen, Y. Lu, T. Li, C. Hao, and D. Chen, "VecQ: Minimal loss DNN model compression with vectorized weight quantization," *IEEE Transactions on Computers*, vol. 70, no. 05, pp. 696–710, May 2021.

[152] X. Liu, M. Mao, B. Liu, H. Li, Y. Chen, B. Li, Y. Wang, H. Jiang, M. Barnell, Q. Wu et al., "RENO: A high-efficient reconfigurable neuromorphic computing accelerator design," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*. IEEE, 2015, pp. 1–6.

[153] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA'16. Piscataway, NJ, USA: IEEE Press, 2016. [Online]. Available: https://doi.org/10.1109/ISCA.2016.13 pp. 27–39.

[154] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA'16. IEEE Press, 2016, pp. 14–26.

[155] B. Feinberg, S. Wang, and E. Ipek, "Making memristive neural network accelerators reliable," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 52–65.

[156] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy et al., "Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 715–731.

[157] I. C. et al, "GENIEx: A Generalized Approach to Emulating Non-Idealities in Memristive X-bars Using Neural Networks," in *DAC*, 2020.

[158] M. J. Marinella, S. Agarwal, A. Hsia, I. Richter, R. Jacobs-Gedrim, J. Niroula, S. J. Plimpton, E. Ipek, and C. D. James, "Multiscale co-design analysis of energy, latency, area, and accuracy of a ReRAM analog neural training accelerator," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 1, pp. 86–101, 2018.

[159] N. Zmora et al., "Neural network distiller," June 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1297430

[160] TensorFlow, "Model optimization toolkit." [Online]. Available: https://www.tensorflow.org/model_optimization

[161] A. Ren, T. Zhang, S. Ye, J. Li, W. Xu, X. Qian, X. Lin, and Y. Wang, "Admm-nn: An algorithm-hardware co-design framework of dnns using alternating direction methods of multipliers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 925–938.

[162] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei, "Ares: A framework for quantifying the resilience of deep neural networks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[163] Y. Choi, M. El-Khamy, and J. Lee, "Learning low precision deep neural networks through regularization," *arXiv preprint arXiv:1809.00095*, 2018.

[164] C. Sakr and N. R. Shanbhag, "Per-tensor fixed-point quantization of the back-propagation algorithm," in *7th International Conference on Learning Representations, ICLR 2019*, 2019.

[165] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 764–775.

[166] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "Unpu: A 50.6 tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision," in *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2018, pp. 218–220.

[167] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 688–698.

[168] C. Ding, S. Wang, N. Liu, K. Xu, Y. Wang, and Y. Liang, "REQ-YOLO: A resource-aware, efficient quantization framework for object detection on fpgas," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 33–42.

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

[170] Z. Zhu, H. Sun, Y. Lin, G. Dai, L. Xia, S. Han, Y. Wang, and H. Yang, "A configurable multi-precision cnn computing framework based on single bit rram," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[171] W. Zhang, X. Peng, H. Wu, B. Gao, H. He, Y. Zhang, S. Yu, and H. Qian, "Design guidelines of rram based neural-processing-unit: A joint device-circuit-algorithm analysis," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[172] M. N. Bojnordi and E. Ipek, "Memristive Boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 1–13.

[173] M. Cheng, L. Xia, Z. Zhu, Y. Cai, Y. Xie, Y. Wang, and H. Yang, "Time: A training-in-memory architecture for memristor-based deep neural networks," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 26.

[174] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined ReRAM-based accelerator for deep learning," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 541–552.

[175] F. Chen, L. Song, and Y. Chen, "ReGAN: A pipelined ReRAM-based accelerator for generative adversarial networks," in *Design Automation Conference (ASP-DAC), 2018 23rd Asia and South Pacific*. IEEE, 2018, pp. 178–183.

[176] A. Ankit, I. El Hajj, S. Chalamalasetti, S. Agarwal, M. Marinella, M. Foltin, J. P. Strachan, D. Milojicic, W.-m. Hwu, and K. Roy, "Panther: A programmable architecture for neural network training harnessing energy-efficient reram," *IEEE Transactions on Computers*, 2020.

[177] Y. Ji, Y. Zhang, S. Li, P. Chi, C. Jiang, P. Qu, Y. Xie, and W. Chen, "Neutrams: Neural network transformation and co-design under neuromorphic hardware constraints," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 21.

[178] Y. Ji, Y. Zhang, W. Chen, and Y. Xie, "Bridge the gap between neural networks and neuromorphic hardware with a neural network compiler," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 448–460.

[179] Y. Wang, W. Wen, B. Liu, D. Chiarulli, and H. H. Li, "Group scissor: Scaling neuromorphic computing design to large neural networks," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 85.